Final Project Report

SOFTWARE MEASUREMENT (SOEN 6611)

Summer 2019

By

Team G

Project github repository - https://github.com/ahmedmaria/Software-Measurement

Name	Student_Id	Email
Maria Ahmed	40070844	ritu329@gmail.com
Shehnaz Ansari	40070137	shehnaz1191@gmail.com
Samaneh Shirdel Farimani	40075615	Samaneh.shirdelk@gmail.com
Mahy Salama	40074737	mahyhanysalama@gmail.com
Gagandeep Jaswal	40066862	gagandeepjaswal227@gmail.com

Correlation analysis among software measurement metrics on open source projects

SOEN 6611 Summer 2019 – Project Report

Maria Ahmed
dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
ritu329@gmail.com

Shehnaz Ansari
dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
shehnaz1191@gmail.com

Samaneh Shirdel Farimani
dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
samaneh.shirdelk@gmail.com

Mahy Salama
dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
mahyhanysalama@gmail.com

Gagandeep Jaswal
dept of Software Engineering and Computer Science
Concordia University
Montreal, Canada
gagandeepjaswal227@gmail.com

***Abstract*— Software quality is defined as a field of study and practice that describes the desirable attributes of software products. Testing is required for an effective performance of software application or product. We test different metrics in a software to evaluate its quality and functionality and it is particularly useful when deciding the maintenance operations that we should apply to the software. There are different tools to measure various metrics of the software. In our study, we analyzed 5 open-source Java projects in order to measure test coverage, McCabe Complexity, Relative Code Churn and Post Release Defect Density and investigated the correlation among them.**

Keywords—Test-suite effectiveness, Relative code churn, Post-release defect density, Code coverage, McCabe complexity, Mutation testing

I. INTRODUCTION

Software developers strives to make softwares more maintainable to make it secure and reliable to use, also modifiable and testable for evolution. Software maintenance is expensive and time consuming. There are many factors which can affect software maintainability and overall quality and software metrics helps us to analyze the performance of the system in real world.

The environment in which an open source software is being developed is very different than the environment of a closed sourced software. Formal definition of software process or a defined organization culture is absent which makes maintaining software quality more difficult. Therefore, software measurement has become significant in recent years. For this project, we have analyzed six different software metrics (Statement coverage, Branch coverage, Mutation score, McCabe complexity, relative code churn, Post-release defect density) from five open source softwares and tried to find the correlation among these metrics.

II. ORGANISATION

The report is organized in the following format-

1. Related work – work done on these metrics previously
2. Hypotheses – Hypotheses that we want to prove
3. Description of softwares – Five open source softwares that are used for the project are described in this section
4. Metric description – General description of the metrics used in this project
5. Data collection – the procedure followed to collect data
6. Data analysis – steps to analyze data
7. Threats to validity
8. Results

III. RELATED WORK

A. An empirical study on object-oriented metrics [1]: This study, conducted on three industrial real-time programs, has two parts; the first part of the study is the validation of CK metrics on these programs. Then, through the observations obtained from the first part of the study, in the second part, they identify a set of new metrics that might better serve their needs. In this study, object-oriented metrics have been studied and proposed as good predictors for fault-prone modules/classes, program maintainability, and software productivity. This empirical study is aimed at identifying object-oriented metrics that can be utilized to characterize the degree of object-orientation that an object-oriented program contains so that the likelihood of object-oriented faults occurring can be estimated. In addition to 5 CK metrics they also added new metrics which are: Inheritance Coupling (IC) - Coupling Between Methods: (CBM) - Number of Object/Memory Allocation: (NOMA) - Average Method Complexity: (AMC).

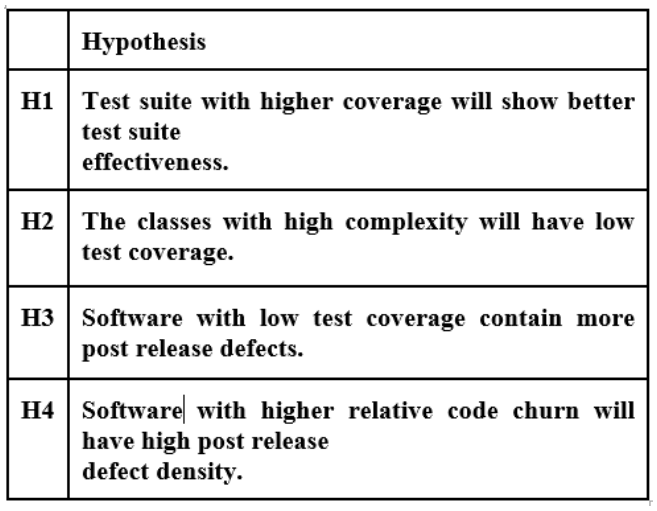
B. Program State Coverage: A Test Coverage Metric Based on Executed Program States [2]: they discussed that in software testing, different metrics are proposed to predict and compare test suites effectiveness and Mutation Score (MS) is one of the most accurate metrics. They propose a novel test coverage metric, called Program State Coverage (PSC), which improves the accuracy of LC. PSC works almost the same as LC and it can be calculated by executing test suites only once. However, it further considers the number of distinct program states in which each line is executed. Their experiments on 120 test suites from four packages of Apache Commons Math and Apache Commons Lang showed that, compared to LC, PSC is more strongly correlated with normalized MS. As a result, they conclude that PSC is a promising test coverage metric. Also, the author found that statement coverage is good for test suite effectiveness.

C. N. Nagappan et.al stated in the paper that instead of absolute metrics relative code churns is excellent predictors of post-release defect density on real world large industrial software systems. They conducted experiment on Windows Server 2003 SP 1. They used 8 relative code churn metrics using normalization and creating variants such as the churns in file, code added, removed and modified etc. and intern using spearman ranking correlation proved with the increase in code churns the defects in the systems also increase. In this paper, they proposed seven different measures to calculate code churn. [4]

D. Daskalantonakis (1992) described Motorola's software metrics program in his paper "A Practical View of Software Measurement and Implementation Experiences Within Motorola". Motorola used Goal/Question/Metric paradigm and one of the goals was to "Decrease software defect density" and established metric was defect density [8]. Post release defect density can be reliable metric to measure software quality. Also, According to Rahmani and Khazanchi defect density is a promising metric to address quality issue in an open source software system. Rahmani and Khazanchi analysed forty-four randomly selected open source software projects retrieved from SourceForge.net and conclude that there is no statistically, significant linear relationship between KLOC and software defect density [9]

IV. Hypotheses

The goal of our study is to investigate the correlation between a) statement coverage, branch coverage and mutation score b) statement coverage, branch coverage and McCabe complexity c) statement coverage, branch coverage and Post-release defect density d) Relative Code churn and Post-release defect density. In this project, we will try to prove these hypotheses with data we collected.



	Hypothesis
H1	**Test suite with higher coverage will show better test suite effectiveness.**
H2	**The classes with high complexity will have low test coverage.**
H3	**Software with low test coverage contain more post release defects.**
H4	**Software with higher relative code churn will have high post release defect density.**

V. Description of softwares

We selected five open source softwares, mostly written in Java, based on criteria mentioned in the following table. All of the softwares have public github repository and accessible Jira issuetracker.

Project	Versions	SLOC (of latest version)	Last commit till June 2019
Apache Commons Configurations	2.4,2.3,2.2, 2.1,2.0	847K	Over one year
Apache Commons Math	3.5,3.4, 3.3, 3.2, 3.1	186K	6 days
Apache Commons Codec	1.11,1.10,1.9,1.8,1.7	22.1K	7 months
Apache Commons Collections	4.2,4.1, 4.0, 3.2,3.1	132K	9 months
Apache Commons text	1.6, 1.5, 1.4, 1.3, 1.2	24.4K	28 days

VI. METRIC DESCRIPTION

Test coverage: Test coverage is a measure which describes how much of the code was executed when a particular test suit was executed. A high-test coverage means that most of the code were executed my test cases so general assumption is that the codes will contain less defects as those were tested. We choose statement and branch coverage metrics as our test coverage metric.

Statement and branch coverage are both white box testing technique. Statement coverage, as the name suggests, considers all the executed statements by the test suit. Branch coverage, also known as decision coverage, checks all the conditions or decisions of the source code. Difference between these two metrics is that statement coverage does not check false condition but branch coverage does. Also 100% branch coverage means 100% statement coverage but not the other way around.

Test suite effectiveness: While testing a software it is important to know what kind of test cases are required to test the defects that may occur. Test suit effectiveness measures the quality of the test suit being used for the software. We choose mutation score as our test suite effectiveness metric.

In Mutation testing certain code is changed to get a mutant version of the software and then same test suite is applied in both mutant and original. If the final result is same for both, the mutant is not killed but if it gives different result the mutant is considered to be killed by the test suit. Mutation score can be used to design effective test cases. Important condition to note while designing the test suit is every mutant only have one particular code change one mutant does not contain multiple code changes.

Complexity Metric: Complexity Metric is a measure to indicate how complex a code is. Complex code is difficult to understand as well as maintain. Also, there is a high chance of inducing defects if the system is complex. We choose McCabe Cyclomatic complexity as our complexity metric. McCabe cyclomatic complexity is defined as "A quantitative measure of the number of linearly independent paths through a program's source code…computed using the control flow graph of the program." [3]

McCabe Complexity can be calculated by following two formulas.

*McCabe Complexity = No. of Edges - No. of Vertices + 2 * Connected components*

McCabe Complexity = No. of Control Predicates + 1

As we calculated McCabe Complexity from the report generated by jacoco. We added missed and covered complexity to get the total McCabe complexity by class.

McCabe complexity = Missed complexity + Covered complexity.

Software maintenance effort: Maintaining a software can be daunting task. There are some metrics that gives us the measure about software maintenance effort. We choose relative code churn as our metric for maintenance effort.

Relative code churn = $\frac{\text{Churned Code}}{\text{Size of the Software}}$

Code Churn is a measure of the amount of the code changed within software over a period of time. It is found by checking the change history of a system which is recorded by version control systems. The version Control systems use a comparison technique to find the number of lines added, changed or deleted by programmer by comparing the previous version of the system to new version. The results obtained are used to measure code Churn. [4] There are multiple different ways to calculate code churn to get relative
code churn. The measured use to calculate churned code for this project are discussed in detail in "Data Analysis" part of the report. Source line of code (KSLOC) is used to measure size of the software.

Software quality attribute: A lot of factors can be responsible for effecting a software's overall quality. The
environment in which an open source software is being developed is very different than the environment of a closed sourced software. Formal definition of software process or a defined organization culture is absent so maintaining software quality is necessary to achieve reliability. A way of achieving quality is to minimize number of faults(defects). According to Rahmani and Khazanchi defect density is a promising metric to address quality issue in an open source software system [7] Defect density is a measure of the total known defects divided by the size of the software entity being measured [6].

Defect density = $\frac{\text{Number of known defects}}{\text{Size of the Software(KSLOC)}}$

A defect can be thought of as an occurred failure in the system. As we will collect data from open source software instead of defect density, we are using post release defect density. We will collect data for
number of known defects using available issue tracker or bug tracker of the open source software.

Software size can be measured using source line of code (LOC) or source line of code (SLOC). For our project we aim to use Thousands of source line of code (KSLOC).

$$Post\ release\ defect\ density = \frac{\text{Number of resolved bugs in a version}}{\text{size of software for that version}}$$

How metrics are calculated are described in detail in "Data Analysis" part.

VII. DATA COLLECTION

Code coverage and Branch Coverage:
we have used Jacoco which is a free code coverage

library for java. Standard JVM Tool interface is used by jacoco. Jacoco agent attaches itself to a JVM during a build. After JVM starts and a class is loaded jacoco can uses agent to get information about which class is loaded and which lines are executed. Report with code coverage statistic is created and published in the directory/target/site/jacoco folder when a JVM terminates. Jacoco uses given color scheme to describe code coverage->Green – fully covered lines, Yellow – party covered lines (some instructions or branches missed), Red – lines that have not been executed at all [11]. We added jacoco as a plug in under build in pox.xml file.

After adding the plugin, we had to run maven clean and install command to generate the html and csv reports.

We considered the code and branch coverage by class for each package. For some classes there was no branch inside, so we did not take those classes into consideration for calculation of branch coverage. We could not find any class with no line coverage, so all classes were considered for statement coverage. For hypothesis 3 we calculated Statement and branch coverage for the whole project.

McCabe complexity: Jacoco also gives missed and covered complexity for each class. We added both missed and covered complexity to get the total complexity by class.

Mutation Score: Pitest tool is used to calculate the mutation score. Pitest tool change the original code to create mutants and then apply test suits. If test result for both mutant and original code is same the mutant is considered to be killed by the test case or if the result is different than original the mutant is considered to be survived. Pitest calculated mutation score by dividing killed mutants with total number of mutants and report it as a percentage. We had to add following code as a plugin in pom file under build to generate the mutation report.

```
<plugin>
      <groupId>org.pitest</groupId>
      <artifactId>pitest-maven</artifactId>
      <version>1.4.6</version>
      <configuration>
         <outputFormats>
            <param>HTML</param>
            <param>CSV</param>
         </outputFormats>
      </configuration>
</plugin>
```

We took the mutation score by classes for hypothesis 3. Some classes had no line coverage so we got 0% mutation score and we discarded those classes while calculating mutation score. Mostly Abstract classes had 0% or very low mutation score. Also, most mutants survived in boundary and negation condition of yellow line of jacoco.

Relative Code Churn: We used CLOC tool to find Relative code churn in our selected projects. CLOC can be used from command prompt to calculate number of source lines and newly added, modified and removed lines of code by comparing the current version with another version. CLOC can be installed using a number of package managers such as npm, yum, apt or choco. It is a platform independent tool and entirely written in perl. It can produce results in a variety of file formats such as plain text, csv, xml etc. It used several diff algorithms. These diff algorithms find the 'modification script' that will turn Text X to Text Y to find lines modified, inserted or deleted from code. We used the following commands to find the SLOC and Added, modified and removed number of lines by comparing two versions.
Cloc Project_name
Cloc –diff project_name_version 1 project_name_version2

Post Release Defect Density: We selected the projects which have available public issue-tracker(JIRA). These issue-trackers allow access to all the bugs recorded after release of a software. All the bugs have a version number and status associated with them. The issuetracker link for all the projects can be found on the project site. The number of bugs for each particular version were counted using Jira advanced search query. As we are considering post-release defect density, we considered all the closed or resolved bugs for the selected versions for all five projects. We used following filter to find all the resolved or closed bug in version 2.4 for Apache commons configuration. We collected number of bugs for the other projects in similar way.
project = CONFIGURATION and affectedVersion =2.4 AND issuetype = bug status in (Resolved, Closed)

VIII. DATA ANALYSIS

P Value: The p-value is the probability of obtaining a result equal to or extreme to what was expected when null hypothesis (H0) is true. The significance level refers to a pre-selected value of probability. If p-value is less than the significance level, then we can ignore the null hypothesis (H0). Then we consider the alternative hypothesis proposed in this paper. For this study, we selected value of the significance level as 0.05. If p value is less than 0.05, then we ignore the null hypothesis and we consider alternative hypothesis.

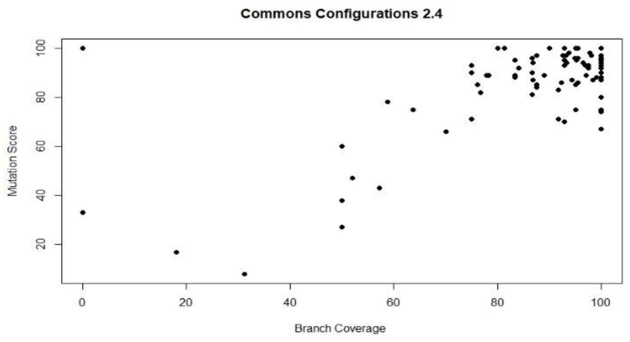
While analysing data we checked the p-value of each project, also for each version for hypotheses 3 and 4, if the p-value is greater than 0.05 we did not consider corresponding spearman coefficient of that project or version. [10]

a) Correlation between code Coverage and Mutation score:

In the following table we can observe that all the p-values for statement coverage and mutation score are less than 0.05 so we can ignore the null hypothesis and take alternative hypothesis in to consideration. Two of the projects have negative correlation, one of them has small but other one high negative correlation and three of the projects have high positive correlation. If do not take collections or codec in to consideration we can conclude that statement coverage has high positive correlation with mutation score

Project Name	Statement coverage & Mutation score	P value (Statement coverage & Mutation score)	Branch coverage & Mutation score	P value (Branch coverage & Mutation score)
Commons-Collections	-0.18	0.014	0.18	0.017
Commons-Codec	-0.61	7.149e-06	0.4	0.006
Commons-Configuration	0.64	<2.2e-16	0.620	6.919e-15
Commons-Math	0.57	<2.2e-16	0.44	<2.2e-16
Commons-Text	0.32	0.01	0.41	0.0007

We can also reject the null hypothesis for branch coverage and mutation score as all the p-values are less than .05 and we can conclude that branch coverage and mutation score have high positive correlation



b) Correlation between Coverage and McCabe Complexity: We assume that class that has methods with higher McCabe complexity will have lower test coverage in terms of Statement Coverage and Branch Coverage. We performed this analysis at class level for all projects. We calculated spearman coefficient (rho) using R. All the results were negative for both statement and branch coverage to McCabe complexity. We used csv report generated by Jacoco as our data source and to perform correlation analysis at class level. Jacoco by default does not provide statement coverage and McCabe complexity in its html report. We used below formula for calculating those:

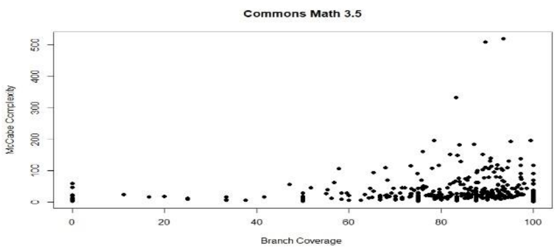
$$Staterment\ Coverage = \frac{Line\ Covered}{Line\ Covered + Line\ Missed} * 100$$

McCabe Complexity = Missed Complexity + Covered Complexity.
Jacoco report provide branch coverage in percentage, we have used that data by class.

TABLE III. CORRELATION BETWEEN COVERAGE AND MCCABE COMPLEXITY

Project Name	rho-Statement Coverage and McCabe complexity	P-value (Statement Coverage and McCabe complexity)	rho-Branch Coverage and McCabe complexity	P-value (Branch Coverage and McCabe complexity)
Commons Configurations	-0.3733	6.557e-08	-0.2711	0.0001
Commons Math	-0.1828	3.828e-06	-0.1283	0.001
Commons Codec	-0.3712	0.004	-0.2729	0.03
Commons Collections	-0.1971	0.0003	0.013	0.8
Commons Text	-0.08	0.4	-0.35	0.0006

As Observed in Table 3, for all projects, there is a small to moderate negative correlation between observed entities. Therefore, statement and branch coverage have small negative correlation with McCabe Complexity which confirms our hypothesis.

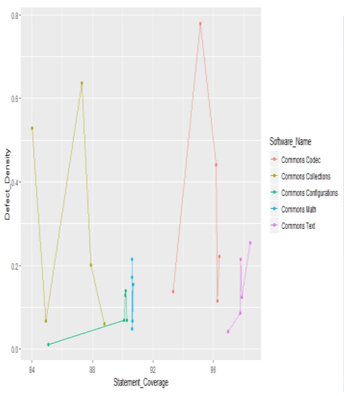


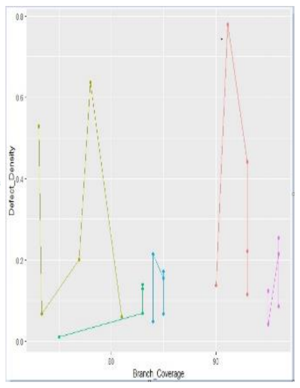
c)Correlation between code coverage and Defect Density

Project Name	Spearman Coefficient (Statement Coverage & Post-Release Defect Density)	P value (Statement Coverage & Post-Release Defect Density)	Spearman Coefficient (Branch Coverage & Post-Release defect Density)	P value (Branch Coverage & Post-Release defect Density)
Commons Configuration	0.7	0.1	0.7	0.18
Commons Math	-0.22	0.7	0	1
Commons Collections	-0.5	0.3	-0.3	0.6
Commons Codec	-0.2	0.7471	-0.11	0.8579
Commons Text	0.9	0.03	0.57	0.3

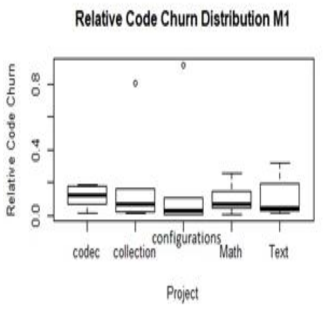
From the table, we can conclude that all the p values for selected projects are greater than 0.05 except one. Only one project named Apache Commons Text has p_value =0.03 which is less than the significance level. So, the null hypothesis cannot be ignored. The alternative hypothesis (Softwares with low test coverage contain more bugs) cannot be taken into consideration. For this it can be concluded that there is no relation between statement coverage/ branch coverage and post-release defect density.

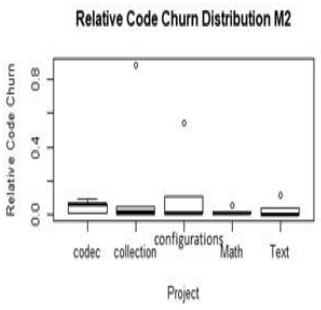
Statement coverage vs Defect Density and Branch coverage vs defect density graphs are shown below

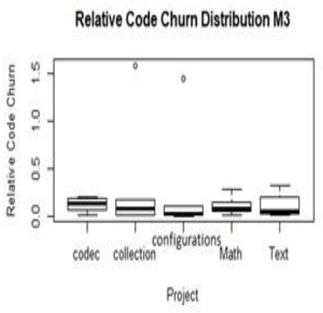




d)Correlation between relative code churn and Defect Density:







<u>Box-plots for metric Relative code churn:</u>
We have considered the following three separate measures to calculate code churn of a software.

$$\text{Measure 1 (M1)} = \frac{Modified\ SLOC+Deleted\ SLOC}{Size\ of\ the\ Software\ (SLOC)}$$

$$\text{Measure 2 (M2)} = \frac{Added\ SLOC+Modified\ SLOC}{Size\ of\ the\ Software\ (SLOC)} [4]$$

$$\text{Measure 3 (M3)} = \frac{Added\ SLOC+Modified\ SLOC+Deleted\ SLOC}{Size\ of\ the\ Software\ (SLOC)} [5]$$

we considered all languages while calculating for code churn as defect can originate from any part of the system and we do not have a way to take the defects originated only from java.
Data distribution for all three measures are shown in the boxplot.

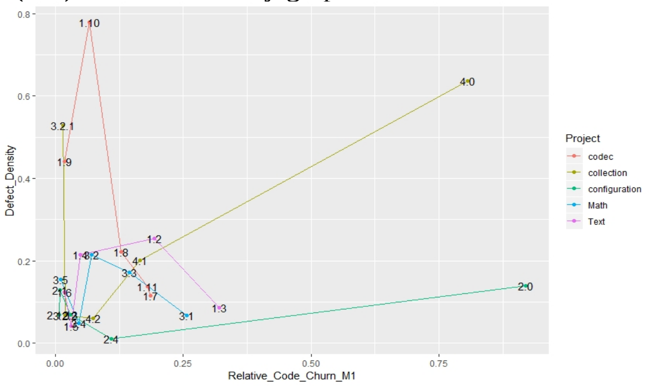
We can infer from the boxplot that as median for most of the projects are not in the middle data are not normally distributed. From the boxplot, It can be observed that while considering measure 1 for project Codec, lower median shows that 50% of data have higher relative code churn, but for measure 2, most of the data have less relative code churn than median also,as box size is comparatively smaller than the other two measures, it shows less variability in data for all the projects which makes it more predictable.
It can be noticed that two projects, Collections and Configurations, have outliers for all the measures M1, M2 and M3 but other two projects, Math and Text have outliers only in M2. In M2 four projects do not have maximum value. So, sample data is missing for these projects and length of data cannot be concluded for these projects.

On the other hand, for Apache commons math in M1 and M3 half of data has higher relative code churn than rest of data and but in M2, all the data are close to the median. There is less variability in data in M2 for project Math. But in both M1 and M3, 50% data have higher relative code churn than median value.

It can be concluded that M1 and M3 gives almost similar distribution only difference is in M3 data are more spread than M1 but distribution in M2 is different.

Project Name	rho-M1	P value M1	rho M2	P value M2	rho M3	P value M3
Configura tion	0.1	0.8	0.1	0.8	0.1	0.8
Commons Math	0.1	0.8	0.1	0.8	0.1	0.8
Commons Collection s	0.3	0.6	0.3	0.6	0.3	0.6
Commons Codec	-0.9	0.03	-0.7	0.18	-0.8	0.10
Commons Text	0.2	0.7	0.9	0.03	0.2	0.7

It can be observed from table that p value for all the projects is greater than 0.05 when using measure1(M1) to find relative code churn. Only one project, Commons codec has p-value less than 0.05. The same thing was observed when considering measure 2(M2), all projects have p-value higher than 0.05 except for commons Text. But while considering measure 3(M3) for Relative code churn, all the projects have p value higher than the significance level. So, it can be concluded that alternative hypothesis (Softwares with higher relative code churn will have high defect density) cannot be considered. It shows that there is no relation between the measures we use for relative code churn and post-release defect density. Relative code churn measure 1(M1) vs defect density graph is shown below



IX. Threats to Validity

We have considered all the open source projects, mostly written in Java. The environment in which an open source software is being developed is very different than the environment of a closed sourced software. The results of this study may not be applicable to closed source softwares.

we have used consecutive versions in CLOC to find code churn. There is a possibility that a bug was injected in the older version and found in next to next version. Those bugs we considered for the version in middle. That can affect our results for hypothesis 4.

In Jira issuetracker, there were some bugs without any specified version. We looked into comments to find versions associated with those bugs. But in some cases, we could not find version number some bugs. We did not include those bugs which were without any version associated with them. This can also affect our results for hypothesis 3 & 4.

For hypothesis 3, we have used code coverage for whole project, not by class as we can not trace defects to their originated classes.

X. Results

From the data analysis we can infer that statement and branch coverage has high positive correlation with mutation score. Thus hypothesis 1 is true. Both statement and branch coverage have small negative correlation with McCabe complexity. Thus hypothesis 2 is also true to some extent. As p-value was not significant enough to consider the alternative hypothesis we can conclude there is no correlation between code coverage and defect density, same is true for relative code churn and defect density, so we can reject hypotheses 3 and 4.

XI. References

[1] Mei-Huei Tang, Ming-Hung Kao and Mei-Hwa Chen," An empirical study on object-oriented metrics," Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403), Boca Raton, FL, USA,1999, pp. 242-249

[2] Khashayar Etemadi Someoliayi & Sajad Jalali, Mostafa Mahdieh & Seyed-Hassan Mirian-Hosseinabadi (Feb 2019) . Program State Coverage: A Test Coverage Metric Based on Executed Program States.In 18 march 2019 *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)[SSF1] China,* IEEE.

[3]"An In-Depth Explanation of Code Complexity," *Codacy Blog*, 13-Jul-2018. [Online]. Available: https://www.codacy.com/blog/an-in-depth-explanation-of-code-complexity/. [Accessed: 26-Jun-2019].

[4] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," Pro-ceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.

[5] KathrynEE, "Analyze and report on code churn and code coverage - TFS," Analyze and report on code churn and code coverage - TFS — Microsoft Docs. [Online]. Available: https://docs.microsoft.com/en- us/azure/devops/report/sql-reports/perspective-code-analyze-report-code-churn-coverage?view=azure-devops- 2019.

[6] L. Westfall, \Defect Density," Available: http://www.westfallteam.com/Papers/defect density.pdf.

[7] C. Rahmani and D. Khazanchi, "A Study on Defect Density of Open Source Software," 2010 IEEE/ACIS 9th International Conference on Computer and Information Science, Yamagata, 2010, pp. 679-683

[8] M. Daskalantonakis, "Correction to A practical view of software measurement and implementation ex-periences within Motorola," IEEE Transactions on Software Engineering, vol. 19, no. 2, pp. 199–200,1993

[9] C. Rahmani and D. Khazanchi, "A Study on Defect Density of Open Source Software," 2010 IEEE/ACIS 9th International Conference on Computer and Information Science, Yamagata, 2010, pp. 679-683

[10] Kajuri, S. (2018, December 14). Application of Hypothesis Testing and Spearman's rank correlation coefficient to demystify... Retrieved from https://towardsdatascience.com/application-of-hypothesis-testing-and-spearmans-rank-correlation-coefficient-to-demystify-b3a554730c91

[11] "What you need to know: Code Coverage with JaCoCo," *iCiDIGITAL*, 13-Jun-2017. [Online]. Available: https://www.icidigital.com/blog/web-development/code-coverage-with-jacoco. [Accessed: 26-Jun-2019].
IEEE conference template