ISSN XXXX XXXX © 2017 IJESC

Research Article Volume 7 Issue No.6

Fault Prediction using Metric Threshold Value of Object Oriented Systems

Shruti Gupta[1], D.L Gupta[2]
M.Tech Scholar[1], Associate Professor[2]
Department of Computer Science and Engineering
Kamla Nehru Institute of Technology, Sultanpur, Uttar Pradesh, India

Abstract:
Software metrics helps in analyzing many factors of software quality such as fault proneness, reusability, and maintenance effort. Software metrics are values collected from software source code to analyze and evaluate where problems are more probable to occur. These values are used to flag warnings of the problem causing parts of software code using threshold values. However, the proposed techniques did not consider the data distribution and skewness in data. A methodology based on removing redundancy and then log transformation to improve the metrics quality has been implemented. To explore the effect of removing redundancy and performing log transformation on data analysis, we conduct analysis of using software metrics after transformation in identifying fault-prone areas on multiple releases of 8 products (30 releases). This methodology results show that the removing redundancy and then doing log transformation can be used to derive threshold values for all metrics under investigation and then performing the fault classification for each metric. In this research, we aim to propose a methodology to remove redundancy in datasets record values of 8 products (30 releases) after which again log transformation technique is used to calculate threshold value. The results of the transformation after removing redundancy are then used to conduct fault-proneness classification based on threshold values and compared against the results of fault classification based on threshold value after log transformation without removing redundancy in dataset record values. The fault classification after removing redundancy in dataset record values is better than fault classification without removing redundancy in dataset record values.

Keywords: Software metrics; CK metrics; Software metric thresholds; Software quality; Fault Prediction; Redundancy removal

I. INTRODUCTION

One of the objectives in software engineering is controlling of software quality. Quality of a software design can be measured by quantitative means provided by software design metrics and these software design metrics are may be the basis of techniques that can help in controlling software quality. There have been abundant amount of studies on software metrics in both the procedural and object-oriented (OO) paradigms. We often use software metrics to predict the behaviour of a system or the components of a system. In the OO paradigm, empirical studies have shown that metrics could predict quality factors such as class error proneness, maintenance effort, maintenance performance, design, and project progress. However, the data used in these studies were collected during the development phases of the systems. As systems continue to evolve after their release, significant amounts of resources must be dedicated to maintain the quality of the systems as they evolve [4]. Although the post-release systems tend to have fewer errors than the systems that are under development, they are still not free of errors. Software maintenance effort, which is the effort required to keep the post-release systems functioning properly, has been reported to be the largest share of the entire software cost; one estimate puts the maintenance cost at 60% of all effort expended by a development organization and the figure continues to rise. The challenge of locating and fixing errors still exists in the post-release systems. Fault proneness is measured indirectly using a set of software metrics that measures the internal attributes of software systems. These metrics should have been validated empirically in previous research and can be considered consolidated set of software metrics, although there are some variations in how these metrics are defined and collected [5]. These metrics also have been used in many quality tools, either commercial or open source, to track and audit software quality (e.g., Understand for Java and ckjm). The software developers and testers can use metric tools to audit and track the complex parts of the system that need more attention and maintenance. Many metric tools can be integrated in development environments (e.g., Atollic True STUDIO, Understand for Java, and resource standard metrics (RSM)). For example, for the purpose of maintaining the internal quality of systems, tools can identify code smells such as God Classes in a system. God Classes are such classes that are very big and have very large responsibilities, and usually code analysts consider it as a source of faults in the system. Software metrics in such cases can lead the developers to maintain the internal structure of the system throughout applying many refactoring operations such as Extract class, Extract Superclass, or Extract Interface. Therefore, software metrics are becoming widely in tools to assess software quality. The metrics have shown skewness to the right in many previous researches. There is a belief that software metrics follow other distributions rather than the normal distribution [6,7]. There are many research papers that discussed the log-normal distribution as a possible fit for software metrics [8-10]. In addition, although many research papers have derived threshold values of metrics, these derivations were not successful for all metrics [11-19]. There are two primary uses of metrics: the factor-prediction models and the threshold values. Most previous studies on OO metrics offered empirical models to predict factors such as error proneness and maintenance activities. However, these models did not offer threshold values for the metrics and they are not applicable in early phases of the software design. Metric threshold values can help us identify the problematic classes as we design them. With the help of threshold values, a design engineer can

monitor the design modules (with the help of tools) during the daily engineering work and make quick re-design decisions if the metrics of the modules exceed the threshold values. In this work, we propose to use data transformation after removing redundancy in data to improve the data quality and to reduce data skewness. We suggest to validate the effect of redundancy removal and transformation on two important applications of software metrics in assessing and evaluating software quality: threshold derivation and fault classification. Firstly, we remove redundancy in order to avoid erroroneous observations leading to incorrect statisatics results. And then we derive thresholds for a well-known suite of metrics, the Chidamber and Kemerer (CK) metrics [20]. CK metrics can be used to analyze software quality either by using graphical diagrams or by setting threshold values ([19]; Rosenberg et al. [42]). These threshold values are usually set based on developers experience or derived from research. This research aims to find software metrics threshold values from the transformations of metrics using the log transformation after removing redundancy in the dataset record values and on the basis of these derived thresholds fault prone classification of software is conducted. In this research, we removed redundancy from the dataset record values, then we use the natural log function to transform all metrics. The threshold values are then derived for all log metrics. We validated our findings by using derived thresholds in predicting faulty classes in two scenarios: without removing redundancy and after removing redundancies. The results show better performance in identifying faulty classes using the derived thresholds from the log transformation after removing redundancy in the dataset record values than using the derived thresholds from the log transformation without removing redundancy from the dataset record values. The remainder of the paper is organized as follows. In section 2 we describe the research, software metrics in general, dataset collection, model used, OO metrics threshold values, and the performance evaluation. In Section 3 we present the detailed results analysis and observation in our research. Finally in Section 4 we conclude our research work and present future enhancements.

II. RESEARCH METHODOLOGY

In this section, we discuss the CK metrics that we have considered in our research work. And then we discuss we discuss the model shown in Figure I that we use for removing redundancy in dataset record observations, deriving threshold values using log transformation and then finally using these derived thresholds to predict fault and thus evaluating performance.

A. The Chidamber and Kermerer (CK) Metrics

Chidamber and Kemerer have proposed and validated a suite of 21 metrics. But in our work we have considered 6 metrics from the CK metric suite that covers six different concepts of the internal software quality. These metrics encompasses concepts of cohesion, coupling, complexity, inheritance depth, number of children, and class responsibility. These metrics are defined as follows:

- Weighted Methods Complexity (WMC): The WMC metric is used to count the number of methods in a class. This metric is used to cover the concept of complexity. Values of WMC are directly proportional to the complexity that is if values of WMC are larger than complexity also increases and vice-versa.

- Depth of Inheritance Hierarchy (DIT): The DIT metric covers the concept of the inheritance depth. This metric is used to measure the length of inheritance depth tree from root class to the considered class. This metric helps the developers to understand all the characteristics that are inherited in the considered class like specialization concept in database.

- Number of Child Classes (NOC): This NOC metric encompasses the concept of number of children of a considered class. The NOC metric counts the number of descendant of the considered class. This metric is used to developers the number of times the considered class has been inherited by another class to increase their specialization and uses.

- Coupling Between Objects (CBO): The CBO metric covers the concept of coupling which shows the strength of interconnection among classes in a software system. The metric is measured by counting the number of coupling to other classes. Couplings are counted for method calls, field accesses, inheritance, method arguments, return types, and exceptions. It is told in previous papers that values of CBO are directly proportional to the faults in a software system that is if values of CBO are larger then greater are the chances of the fault occurences and vice-versa.

- Response For Class (RFC): The RFC metric covers the concept of class responsibility. This RFC metric is used for counting the number of responses in the response set for the considered class which includes the number of local methods and the number of remote methods invoked by other local methods. The response set includes classes whose methods in class inheritance and methods that can be invoked in other objects. Larger the values of RFC, it shows that class has many interaction with other classes.

- Lack Of Cohesion Of Methods (LCOM): The LCOM metric is the difference number of the pair of methods in a class that have similar attributes(A) and number of the pair of methods in a class that have no similar attributes(B). If the difference is negative then value of LCOM becomes 0. This metric covers the concept of the cohesion which means that it describes the inter-relatedness of objects among the class. It is stated that low cohesion leads to increase in complexity resulting in more faults in a class.

Relationship between fault-proneness in a class and CK metric has been studied in many previous research papers. In a survey on the effect of metrics on fault proneness, authors found out that object-oriented metrics are more successful in finding faults than the procedural metrics [22]. The study also found out that CK metrics were the most used set of metrics suite to predict fault proneness of classes, that is, NOC (53 papers), DIT (52 papers), RFC (51 papers), LCOM (50 papers), CBO (48 papers), and WMC (44 papers). Three metrics, WMC, CBO, and RFC, were always effective in predicting fault proneness, while LCOM was not effective and DIT and NOC were not very useful. To provide more detailed indicators of the effect of the CK metrics on fault proneness, we summarized the effect of CK metrics on fault proneness as shown in [15]. We summarize these results in Table I. Large values of WMC, RFC, and CBO metrics increase the fault proneness of classes. Large values of DIT and LCOM metrics in most studies also increase the fault proneness of classes. We can notice that NOC is less studied and large values of NOC metric have a negative impact on fault proneness in three studies and a positive impact in only two studies. Although,

these results and the recent survey do not suggest studying all the CK metrics, we intend to include all CK metrics in this study for the comparison with previous findings.

Table.I. Summary of the impact of Chidamber and Kemerer metrics on fault proneness as reported in ([15]: Table I).

Metric	Positively significant	Negatively significant	No significant relationship	Not included
WMC	12	0	0	0
DIT	4	2	6	0
NOC	2	3	4	3
CBO	11	0	1	0
RFC	11	0	0	1
LCOM	6	0	1	5

WMC, weighted methods complexity; DIT, depth of inheritance hierarchy; NOC, number of child classes; CBO, coupling between objects; RFC, response for class; LCOM, lack of cohesion of methods.

B. Dataset Collection

We have conducted our research work in 8 open source projects over their multiple releases. The detailed description of each project having multiple releases in stated in Table II. These open source project have been developed in JAVA and their source code is easily available online. We have collected the metrics for 30 releases and in Table II last two columns describe the number of releases for the project and the number of classes in last release of project. These details have collected using open source tool CKJM. The metric dataset for the 8 projects is publicly reported by PROMISE data repository [23,24].

C. Redundancy Removal

In this research, we propose to remove redundancy from dataset record values after the dataset is extracted from the publicly available PROMISE data repository. The dataset obtained from the PROMISE data repository contains 21 CK metrics for consideration but we consider only 6 CK relevant metrics namely WMC, DIT, NOC, CBO, RFC, LCOM. From these 6 considered metrics we remove duplicate set of record values from them and only unique set of record values remain in the dataset. Redundancy removal avoids the problem of over-fitting and scaling problem [2,3].

Table.2. Systems Under Investigation

System	Brief description	Number of releases	Number of classes in the last release
Ant	A Java building tool http://ant.apache.org	5	745
Camel	A versatile open-source integration framework http://camel.apache.org	4	965
Ivy	A dependence manager focusing on flexibility and simplicity http://ant.apache.org/ivy/	3	352
Jedit	A Java IDE and editor http://jedit.org/	5	492
Log4J	A logging services http://logging.apache.org/log4j	3	205
Lucene	Develops open-source search software http://lucene.apache.org/	3	340
Synapse	Enterprise Service Bus http://synapse.apache.org/	3	256
Xerces	A Java-based XML parser http://xerces.apache.org/xerces-j/	4	588

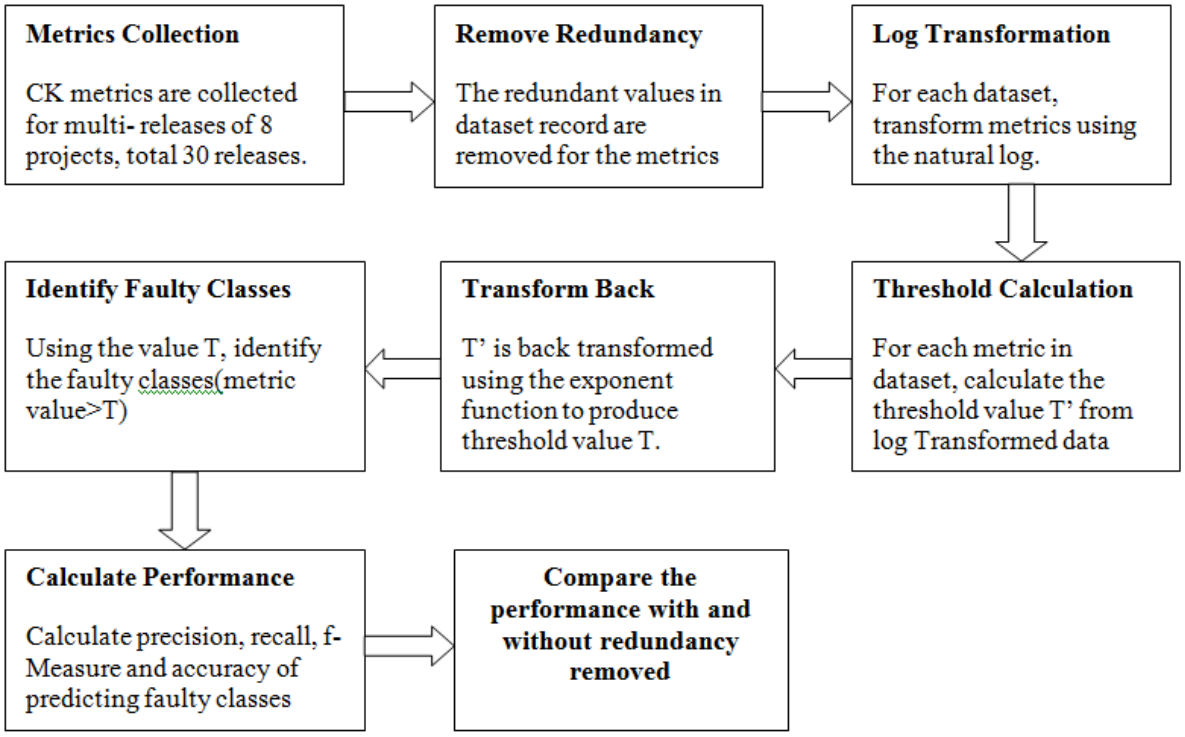

Figure.1. Model used to remove redundancy, threshold derivation and fault classification and performance evaluation

D. Threshold Derivation

In this research, we propose to use log transformation to find threshold values after removing redundancy from the dataset using the mean and standard deviation (two-thirds of data are within one standard distribution). We follow the process as depicted in Figure 1 to find threshold values for a particular

system. In this work, the process is conducted for multi-releases of 8 projects in Java from various application domains and different sizes. The study lays emphasis on the CK-metric suite as it was widely reported on many research papers in both empirical and theoretical validations as quality predictors. CK metrics were used repeatedly to predict software quality factors including fault proneness, software reusability, and maintenance effort. The process starts by collecting the data for the CK metrics for all releases.

The distribution of software metrics data has been analyzed in previous researches such as Basili et al. [43]; [25, 26]. All studied metrics are found positively skewed to the right (Figure 2 shows the histograms of all metrics in Camel version 1.0). All other project releases also have skewed distribution in our data sets. Hence, metrics are not always well characterized by their descriptive statistics. Data skewness affects the interpretation and usage of software metrics in evaluating software systems. Distributions skewed to the right do not necessarily follow the normal distribution.

Therefore, a transformation is needed to produce data that are less skewed and more close to fit a normal distribution. There is a variety of transformations that is usually used to reduce skewness in data, but the most used ones are logarithmic transformation, the square root transformation, and the inverse transformation [27]. The log transformation cannot produce transformation for zero values; therefore, a constant should be added (e.g., 1 is added in this work). The square root does not work for negative values, and if values are continuous between 0 and 1 as well as above one, then the square root is not desirable. The inverse transformation reverses the order of software modules, which is not desirable in this work.

In this research, we propose to use the logarithmic transformation because it reduces the relative distances between data points, which are how this technique reduces skewness in metrics data [27]. As an example, we present the effect of data transformation in Camel 1.0. Table III shows the skewness index before and after reducing redundancy and the log transformation for Camel 1.0. We use the skewness index to show the differences between before and after transformation. Skewness index is a measure of asymmetry in data distribution [27].

The normal distribution has a skewness of zero, and any symmetric data should have a skewness near zero. The skewness parameter after the transformation is close to zero and therefore closer to a normal distribution for all metrics. The NOC metric is the most asymmetric because most of its values are zeroes. All metrics data are transformed into a natural log, and then the parameters, the mean, and the standard deviation are calculated.

Table.3. Skewness statistics before and after redundancy removal in camel 1.0

METRIC	Skewness before removing redundancy and without log transformation	Skewness after removing redundancy and without log transformation	Skewness before removing redundancy and after log transformation	Skewness after removing redundancy and after log transformation
WMC	3.442751	3.416036	0.222655	0.220327
DIT	0.990612	0.968959	0.199898	0.13434
NOC	6.284786	6.130979	3.627331	3.566791
CBO	6.959064	6.857178	0.25098	0.257802
RFC	2.300273	2.285333	0.42717	0.44778
LCOM	9.493816	9.264176	0.671351	0.606597

To find a threshold value for a metric using the distribution parameters, we use the following calculations:

$T' = \mu + \Omega$, $T'' = \mu - \Omega$;

Where, μ is the mean and Ω is the standard deviation.
These are then considered threshold values that can be used to detect where more faults could be introduced. However, the metrics under investigation are lower bounded, and we propose to derive only one threshold value using $\mu + \Omega$ that identifies most complex parts (>threshold value) of the systems under investigation. We use the mean and the standard deviation together to find the one- third of the data, but because we need the data on the right then the identified cases are within the upper one-sixth of the data. The results of the T′ are representative of the transformed data, and we need to reverse the transformation back to produce thresholds on the original data. The values of T′ are reversed back using the exponential function and denoted T in the following calculation:

$T = Exp(T')$

The values of T are produced for all systems (30 releases) under investigation for all six metrics. The derived metrics are then evaluated to identify the faulty classes, and the results are compared with the thresholds that are derived using the parameters after removing redundancy with transformation.

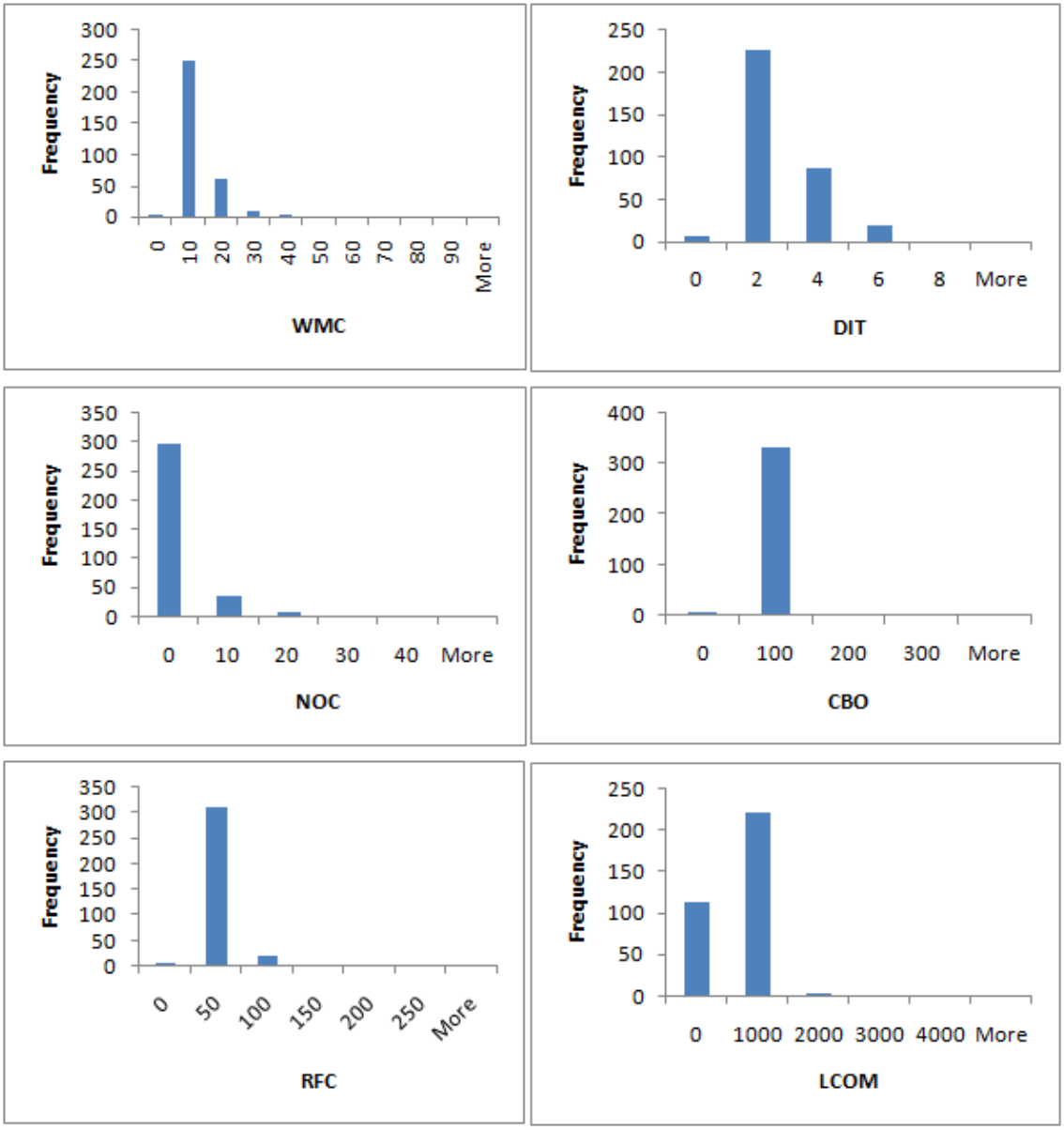

Figure.2. Histograms for all metrics in Camel 1.0 before transformation.

E. Threshold Performance Evaluation

Threshold values are vital to help both developers and testers in locating which classes need more attention. Derived thresholds should be evaluated against one of the software quality factors. In this research, we validate the derived thresholds to classify whether classes are faulty or not. The fault data for the systems under investigation were collected from repositories of the projects and summarized by the Promise Data Repository [23,24]. The authors have used BugInfo to collect the fault data. BugInfo analyzes the history of the classes by studying the code repositories (Subversion or Concurrent Versions System). If a log contains a fault fix description, then the affected classes are marked as faulty. BugInfo uses regular expressions to extract fault information. When a log description fits to a regular expression, then faults count is incremented. Faulty classes are the classes that have more than one fault reported in the repository; otherwise, classes are marked not faulty. These cases are considered actual in the confusion matrices as shown in Table IV. A threshold value is used to classify the classes into two groups: faulty classes, if a metric value>= T and not faulty classes, if a metric value<T. Classes in the first group are considered more fault prone, while classes in the second group are otherwise.

From this classification, we can create a confusion matrix as shown in Table IV. The confusion matrix is used to measure the performance of using thresholds model in identifying actual fault classes using three measures, Recall, Precision, F-measure and accuracy. These measures are calculated as follows:

$$\text{Recall} = \frac{TP}{TP+FN}, \text{Precision} = \frac{TP}{TP+FP}$$

$$\text{F} - \text{measure} = \frac{(\beta^2 + 1) * \text{precision} * \text{recall}}{\beta^2 * \text{precision} + \text{recall}}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

- The term β is used to assign a weight to the Recall. In our work, β is equal to 1, and Recall and Precision are equally weighed.
- True positives (TP): faulty classes that are correctly classified as such, that is, there are faults fixed in the class (faulty) and the metric value exceeds the threshold (faulty).
- False negatives (FN): faulty classes that are misclassified as not faulty, that is, there are faults fixed in the class (faulty), but the metric value is less than threshold (not faulty).

- True negatives (TN): nonfaulty classes that are correctly classified as such, that is, there are no faults fixed in the class (not faulty) and the metric value is less than threshold (not faulty).
- False positives (FP): nonfaulty classes that are misclassified faulty, that is, there are no faults fixed in the class (not faulty), but the metric value exceeds the threshold (faulty).

Table.4. The confusion matrix based on a threshold value

	Actual	
Predicted	Faulty	Not faulty
Metric > = T	True positive	False positive
Metric < T	False negative	True negative
Totals	P	N

The values of both the Recall and Precision are between 0 and 1. Values that are close to 1 mean better results. If the value is 1, then the classifier is ideal and without FN or FP. However, the high values of Recall and Precision do not coincide. In practice, it is hard to achieve high Recall and Precision, that is, high Recall occurs often with low Precision. We use F-measure, which can be used to assess the overall classification performance and combine both Recall and Precision. In addition, F-measure is not sensitive to imbalance in data, which is the case in studying fault proneness of software, that is, few classes are marked fault and the majority is marked not faulty. To find the effect of data transformation, we derived thresholds from two data sets: before transformation and after transformation. Both sets of thresholds are then used to classify classes into faulty or not. We then compare the obtained classification from transformed data against the results of fault classification obtained from original data. The performance of fault classification is compared using F-measure. Large values of F-measure are desired and should be close to 1. To find the significance of the differences in performance (F-measure) between the two techniques (before versus after transformation), we conducted a pairwise Wilcoxon signed-rank test at the 95% confidence level. This test is nonparametric and does not have assumptions about the underlying distribution of the data. The differences are significant between the two groups if the statistical test, p-value, is less than 0.05. There are significant differences among the two thresholds except for DIT. The identified thresholds in both techniques for DIT metric are very close to three; therefore, no differences are expected in threshold applications as well.

III. RESEARCH RESULTS

We conducted the process in Figure 1 to derive thresholds and the performance for predicting faults for all data sets for the six metrics.

A. Threshold Calculation Result

We do not provide detailed results for all releases and only the mean values of all releases in the 8 systems under investigation. We provide details on the releases of only one system, Camel, as shown in Table V. In the following, we discuss the results for each metric separately.

Table.5. Derived Thresholds for all metrics in Camel

	WMC	DIT	NOC	CBO	RFC	LCOM
Camel 1.0	13.86996	3.066556	1.039278	16.49223	36.2753	51.40451
camel 1.2	15.13335	3.089331	1.069965	17.29478	39.64393	61.91362
camel 1.4	15.57284	3.221924	1.049051	18.62215	41.7294	67.78625
camel 1.6	15.55869	3.240955	1.04794	18.93689	41.86846	69.24618

We have calculated threshold values of 6 CK metrics after removing redundancy from datasets for 8 projects. The calculated values are shown as follows in Table VI.

- ***WMC Threshold*:** WMC indicates the complexity of class or interface. It is basically the sum of all complexities of the methods in a class. WMC metrics large values are thought to be problematic and thus those classes are marked as so. The team of developers and testers are then required to go through the marked classes to find possible problems in their coding and designing. In previous studies various threshold values of WMC values have been given but there is still no widespread agreement on a particular value. Table VI shows the calculated thresholds for the 8 systems under investigations. These are the average values for multiple releases of each project. We could not notice a common trend in the evolution of the 8 projects for consecutive releases but there is no large difference among the consecutive releases values. The mean value is approximately 17.

- ***DIT thresholds*:** DIT indicates number of ancestors of a class, that is, DIT metric indicates depth of inheritance in a class. DIT tells the developer the number of times the considered class is inherited by other classes. Larger values of DIT values are considered to have more complex classes, which leads to difficulty in understanding, maintaining, and reusing of the classes, thus these classes are marked to be investigated during testing and maintenance phases of software development lifecycle. There have been many studies regarding the impact of DIT but there is no given widespread acceptable value of DIT. Table VI shows the calculated threshold values for DIT for the 8 projects. We can notice that DIT does not vary like WMC during evolution of consecutive releases of 8 projects, that is, all values are close to three. The mean is equal to three.

- ***NOC thresholds*:** NOC indicates the descendants of a class, that is, NOC tells the number of classes that is inherited by considered class during its execution. NOC indicates both inheritance and abstraction in classes. Larger values of NOC are considered problematic and can be marked as so during maintenance and testing phases. Various previous studies have tried to reach common agreeable NOC threshold values but there is no widespread acceptance for particular value. Table VI shows the calculated threshold values for the 8 projects. We notice that the all the values of threshold are approximately equal to 1.

- ***CBO thresholds*:** CBO metric counts the number of other classes to which a class is coupled. CBO is used for

measuring coupling among classes. High coupling is considered to be problematic leading to more complex classes making it difficult to understand, increasing testing and maintenance efforts and thus these classes are marked in both the designing and coding phase. Coupling among classes can be reduced by code cleaning and refactoring continuously. Various previous studies reported many threshold values for CBO. Table VI shows the threshold values. We notice that CBO values does not vary much among the 8 projects, and we could notice that values have increasing trend for later releases. Mean value is approximately 17.

- ***RFC threshold*:** RFC is an indicator of the amount of responsibility performed by class. The RFC metric counts the number of methods in the response set for a class, which includes the number of methods in the class and the number of remote methods invoked by the methods in the class. Classes having large response set are prone to more faults and thus needing more maintenance. Such classes require more rigorous testing to be sure that all requirements are met by classes assigned to them. RFC values show great variation among the threshold values and thus there is no common threshold value for all systems. Table VI shows the threshold values for the RFC metric. Mean value is approximately 51.

- ***LCOM thresholds*:** LCOM metric is the number of pairs of methods in the class using no attribute in common, minus the number of pairs of methods that do. The LCOM is set to zero if this difference is negative. Lack of cohesion in class has a harmful effect on the quality of design. Low-cohesive classes show ill-structuring and difficult to maintain and test of the classes. Low-cohesive classes affect both encapsulation and abstraction levels in class design. To increase cohesion in class we can code cleaning and refactoring. However, to complete their tasks correctly, the developers need to know which level of cohesion is required. The LCOM metric was recognized as the most ill-defined metric among other CK metrics. Table VI shows the threshold values the LCOM metric. We notice that LCOM values show great variations among the 8 projects for different releases. The mean value is approximately equal to 66.

Table.6. Threshold values for 6 CK Metrics after removing redundancy for 8 projects

	wmc threshold (mean)	dit threshold (mean)	noc threshold (mean)	cbo threshold (mean)	rfc threshold (mean)	lcom threshold (mean)
ant	18.12077	3.685256	1.052722	17.18905	61.83121	70.37726
camel	15.03371	3.154691	1.051558	17.83651	39.87927	62.58764
ivy	17.85609	2.773193	0.783952	18.697	57.41576	87.72123
jedit	19.30622	4.343282	0.836688	21.08781	69.70872	80.05485
log4j	12.53094	2.436433	0.661855	11.94717	42.07368	34.988
lucene	15.76911	2.555124	1.297417	16.77015	40.68402	28.64256
synapse	12.38651	2.233747	0.776952	20.94617	54.74754	34.56419
xerces	22.47552	3.030955	0.993855	12.89372	45.6356	125.7596

An important concept of object oriented systems is inheritance. This concept of inheritance helps developers of software to reuse existing system components. It is suggested to favour composition as a means of reusing existing system components in the object-oriented designing of system more than the concept of inheritance [28,29]. If software developers have no intention to change the design of inherited classes, then the exposure of design of super-classes takes place in the subclasses, which may result in making subclasses more fault-prone. The inheritance of a program is measured using two CK metrics namely, DIT and NOC. In this research, we have identified threshold values for these metrics, DIT ≡ 3 and NOC ≡ 1. However, the data values in dataset considered in this research show that nearly all the classes in dataset have value of zero for the NOC metric, therefore mean value of NOC metric is very small in our observation result. The selected threshold for NOC may not be meaningful, because most classes have NOC= 0. Therefore NOC>=1 separates between parent and leaf classes. On the other hand, the selected threshold for DIT is also small. However, there are many research papers that published similar thresholds for DIT (Daly et al.[30-32]), and our results support those previous findings. The low variability in inheritance metrics (DIT and NOC) has been observed in many previous research papers. A large number of classes have NOC= 0. Table I also shows that NOC and DIT have no significant relationship with fault proneness in many research papers. However, the CK metrics are used as a suite to measure different aspects of software design and code, and inheritance metrics are still needed to fully measure a software system. The work of McCabe and Rosenberg has not suggested any thresholds for the NOC (Rosenberg et al. 1999)[21]. In addition, Shatnawi et al. [14]could not report a threshold for NOC using Receiver Operating Characteristic (ROC) analysis[14].

B. Performance Calculation Result

We do not provide detailed results for all releases and only the mean values of all releases in the 8 systems under investigation. We provide details on the releases of only one system, Camel. In the following, we discuss the results for each metric separately.

- **Precision Calculation:**

Table.7.Values of precision before and after removing redundancy and after log transformation for camel for WMC

	WMC before removing redundancy	WMC after removing redundancy
camel 1.0	0.101695	0.105263
camel 1.2	0.455357	0.5
camel 1.4	0.335404	0.368852
camel 1.6	0.329545	0.351563

Table.8.Values of precision before and after removing redundancy and after log transformation for camel for DIT

	DIT before removing redundancy	DIT after removing redundancy
camel 1.0	0.029412	0.029412
camel 1.2	0.25	0.25
camel 1.4	0.222222	0.222222
camel 1.6	0.173554	0.173554

Table.9. Values of precision before and after removing redundancy and after log transformation for camel for NOC

	NOC before removing redundancy	NOC after removing redundancy
camel 1.0	0.097561	0.071429
camel 1.2	0.452055	0.45283
camel 1.4	0.25	0.25
camel 1.6	0.333333	0.352113

Table.10.Values of precision before and after removing redundancy and after log transformation for camel for CBO

	CBO before removing redundancy	CBO after removing redundancy
camel 1.0	0.184211	0.176471
camel 1.2	0.423077	0.442857
camel 1.4	0.261905	0.267241
camel 1.6	0.330827	0.364407

Table.11.Values of precision before and after removing redundancy and after log transformation for camel for RFC

	RFC before removing redundancy	RFC after removing redundancy
camel 1.0	0.085106	0.073171
camel 1.2	0.443299	0.460674
camel 1.4	0.321918	0.326241
camel 1.6	0.318182	0.323741

Table.12. Values of precision before and after removing redundancy and after log transformation for camel for LCOM

	LCOM before removing redundancy	LCOM after removing redundancy
camel 1.0	0.125	0.125
camel 1.2	0.472222	0.5
camel 1.4	0.343373	0.340278
camel 1.6	0.320225	0.335484

Table.13. Values of precision(mean) before and after removing redundancy and after log transformation for 8 projects

	WMC before	WMC after	DIT before	DIT after	NOC before	NOC after	CBO before	CBO after	RFC before	RFC after	LCOM before	LCOM after
Ant	0.427704	0.435685	0.214033	0.214033	0.253254	0.253254	0.376957	0.362688	0.524069	0.532912	0.422449	0.433296
Camel	0.3055	0.33142	0.168797	0.168797	0.283237	0.281593	0.300005	0.312744	0.292126	0.295957	0.315205	0.32519
Ivy	0.43555	0.43555	0.259921	0.259921	0.246528	0.246528	0.437831	0.437831	0.445617	0.447034	0.446066	0.460185
Jedit	0.480016	0.496654	0.320279	0.320279	0.184056	0.184056	0.409996	0.41564	0.508439	0.516021	0.4717	0.485535
Log4j	0.895753	0.895753	0.359274	0.359274	0.674074	0.674074	0.822129	0.822129	0.866667	0.916667	0.829086	0.857998
Lucene	0.907317	0.907317	0.531403	0.531403	0.584066	0.584066	0.727626	0.727626	0.864621	0.864621	0.798669	0.798669
Synapse	0.432957	0.445238	0.144048	0.144048	0.148448	0.148448	0.487447	0.487447	0.549489	0.555045	0.454666	0.465895
xerces	0.496407	0.541534	0.370226	0.398448	0.565642	0.565642	0.561833	0.642511	0.558267	0.634475	0.456715	0.482474

- **Recall Calculation:**

Table.14. Values of Recall before and after removing redundancy and after log transformation for camel for WMC

	WMC before removing redundancy	WMC after removing redundancy
camel 1.0	0.461538	0.307692
camel 1.2	0.236111	0.189815
camel 1.4	0.372414	0.310345
camel 1.6	0.308511	0.239362

Table.15. Values of recall before and after removing redundancy and after log transformation for camel for DIT

	DIT before removing redundancy	DIT after removing redundancy
camel 1.0	0.076923	0.076923
camel 1.2	0.074074	0.074074
camel 1.4	0.165517	0.165517
camel 1.6	0.111702	0.111702

Table.16. Values of recall before and after removing redundancy and after log transformation for camel for NOC

	NOC before removing redundancy	NOC after removing redundancy
camel 1.0	0.307692	0.153846
camel 1.2	0.152778	0.111111
camel 1.4	0.186207	0.110345
camel 1.6	0.207447	0.132979

Table.17. Values of recall before and after removing redundancy and after log transformation for camel for CBO

	CBO before removing redundancy	CBO after removing redundancy
camel 1.0	0.538462	0.461538
camel 1.2	0.152778	0.143519
camel 1.4	0.227586	0.213793
camel 1.6	0.234043	0.228723

Table.18. Values of recall before and after removing redundancy and after log transformation for camel for RFC

	RFC before removing redundancy	RFC after removing redundancy
camel 1.0	0.307692	0.230769
camel 1.2	0.199074	0.189815
camel 1.4	0.324138	0.317241
camel 1.6	0.260638	0.239362

Table.19. Values of recall before and after removing redundancy and after log transformation for camel for LCOM

	LCOM before removing redundancy	LCOM after removing redundancy
camel 1.0	0.615385	0.538462
camel 1.2	0.236111	0.208333
camel 1.4	0.393103	0.337931
camel 1.6	0.303191	0.276596

Table .20. Values of recall(mean) before and after removing redundancy and after log transformation for 8 projects

	WMC before	WMC after	DIT before	DIT after	NOC before	NOC after	CBO before	CBO after	RFC before	RFC after	LCOM before	LCOM after
Ant	0.413468	0.387218	0.231264	0.231264	0.097186	0.097186	0.280313	0.242124	0.465733	0.461385	0.44241	0.433243
Camel	0.344644	0.261803	0.107054	0.107054	0.213531	0.12707	0.288217	0.261893	0.272886	0.244297	0.386948	0.34033
Ivy	0.375992	0.375992	0.199868	0.199868	0.163955	0.163955	0.287368	0.287368	0.440741	0.43545	0.367989	0.341865
Jedit	0.376025	0.352129	0.375717	0.375717	0.082681	0.082681	0.314661	0.30574	0.428882	0.423683	0.406811	0.388525
Log4j	0.332702	0.332702	0.086023	0.086023	0.148946	0.148946	0.276537	0.276537	0.283161	0.245726	0.305675	0.293139
Lucene	0.231542	0.231542	0.186246	0.186246	0.134123	0.134123	0.20397	0.20397	0.227588	0.227588	0.248045	0.248045
Synapse	0.310788	0.293605	0.069251	0.069251	0.032687	0.032687	0.252455	0.252455	0.342733	0.342733	0.312242	0.308366
Xerces	0.220908	0.195065	0.266444	0.150238	0.23162	0.23162	0.264279	0.20545	0.256726	0.214591	0.195641	0.170003

- **F-Measure Calculation:**

Table.21. Values of F-measure before and after removing redundancy and after log transformation for camel for WMC

	WMC before removing redundancy	WMC after removing redundancy
camel 1.0	0.166667	0.156863
camel 1.2	0.310976	0.275168
camel 1.4	0.352941	0.337079
camel 1.6	0.318681	0.28481

Table.22. Values of F-measure before and after removing redundancy and after log transformation for camel for DIT

	DIT before removing redundancy	DIT after removing redundancy
camel 1.0	0.042553	0.042553
camel 1.2	0.114286	0.114286
camel 1.4	0.189723	0.189723
camel 1.6	0.135922	0.135922

Table.23. Values of F-measure before and after removing redundancy and after log transformation for camel for NOC

	NOC before removing redundancy	NOC after removing redundancy
camel 1.0	0.148148	0.097561
camel 1.2	0.228374	0.178439
camel 1.4	0.213439	0.15311
camel 1.6	0.255738	0.19305

Table.24. Values of F-measure before and after removing redundancy and after log transformation for camel for CBO

	CBO before removing redundancy	CBO after removing redundancy
camel 1.0	0.27451	0.255319
camel 1.2	0.22449	0.216783
camel 1.4	0.243542	0.237548
camel 1.6	0.274143	0.281046

Table.25. Values of F-measure before and after removing redundancy and after log transformation for camel for RFC

	RFC before removing redundancy	RFC after removing redundancy
camel 1.0	0.133333	0.111111
camel 1.2	0.27476	0.268852
camel 1.4	0.323024	0.321678
camel 1.6	0.28655	0.275229

Table.26. Values of F-measure before and after removing redundancy and after log transformation for camel for LCOM

	LCOM before removing redundancy	LCOM after removing redundancy
camel 1.0	0.207792	0.202899
camel 1.2	0.314815	0.294118
camel 1.4	0.366559	0.3391
camel 1.6	0.311475	0.303207

Table.27. Values of F-measure (mean) before and after removing redundancy and after log transformation for 8 projects

	WMC before	WMC after	DIT before	DIT after	NOC before	NOC after	CBO before	CBO after	RFC before	RFC after	LCOM before	LCOM after
Ant	0.413557	0.403352	0.214468	0.214468	0.138517	0.138517	0.315609	0.284876	0.483966	0.48523	0.425792	0.425948
Camel	0.287316	0.26348	0.120621	0.120621	0.211425	0.15554	0.254171	0.247674	0.254417	0.244218	0.30016	0.284831
Ivy	0.320064	0.320064	0.169239	0.169239	0.14541	0.14541	0.265502	0.265502	0.36131	0.359697	0.324836	0.313723
Jedit	0.364433	0.350947	0.317162	0.317162	0.095903	0.095903	0.297835	0.295492	0.40983	0.412394	0.37955	0.373287
Log4j	0.471274	0.471274	0.136403	0.136403	0.231498	0.231498	0.399753	0.399753	0.415426	0.379722	0.432218	0.424853
Lucene	0.368764	0.368764	0.275394	0.275394	0.217755	0.217755	0.318386	0.318386	0.359684	0.359684	0.377553	0.377553
Synapse	0.345965	0.335621	0	0	0	0	0.323484	0.323484	0.39757	0.401073	0.345704	0.346482
xerces	0.271312	0.257251	0.271058	0.187307	0.30436	0.30436	0.337628	0.294154	0.309448	0.290776	0.236909	0.219154

- **Accuracy Calculation:**

Table.28. Values of Accuracy before and after removing redundancy and after log transformation for camel for WMC

	WMC before removing redundancy	WMC after removing redundancy
camel 1.0	82.30%	87.32%
camel 1.2	62.83%	64.47%
camel 1.4	77.29%	79.70%
camel 1.6	74.30%	76.58%

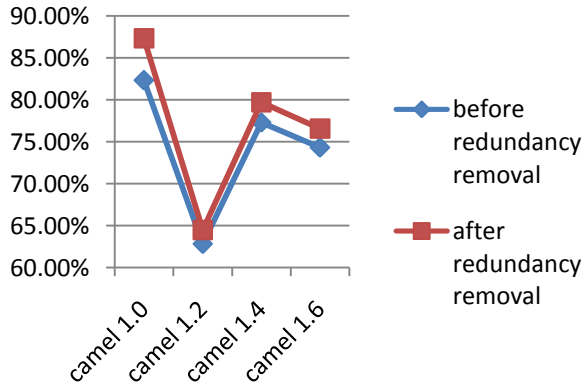

Figure.3. Comparison of accuracy before and after removing redundancy and after log transformation for camel for WMC

Table.29. Values of Accuracy before and after removing redundancy and after log transformation for camel for DIT

	DIT before removing redundancy	DIT after removing redundancy
camel 1.0	86.73%	86.73%
camel 1.2	59.21%	59.21%
camel 1.4	76.49%	76.49%
camel 1.6	72.33%	72.33%

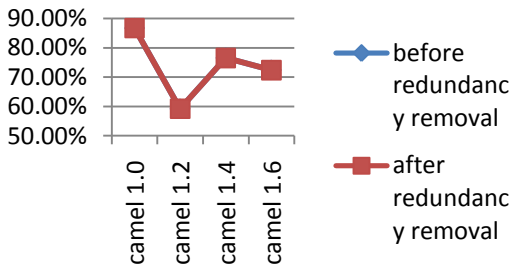

Figure.4. Comparison of accuracy before and after removing redundancy and after log transformation for camel for DIT

Table.30. Values of Accuracy before and after removing redundancy and after log transformation for camel for NOC

	NOC before removing redundancy	NOC after removing redundancy
camel 1.0	86.43%	89.09%
camel 1.2	63.32%	63.65%
camel 1.4	77.18%	79.70%
camel 1.6	76.48%	78.34%

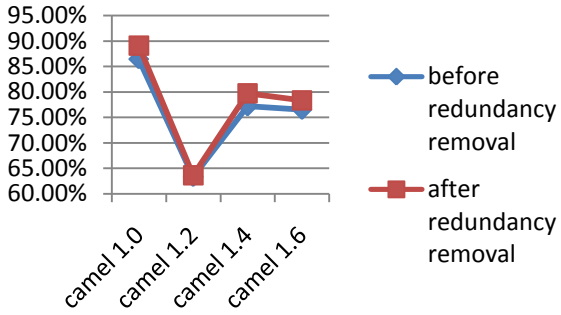

Figure.5. Comparison of accuracy before and after removing redundancy and after log transformation for camel for NOC

Table.31. Values of Accuracy before and after removing redundancy and after log transformation for camel for CBO

	CBO before removing redundancy	CBO after removing redundancy
camel 1.0	89.09%	89.68%
camel 1.2	62.50%	63.16%
camel 1.4	76.49%	77.18%
camel 1.6	75.85%	77.20%

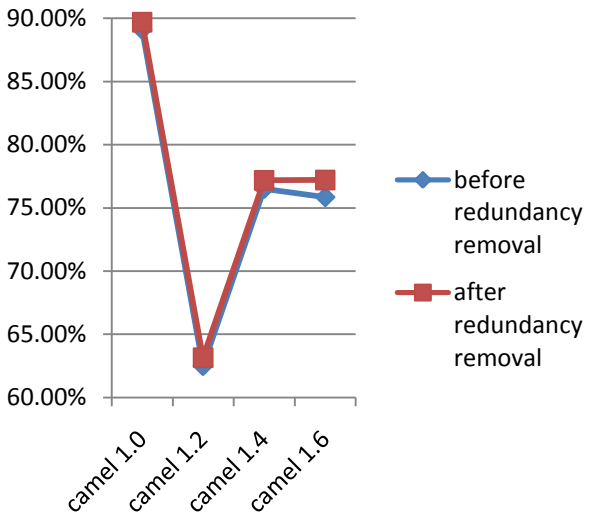

Figure.6. Comparison of accuracy before and after removing redundancy and after log transformation for camel for CBO

Table.32. Values of Accuracy before and after removing redundancy and after log transformation for camel for RFC

	RFC before removing redundancy	RFC after removing redundancy
camel 1.0	84.66%	85.84%
camel 1.2	62.66%	63.32%
camel 1.4	77.41%	77.75%
camel 1.6	74.72%	75.44%

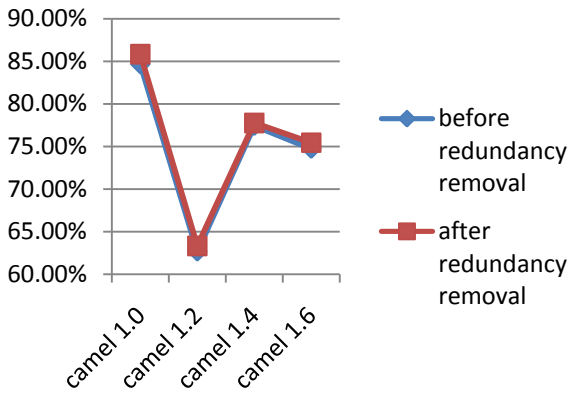

Figure.7. Comparison of accuracy before and after removing redundancy and after log transformation for camel for RFC

Table.33. Values of Accuracy before and after removing redundancy and after log transformation for camel for LCOM

	LCOM before removing redundancy	LCOM after removing redundancy
camel 1.0	82.01%	83.78%
camel 1.2	63.49%	64.47%
camel 1.4	77.41%	78.10%
camel 1.6	73.89%	75.23%

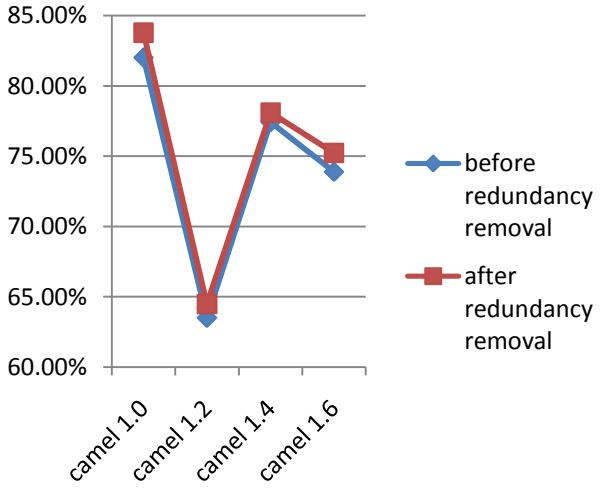

Figure.8. Comparison of accuracy before and after removing redundancy and after log transformation for camel for LCOM

Table.34. Values of Accuracy (mean) before and after removing redundancy and after log transformation for 8 projects

	WMC before	WMC after	DIT before	DIT after	NOC before	NOC after	CBO before	CBO after	RFC before	RFC after	LCOM before	LCOM after
Ant	77.95%	78.54%	68.37%	68.37%	76.63%	76.63%	77.10%	76.91%	81.22%	81.47%	77.41%	78.00%
Camel	74.18%	77.02%	73.69%	73.69%	75.85%	77.70%	75.98%	76.80%	74.86%	75.59%	74.20%	75.39%
Ivy	72.42%	72.42%	64.15%	64.15%	68.25%	68.25%	72.34%	72.34%	73.10%	73.17%	72.86%	73.23%
Jedit	79.40%	79.90%	72.84%	72.84%	74.40%	74.40%	77.45%	77.93%	80.24%	80.76%	78.87%	79.51%
Log4j	62.54%	62.54%	47.41%	47.41%	52.87%	52.87%	58.86%	58.86%	60.02%	59.26%	60.40%	60.32%
Lucene	56.37%	56.37%	46.56%	46.56%	47.31%	47.31%	52.19%	52.19%	55.55%	55.55%	55.24%	55.24%
Synapse	74.62%	75.01%	65.34%	65.34%	72.18%	72.18%	76.75%	76.75%	77.78%	78.00%	75.05%	75.41%
xerces	60.93%	62.23%	54.21%	56.21%	63.52%	63.52%	65.16%	65.67%	62.78%	64.35%	58.33%	59.81%

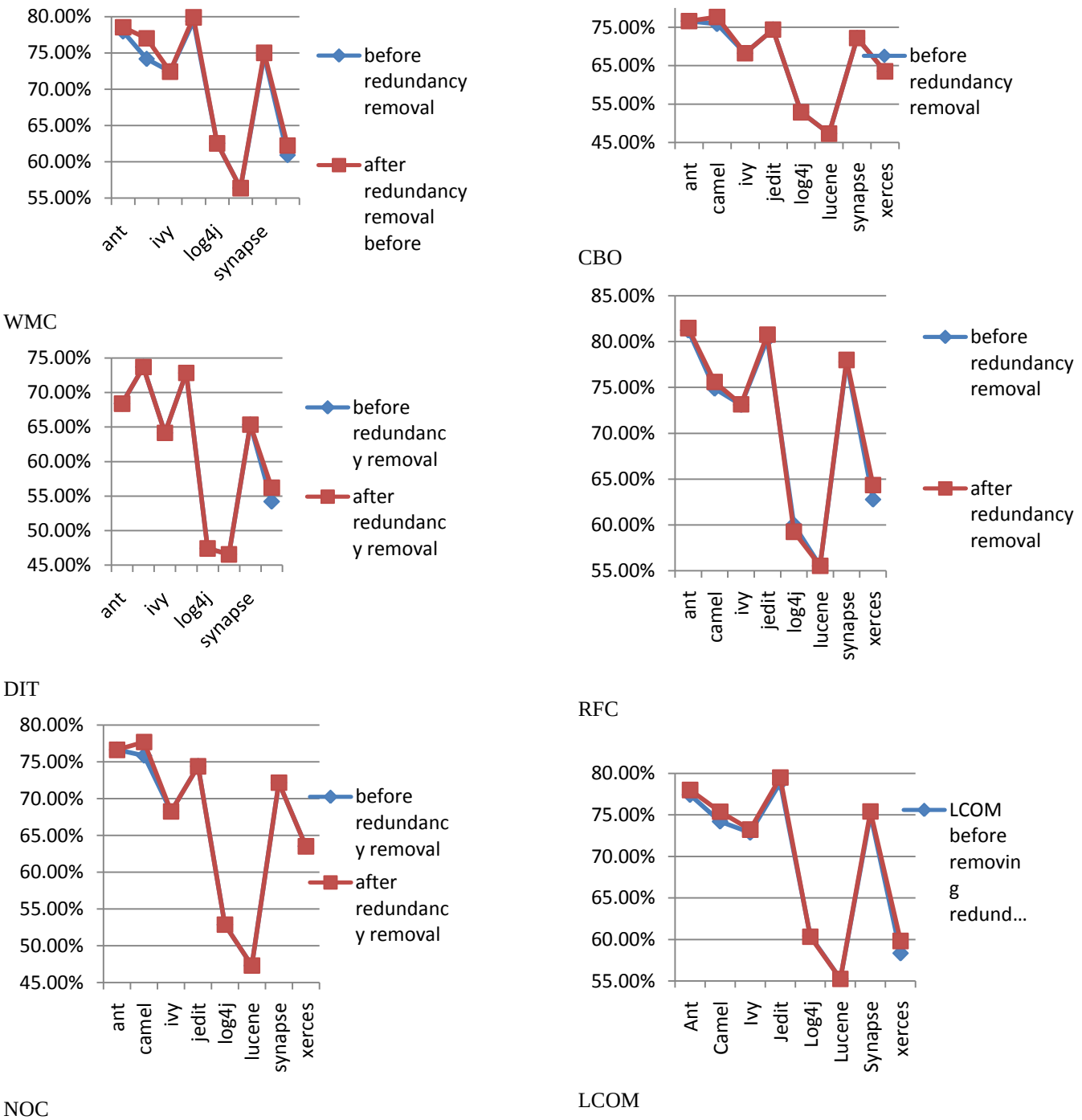

Figure. 9. Comparison of accuracy before and after removing redundancy and after log transformation for 8 projects

C. *Observation*

There were several values of threshold given based on previous research papers as shown in Table XXXV [44-45,25,34-41,11-17].

In this thesis we have observed several threshold values for 6 CK metrics namely, WMC, DIT, NOC, CBO, RFC and LCOM for all 8 Projects (30 releases) as shown in TABLE XXXVI.

Table.35. Summary of previously identified thresholds

	Rosenberg [35]	Shatnawi [15] (several levels of risk)	Shatnawi *et al.* [14]	Alves *et al.* [36]	Herbold *et al.* [46]	McCabe [21]	Ferreira *et al.* [17]
	No case studies	Three releases of Eclipse	Three releases of Eclipse	100 projects	Two large-scale open-source projects	No case studies	40 open-source Java software systems
WMC	100	20, 46, or 98	24	29, 42, and 73 at (70%, 80%, and 90%, respectively)	20	14	—
DIT	2 or 3	—	—	—	Not included	7	2 or 1
NOC	—	—	—	—	Not included	3	—
CBO	5	9, 16, or 29	13	—	5	2	—
RFC	100	40, 80, or 164	44	—	100	100	—
LCOM	—	—	—	—	Not included	75	20

Table.36.Threshold obtained after removing redundancy and after log transformation

Metric	Threshold values after removing redundancy after log transformation
WMC	17
DIT	3
NOC	1
CBO	17
RFC	51
LCOM	65

Based on these threshold values derived after removing redundancy from record values of datasets and log transformation, we have performed fault classification for each metric and thus obtained performance of fault prediction for each metric. We have observed accuracy of predicting fault after removing redundancy and after log transformation for each metric is better than accuracy of predicting fault before removing redundancy and after log transformation for each metric for all 8 projects(30 releases).

Table.37. Accuracy of predicting fault after removing redundancy and after log transformation and before removing redundancy and after log transformation

METRIC	Accuracy after removing redundancy and after log transformation	Accuracy before removing redundancy and after log transformation
WMC	70.50%	69.80%
DIT	61.82%	61.57%
NOC	66.61%	66.38%
CBO	69.68%	69.48%
RFC	71.02%	70.69%
LCOM	69.61%	69.05%

IV. CONCLUSION AND FUTURE ENHANCEMENT

Finding where quality can be improved is a vital issue in software quality and is one of the major uses of software metrics. Appropriate metric tools and analysis techniques are needed to identify the classes that are more fault-prone during both development and testing phases. Software practitioners can analyze software quality using metrics by setting threshold values to mark the most complex classes. However, the currently identified thresholds do not account for the skewness in data distribution. In this work, we proposed to remove redundancy and to use data transformation to improve two techniques of software quality assessment: derive threshold values and fault classification using the derived metrics. Data distribution has been used before to identify thresholds values using the mean and the standard deviation. However, most previous works have not considered for removing redundancy in dataset that ultimately result in over-fitting and scaling problems. So we use redundancy removal technique in dataset to reduce problem of over-fitting and scaling. along with that most previous works have not considered the effect of data transformation on quality assessment. We used the log transformation to reduce the effect of skewness in data. The results were attained from studying 8 different Java projects from the open-source field. To evaluate the effectiveness of the results obtained after transformation, we used the transformed metrics to derive thresholds. The derived thresholds were used to classify classes into either faulty (<thresholds) or not faulty otherwise, and we repeated the classification using the thresholds that were derived from metrics before the transformation. The statistical comparison showed better fault classification after redundancy removal and log-transformed data than otherwise. We suggest to remove redundancy and to use data transformation on software metrics before assessing software quality.

In the future, we intend to use complete suite of CK metric suite to derive threshold and perform fault classification and thus evaluate their performance and compare them against previous studies. In future, we plan to consider more factors that affect the derivation of consistent and practical thresholds. In addition, we plan to validate the effect of using thresholds on development and maintenance activities such as code refactoring. And in addition the researcher plans to expand this study to more diverse datasets.

V. REFERENCES

[1].Shatnawi R. Deriving metrics thresholds using log transformation. Journal of software: evolution and process, j. Softw. Evol. And proc. 2015; 27:95–113.

[2].Tamilselvi J, Gifta C. Handling Duplicate Data in Data Warehouse for Data Mining. International Journal of Computer Applications (0975 – 8887) Volume 15– No.4, 2011

[3].Cheng A. The Causes, Impact and Detection of Duplicate Observations. Pfizer, Inc., New York.

[4].Zimmermann T, Nagappan N, Zeller A. Predicting bugs from history. Software Evolution (Software Evolution), 2008; 69–88.

[5].Lincke R, Lundberg J, Löwe W. Comparing software metrics tools. Proceedings of the 2008 international symposium on Software testing and analysis, ISSTA, New York, NY, USA, 2008; 131–142.

[6].Baxter G, Frean M, Noble J, Rickerby M, Smith M, Visser M, Melton H, Tempero E. Understanding the shape of Java software. SIGPLAN Not 2006; 41(10):397–412.

[7].Barkmann H, Lincke R, Löwe W. Quantitative evaluation of software quality metrics in open-source projects. Proc Works. Advanced Information Networking and Applications, 2009; 1067–1072.

[8].Herraiz I, Rodriguez D, Harrison R. On the statistical distribution of object-oriented system properties, emerging trends in software metrics (WETSoM). 3rd International Workshop on 2012; 56–62.

[9].Concas G, Marchesi M, Pinna S, Serra N. Power-laws in a large object-oriented software system. IEEE Transactions on Software Engineering 2007; 33(10):687–708.

[10].Louridas P, Spinellis D, Vlachos V. Power laws in software. ACM Transactions on Software Engineering and Methodology 2008; 18(1):1–26.

[11].Erni K, Lewerentz C. Applying design–metrics to object–oriented frameworks. Proc. of the Third International Software Metrics Symposium 1996; 25–26.

[12].Benlarbi S, El Emam K, Goel N, Rai S. Thresholds for object–oriented measures. 11th International Symposium on

Software Reliability Engineering. IEEE Computer Society: Los Alamitos CA, 2000; 24–38.

[13].El Emam K, Benlarbi S, Goel N, Melo W, Lounis H, Rai S. The optimal class size for object–oriented software. IEEE Transactions on Software Engineering 2002; 28(5):494–509.

[14].Shatnawi R, Wei L, Swain J, Newman T. Finding software metrics threshold values using ROC curves. Journal of Software Maintenance & Evolution, Research & Practice 2010; 22(1):1–16.

[15].Shatnawi R. Quantitative investigation of the acceptable risk levels of object–oriented metrics in open–source systems. IEEE Transactions on Software Engineering 2010; 36(2):216–225.

[16].Catal C, Alan O, Balkan K. Class noise detection based on software metrics and ROC curves. Information Sciences 2011; 181(21):4867–4877.

[17].Ferreira K, Bigonha M, Bigonha S, Mendes L, Almeida H. Identifying thresholds for object–oriented software metrics. Journal of Systems and Software 2012; 85(2):244–257.

[18].Foucault M, Palyart M, Falleri J, Blanc X. Computing contextual metric thresholds. published in 29th Symposium on Applied Computing, Gyeongju, Korea, Republic Of, 2014.

[19].Oliveira P, Tulio F, Lima V. Extracting relative thresholds for source code metrics, IEEE CSMR- WCRE, Antwerp, Belgium, 2014; 254–263.

[20].Chidamber S, Kemerer C. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 1994; 20(6):476–493.

[21].McCabe Software. Using code quality metrics in management of outsourced development and maintenance, white paper, last accessed November 2013. Available from:http://www.mccabe.com/pdf/McCabeCodeQualityMetrics–OutsourcedDev.pdf.

[22].Radjenović D, Heričko M, Torkar R, Živkovič A. Software fault prediction metrics: a systematic literature review. Information and Software Technology 2013; 55(8):1397–1418.

[23].Jureczko M, Madeyski L. Towards identifying software project clusters with regard to defect prediction. Proceedings of the 6th International Conference on Predictive Models in Software Engineering, 2010; 1–10.

[24].Jureczko M, Spinellis D. Using object-oriented design metrics to predict software defects. Proceedings of the 5th International Conference on Dependability of Computer Systems, 2010: 69–81.

[25].Gyimothy T, Ferenc R, Siket I. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering 2005; 31(10):897–910.

[26].Olague H, Etzkorn L, Gholston S, Quattlebaum S. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes, IEEE Transactions on Software Engineering 2007; 33(8):402–419.

[27].Osborne J. Notes on the use of data transformations. Practical Assessment, Research & Evaluation 2002; 8(6). Available from: http://pareonline.net/getvn.asp?v=8&n=6.

[28].Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Publishing Company: One Jacob Way, Reading, Massachusetts, 1994.

[29].Gil J, Maman I. Micro patterns in Java code. OOPSLA, 2005; 97–116.

[30].Daly J, Brooks A, Miller J, Roper M, Wood M. Evaluating inheritance depth on the maintainability of object-oriented software, Journal of Empirical Software Engineering 1996; 1(2):109–132.

[31].Cartwright M. An empirical view of inheritance. Information and Software Technology 1998; 40(14):795–799.

[32].Prechelt L, Unger B, Philippsen M, Tichy W. A controlled experiment on inheritance depth as a cost factor for code maintenance. Journal of Systems and Software 2003; 65(2):115–126.

[33].Gronback R. Software remodeling: improving design and implementation quality, using audits, metrics and refactoring in Borland Together ControlCenter. A Borland White Paper, January 2003.

[34].Marinescu R. Measurement and quality in object-oriented design. Proceedings of the 21st IEEE International Conference on Software Maintenance ICM05, 2005; 701–704.

[35].Rosenberg L. Metrics for object oriented environment. Proc., EFAITP/AIE 3rd Annual Software Metrics Conference, 1997.

[36].Alves TL, Ypma C, Visser J. Deriving metric thresholds from benchmark data. Proceedings of the IEEE International Conference on Software Maintenance (ICSM ’10), September 2010; 1–10.

[37].Alves TL, Correia JP, Visser J. Benchmark-based aggregation of metrics to ratings. Proceedings of the 21st International Workshop and 6th International Conference on Software Process and Product Measurement (IWSM-MENSURA ’11), 2011; 20–29.

[38]. Sánchez-González L, Garcia F, Mendling J, Ruiz F. Quality assessment of business process models based on thresholds. On the Move Federated Conference - 18th International Conference on Cooperative Information Systems, Crete, Greece, October 2010; 27–29.

[39]. Sánchez-González L, Garcia F, Mendling J, Ruiz F. A study of the effectiveness of two threshold definition techniques. 16th International Conference on Evaluation and Assessment in Software Engineering (EASE 2012), Ciudad Real, Spain, 14-15 May 2012.

[40].Perez-Castillo R, Sánchez-González L, Piattini M, Garcia F, Garcia-Rodriguez I. Obtaining thresholds for the effectiveness of business process mining. Proceedings of the

International Symposium on Empirical Software Engineering and Measurement 2011; 453–462.

[41].Mendling J, Sánchez-González L, García F, La Rosa M. Thresholds for error probability measures of business process models. Journal of Systems and Software 2012; 85(5):1188–1197.

[42].Rosenberg LH, Stapko R, Gallo A. Risk-based object oriented testing. 24th Annual Software Engineering Workshop. Goddard Space Flight Center 1999.

[43].Basili V, Briand L, Melo W. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 1996; 22(10): 751–761.

[44]. Cartwright M, Shepperd M. An empirical investigation of an object-oriented software system. IEEE Transactions on Software Engineering 2000; 26(8): 786–796.

[45].Subramanyam R, Krishnan M. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Transactions on Software Engineering 2003; 29(4): 297–310.

[46].Herbold S, Grabowski J, Waack S. Calculation and optimization of thresholds for sets of software metrics. Empirical Software Engineering 2011; 16(6): 812–841.