



# **Investigation for Software Fault Proneness Prediction at Method Level**

By

**Samah Aldiabat**

Supervisors

**Prof. Dr. Bilal Abul-Huda**

**Dr. Mohamed Akour**

Computer Information Systems

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF REQUIREMENT FOR THE  
DEGREE OF THE MASTER OF COMPUTER INFORMATION SYSTEMS AT YARMOUK  
UNIVERSITY, IRBID, JORDAN.**

**December, 2018**

# Investigation for Software Fault Proneness Prediction at Method Level

By  
**Samah Aldiabat**

B.S c. Computer Information Systems/Jordan University of  
Science and Technology, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF REQUIRMENT  
FOR THE DEGREE OF THE MASTER OF COMPUTER INFORMAION  
SYSTEMS AT YARMOUK UNIVERSITY, IRBID, JORDAN.

Approved By:

**Bilal Abul-Huda** .....Advisor  
Professor of Management Information Systems, Yarmouk University.

**Mohammed Akour** ..... Co-advisor  
Associate Professor of Software Engineering, Yarmouk University.

**Ahmad Saifan**.....Member  
Associate Professor of Software Engineering, Yarmouk University.

**Yahya Tashtoush**.....Member  
Associate Professor of Computer Engineering, Jordan University of science and  
Technology.

December, 2018

## Acknowledgment

At the beginning and before any one I would like to thank God for my success and completing this thesis that otherwise I would not have.

Big thank to my supervisors Dr. Bilal and Dr. Mohammed for the support and follow-up that gives me the appropriate guidance to complete this thesis.

Great thanks and gratitude to my family who supported me at all times and encouraged me to reach the top.

Also, many thanks to my friends and especially Hiba for supporting and helping me in the most difficult times.

# Table of Contents

List of Tables .....	VI
List of Figures.....	VII
المخلص.....	VIII
Abstract.....	IX
Chapter One .....	1
1. Introduction: .....	1
1.1 General Overview .....	1
1.2 Background .....	4
1.3 Problem Statement.....	7
1.3.1 Research Purpose.....	7
1.3.2 Research Motivation.....	10
1.3.3 Research Questions.....	10
1.3.4 Research Significance .....	11
1.3.5 Operational Definitions .....	11
Chapter Two.....	12
2. Literature Review .....	12
2.1 Software fault, bug and defect .....	12
2.2 Software Fault Proneness prediction .....	13
2.3 Software Fault Prediction using Machine Learning Techniques .....	16
2.4 Software Fault Prediction Using Metrics.....	21
Chapter Three .....	24
3. Research Methodology .....	24
3.1 Overall Research Design .....	24
3.2 Research Phases .....	25
3.2.1 Phase one: Collecting Java codes.....	25
3.2.2 Phase Two: Producing OO metrics on the method level with JAVA dataset using IntelliJ IDEA tool (Creating Dataset 1) .....	29
3.2.3 Phase Three: Producing faulty & non-faulty methods using manual error seeding (Creating Dataset 2).....	30
3.2.4 Phase Four: Applying OO metrics and machine learning techniques on Java Dataset .....	31
3.2.5 Phase Five: Comparing between OO metrics and historical metrics.....	32

<b>3.2.6 Phase Six: Comparing between machine learning techniques that used on the method level</b>	<b>33</b>
3.2.6.1 Naïve Bays Algorithm:	33
3.2.6.2 J48 Algorithm:	34
3.2.6.3 Decision Table Algorithm:	35
3.2.6.4 Random Forest Algorithm:	37
3.2.6.5 SVM Algorithm:	38
<b>3.3 Datasets</b>	<b>40</b>
<b>3.4 Research Tools and Applications</b>	<b>44</b>
<b>Chapter Four</b>	<b>45</b>
<b>4. Experiment Setup</b>	<b>45</b>
4.1 Evaluation Measures	45
4.2 Experiment 1: Extracting Metrics	46
4.3 Experiment 2: Error Seeding and Mutation	46
4.4 Experiment 3: implementation	49
<b>Chapter Five</b>	<b>58</b>
<b>5. Results Discussion</b>	<b>58</b>
<b>Chapter Six</b>	<b>60</b>
<b>6. Conclusion and Future Work</b>	<b>60</b>
<b>7. References</b>	<b>62</b>

## List of Tables

TABLE 1: OBJECT ORIENTED METRICS USED (JETBRAINS.COM/IDEA, 2018).....	8
TABLE 2: HISTORICAL METRICS DESCRIPTION (HATA, ET AL, 2012).....	33
TABLE 3: DETAILS OF DATASETS .....	40
TABLE 4: NOT USED OBJECT ORIENTED METRICS (JETBRAINS.COM/IDEA, 2018).....	42
TABLE 5: EVALUATION MEASURES .....	45
TABLE 8: NAIVE BAYS ALGORITHM ON LARGE SCALE PROCESSED DATASET .....	49
TABLE 9: NAIVE BAYS ALGORITHM ON LARGE SCALE UNPROCESSED DATASET.....	49
TABLE 10: J48 ALGORITHM ON LARGE SCALE PROCESSED DATASET .....	50
TABLE 11: J48 ALGORITHM ON LARGE SCALE UNPROCESSED DATASET .....	50
TABLE 12: DECISION TABLE ALGORITHM ON LARGE SCALE PROCESSED DATASET .....	50
TABLE 13: DECISION TABLE ALGORITHM ON LARGE SCALE UNPROCESSED DATASET .....	51
TABLE 14: RANDOM FOREST ALGORITHM ON LARGE SCALE PROCESSED DATASET .....	51
TABLE 15: RANDOM FOREST ALGORITHM ON LARGE SCALE UNPROCESSED DATASET .....	51
TABLE 16: SVM ALGORITHM ON LARGE SCALE PROCESSED DATASET.....	51
TABLE 17: SVM ALGORITHM ON LARGE SCALE UNPROCESSED DATASET .....	52
TABLE 18: NAIVE BAYS ALGORITHM ON SMALL SCALE PROCESSED DATASET .....	52
TABLE 19: NAIVE BAYS ALGORITHM ON SMALL SCALE UNPROCESSED DATASET .....	53
TABLE 20: J48 ALGORITHM ON SMALL SCALE PROCESSED DATASET.....	53
TABLE 21: J48 ALGORITHM ON SMALL SCALE UNPROCESSED DATASET.....	54
TABLE 22: DECISION TABLE ALGORITHM ON SMALL SCALE PROCESSED DATASET .....	54
TABLE 23: DECISION TABLE ALGORITHM ON SMALL SCALE UNPROCESSED DATASET .....	55
TABLE 24: RANDOM FOREST ALGORITHM ON SMALL SCALE PROCESSED DATASET.....	55
TABLE 25: RANDOM FOREST ALGORITHM ON SMALL SCALE UNPROCESSED DATASET .....	56
TABLE 26: SVM ALGORITHM ON SMALL SCALE PROCESSED DATASET .....	56
TABLE 27: SVM ALGORITHM ON SMALL SCALE UNPROCESSED DATASET.....	57
TABLE 28: COMPARISON OF ERROR RATE .....	59

## List of Figures

FIGURE 1: RESEARCH METHODOLOGY .....	24
FIGURE 2: JAVA SOURCE CODE .....	25
FIGURE 3: INTELLIJ IDEA TOOL RESULTS (METRICS) .....	29
FIGURE 4: METHOD BEFORE ERROR SEEDING .....	30
FIGURE 5: METHOD AFTER ERROR SEEDING .....	30
FIGURE 6: CLASS LABEL .....	31
FIGURE 7: METRICS AFTER ERROR SEEDING .....	32

## الملخص

الذيات، سماح عبدالعزيز. التحقق من تنبؤ البرمجيات بالتعرض للخطأ على مستوى الوحدة البرمجية. ماجستير في نظم المعلومات الحاسوبية، رسالة، قسم نظم المعلومات الحاسوبية، جامعة اليرموك، 2018. (المشرفون: أ.د. بلاب أبو الهدى، د. محمد عكور)

يركز التنبؤ بأخطاء البرمجيات على فحص واختبار الملفات والوحدات البرمجية لإظهار احتمال وجود أخطاء موجودة أم لا، ذلك يؤدي إلى بناء نموذج للتنبؤ يعتمد على مقاييس الجودة باستخدام السمات الداخلية والبيانات الخاطئة. بعد التنبؤ بالوحدة البرمجية الخاطئة، يمكن استخدام عملية التصحيح لفحص واختبار الوحدة البرمجية. و عندما يركز الجهد على الوحدات البرمجية التي تحتوي على أخطاء ، ذلك يمكن أن يساعد في الاستفادة من موارد البرامج وإدارتها بشكل فعال ، مما سيعزز مرحلة الصيانة لتكون سهلة.

في هذه الأطروحة ، يتم استخدام خمسة خوارزميات تعلم الآلة للتنبؤ بالوحدات البرمجية الخاطئة وهي Naïve Bays و SVM , Decision Table , J48, Random Forest. هذه الخوارزميات تم تطبيقها على قواعد بيانات جافا (المقاييس الكبيرة والصغيرة) التي تم معالجتها مسبقاً للحصول على نتائج أفضل. المقارنات التي تمت بين مجموعات البيانات غير المعالجة و المعالجة تبعاً لخوارزميات التعلم الآلي تشير الى ان هناك اختلاف في النتائج. بعد مقارنة النتيجة في منهجنا المطور مع (Hata, et al, 2012) اعتماداً على معدل الخطأ، من الواضح أن النتيجة في نهجنا المتطور كانت أفضل من نتائجهم باستخدام Random Forest.

استناداً إلى المقارنة بين مجموعات البيانات التي سبق معالجتها وغير المعالجة لمجموعة البيانات كبيرة الحجم، كانت البيانات المعالجة أفضل قليلاً من البيانات غير المعالجة. و بالاستناد إلى المقارنة بين البيانات المعالجة وغير المعالجة لمجموعات البيانات صغيرة الحجم ، فمن الواضح أنه لا يوجد أي تأثير للمعالجة المسبقة لمجموعة البيانات للحصول على نتائج أفضل.

بناء على نتائج مقاييس التقييم توضح أن أفضل خوارزمية للدقة ومعدل الخطأ هي Random Forest والترتيب

التنازلي للخوارزميات هو Decision Table , Random Forest , SVM , J48 و الأخير هو Naïve Bays.



## Abstract

**Aldiabat, Samah Abedalaziz. Investigation for Software Fault Proneness Prediction at Method Level. Master of Computer Information Systems, Thesis, Department of Computer Information Systems, Yarmouk University, 2018. (Supervisors: Prof. Dr. Bilal A. Abul-Huda, Dr. Mohammed A. Akour)**

Software fault prediction is focusing on examining and testing files, packaging, classes or methods to show the probability of existing faults or not. That led to build a predictive model based on quality metrics by using internal attributes and faulty data. After predicting the faulty method, the correction process can be used to inspect and test the method. When effort focuses on the methods that have faults, it can help in utilizing and managing the software resources effectively and that will enhance the maintenance phase to be easy.

In this thesis, five machine learning algorithms are used which are, Naïve Bays, Random Forest, J48, Decision Table and SVM to predict the faulty methods. These algorithms applied on Java Datasets (Large and Small scales) contains object oriented metrics (B, CALL, CLOC, COM\_RAT, D, E, EXEC, EXP, IV(G), LOC, N, n, NCLOC, NP, STAT, QCP\_CRCT, QCP\_MAINT, QCP\_RLBTY, TCOM\_RAT, V,V(G)), that normalized and preprocessed to gain better results. Comparisons done between preprocessed and unprocessed datasets depending on the machine learning algorithms and there was a variation in the results. The result in our developed approach is compared with (Hata, et al, 2012) depending on error rate. It is obvious that the result in our developed approach using Random Forest is better than their results.

Based on comparison between preprocessed and unprocessed datasets for the large scale as, it is obvious that the processed data is a little bit better than the unprocessed data for the large scale datasets. While based on comparison between processed and unprocessed data for

the small scale datasets as, it is obvious that there is no effect of preprocessing the dataset to get better results.

The evaluation measures results show that for the accuracy and error rate the best algorithm is Random Forest and the descending order for the algorithms is: Random Forest, Decision Table, J48, SVM and the last one is Naïve Bays. While for the precision the descending order is Naïve Bays, Random Forest, J48, SVM and Decision Table. For the recall the order is Decision Table, SVM, J48, Naïve Bays and Random Forest. The false-positive order is Decision Table, SVM, J48, Random Forest and Naïve Bays. F-measure order is Random Forest, SVM, Decision Table, J48 and Naïve Bays. Finally, the specificity and True-negative order is Naïve Bays, Random Forest, J48, Decision Table and SVM.

**Keywords:** Fault prediction, Fault Proneness, machine learning, Object Oriented Metric

# Chapter One

## 1. Introduction:

### 1.1 General Overview

Software systems significant role in the applications that have critical mission, demands working in a reliable way with their requirements. A comprehensive assessment for these software systems using manual testing or automatic techniques is needed for assuring the software quality. To verify the areas that have problems in the system under the development, predicting the modules that are fault prone by software quality models is needed to help the experts. Therefore, by enforcing the software quality models at early stages of the software development life cycle might help in producing reliable software by efficiently removing faults.

Software testing is the procedure of executing the program with the purpose of detecting errors and making sure the software does what it supposes to do. Typically, programs include large number of errors. One of the reasons for continuing these errors over the software development life cycle is the restrictions of the testing resources, such as the time and cost. To produce software system with high reliability, high quality and low cost, the resources should be used in an effective way through concentrating the effort of testing on the system parts that contains more errors (Banitaan, et al, 2013).

Software quality can be measured according to different attributes; one of these attributes is the fault proneness. Fault proneness is defined as “the probability of fault detection in a class”. This means that fault proneness is the probability to be fault prone.

The importance of measuring the software fault proneness can be clear in minimize the cost and improve the overall testing process effectiveness. Fault proneness of the software cannot be measured directly, It can be estimated by using the software metrics to provide descriptions of the attributes of the program, and that descriptions are quantitative. (Malhotra and Jain, 2012).

By taking into account software size and complexity, producing software with high quality without faults is a complex task and big challenge to achieve. The most costly and challenging phase in the system development life cycle is the maintenance phase. To deal with this challenge, we must identify which parts of source code that perhaps include faults and need to be changed. Software fault prediction is focusing on examining and testing files, packaging, classes or methods to show the probability of existing faults or not. The solution is to build a predictive model based on quality metrics by using internal attributes and faulty data that collected previously, the most repeatedly dependent variable is the fault proneness. After predicting the faulty class, the correction process can be used to inspect and test the class. When effort focuses on the classes that have faults, it can help in utilizing and managing the software resources effectively and that will enhance the maintenance phase to be easier than before (Alenezi, et al, 2014).

There are different approaches for prediction in the area of software engineering like, correction cost prediction, test effort prediction, reusability prediction, fault prediction, quality prediction, security prediction and effort prediction. But most of these approaches need more research to reach the model that is robust. The most common research area in the prediction approach is the software fault prediction (Catal, 2011).

Producing a system with high robustness, reliability, efficiency and with no errors is critical. So fault prediction techniques must use in an efficient and accurate way. The purpose of the software fault prediction is categorizing the modules that under the test into error free or error prone modules. This categorization of the modules is a major step in the early phases of the software development life cycle especially in the testing phase, as exhaustive testing is impossible and costly. Machine learning techniques are used widely for building predictive models (Akour, et al, 2017).

Accurate prediction model of the faulty software depends on the availability of the software metrics information; also it depends on the quality metrics. So the main part of the process of model building is selecting the subset of software metrics and that will save the time in collecting and managing them. And select the appropriate classifier that known as the fault predictors from the machine learning techniques (Alenezi, et al, 2014).

Most researches investigate the fault prediction field, especially on the class level and used metrics and machine learning techniques to build the predictive model. There is a noticeable lack of research that are interests in the fault prediction on the method level and these researches not using all the techniques that used in the class level. We assume that addressing the fault proneness at the method level might be providing more promises. Since the method level is better than the class level, because of its effectiveness in the quality assurance. File level prediction is more efficient than package level prediction. So, method level prediction is more efficient than file level and package level prediction, and that means finding more bugs through the activities of quality assurance in the method level prediction when same amount of lines of code investigated is possible (Hata, et al, 2012). We will investigate if the fault proneness prediction on the method level will be efficient

according to the importance of the fine grained level and its effectiveness in discovering more faults because of the more details in the method level rather than the class level.

The remainder of this thesis is organized as follows; background, problem statement. The literature review was in chapter two. Chapter three presents our methodology, while the experiments discussed in chapter four. In chapter five the results discussion was presented and finally the conclusion and future work were in chapter six.

## 1.2 Background

Software dependency and complexity cause in increasing the need to deliver maintainable software, with high quality and low cost. Therefore, software fault prediction is considered as an important activity for improving the quality of software and reducing the effort of maintenance before deploying the system. To build a predictive model, metrics, predictors, faulty data is needed. Software fault prediction can categorize the module or the class to be either '*not fault-prone*', or '*fault-prone*'. Techniques of machine learning can be used in software fault proneness prediction (Malhotra, 2015, Rathore and Kumar, 2017).

There are different approaches for prediction in software engineering like, correction cost prediction, test effort prediction, reusability prediction, fault prediction, quality prediction, security prediction and effort prediction. But most of these approaches need more research to reach the model that is robust. The fault data in the module expressed by 1, else 0 when the error is notified through the test of the system or the field. The software metrics are utilized as the independent variables and the fault data is utilized as the dependent data in modeling prediction.

Thus, the need of version control system like Subversion for storing the source code, a change management system like Bugzilla for recording the faults and the tool that collecting the product metrics from the version control system. One of the techniques for software fault prediction is applying X-means method for clustering modules and identifying the top number of cluster. After that, checking the mean vector of every cluster against vector of metrics thresholds is necessary. So if the mean vector has one metric at least higher than the threshold value of the same metric, the cluster is predicted as “fault prone”. Another clustering methods used are the K-means and fuzzy. Experiments show that by using the X-means clustering, the software fault prediction that is not acting under supervision can produce efficient results and be completely automated. By using the algorithms of supervised classification, the model of prediction is built with the prior labels of fault and prior software metrics in machine learning (Catal, 2011).

One of the machine learning techniques is called ensemble learning which is combines more than one algorithm of machine learning and trained them for producing output better than the output of anyone of them separately. There are two types of ensemble machine learning techniques, heterogeneous and homogenous ensembles. For the heterogeneous ensemble, it develops each type of the base learner in different way by using several techniques of Machine learning. By merging each prediction of the base learner together, the dataset and the prediction are created. While for the homogeneous ensemble, it uses different subsets of the whole training dataset for each base learner. To produce satisfied conditions and to reach the suitable ensemble, there are two vital and primary conditions; which are the accuracy and diversity (Akour, et al, 2017).

The most repeatedly dependent variable is the fault proneness. When predict the fault proneness classes, it can focus on the chance of verification and validation in finding the faults. After predicting the faulty class, the actions of correction can be used to inspect and test the class. When the effort focuses on the classes that have faults, it can help in utilizing and managing the software resources effectively and that will enhance the maintenance phase to be easier than before (Alenezi, et al, 2014).

For estimating the fault proneness, model predicted using the QMOOD (Quality Model Object Oriented Design) and OOCK (Object Oriented Chidamber and Kemerer) metrics by applying six methods of machine learning and one method statistic. Different attributes can measure the software quality like, testing effort, fault proneness and testing effort. Machine learning is used in several domains such as, bioinformatics, retail companies, and financial institutions. The methods of machine learning used for predicting the accuracy of the predicted model (genetic programming, multilayer perceptron, support vector machine, adaboost, bagging and random forest). Apache POI dataset was used for the applications that extract text like, content management systems, web spiders and index builders. The techniques of machine learning used for predicting the accuracy of models when used more than one metric together. Decision trees have been used to predict the fault proneness, while the artificial neural networks used in predicting value of fault proneness continuous measure. As for the support vector machine, it is used to perform class classification to non-fault prone and fault prone. The other techniques of machine learning used for predicting the classes with faults like, boosting, random forest and bagging. These techniques can be used in WEKA tool (Malhotra, and Jain, 2012).



## 1.3 Problem Statement

### 1.3.1 Research Purpose

To the best of our knowledge, the work presented in this thesis trying to be a new contribution in the field of method level addressing for prediction purposes. This thesis tried to predict the fault proneness on the method level using object oriented metrics (B, CALL, CLOC, COM\_RAT, D, E, EXEC, EXP, IV(G), LOC, N, n, NCLOC, NP, STAT, TCOM\_RAT, QCP\_CRCT, QCP\_MAINT, QCP\_RLBTY, V,V(G)), shown in table 1 below, as features applied on JAVA datasets and using several machine learning techniques to build the predictive model such as Decision Table, SVM, Naïve Bays, J48 and Random Forest.

These metrics were chosen on the base of the experiment and information gain using WEKA tool which is a good measure for deciding the relevance attributes with maximal information and most effective features and removes the unrelated features that haven't any effect; after running the 44 extracted metrics from IntelliJ IDEA tool on WEKA tool and see how the variation and the results were, we decided to choose the 18 metrics that have an effective results.

Also, this thesis tried to compare between these techniques to determine the best technique for software fault proneness prediction on the method level. As another aim, planning to compare between the performance of the historical metrics and the object oriented metrics that applied on different datasets (JAVA Open Source Projects) to conclude the most appropriate metrics for the software fault proneness prediction on the

method level through measuring the performance measures such as accuracy, error rate, precision, recall, F-measure, False positive, True Negative and Specificity.

**Table 1: Object Oriented Metrics used (jetbrains.com/idea, 2018)**

#	Metrics	Description
1	B	Calculates the Halstead Bugs metric for a method. The Halstead Bugs is intended as an estimate of the number of bugs in a method. In practice, it has usually been found to underestimate. ( $B=V/3000$ )
2	CALL	Calculates the total number of method call expressions in each method.
3	CLOC	Calculates the number of lines of comments in each method. Whitespace is not counted for purposes of this metric.
4	COM_RAT	Calculates the ratio of lines of comments to total lines of code in each method. Whitespace is not counted for purposes of this metric.
5	D	Calculates the Halstead Difficulty metric for a method. The Halstead Difficulty is intended to correspond to the level of difficulty of understanding a method. ( $D=(n_1/2)*(N_2*n_2)$ )
6	E	Calculates the Halstead Effort metric for a method. The Halstead Effort is intended to correspond to the level of effort necessary to understand a method. ( $E=D*V$ )
7	EXEC	Calculates the total number of executable statements in each method. Executable statements are defined to be any non-control statement.
8	EXP	Calculates the total number of expressions in each method.
9	iv(G)	Calculates the design complexity of a method. The design complexity is related to how interlinked a methods control flow is with calls to other methods. Design complexity ranges from 1 to v(G), the cyclomatic complexity of the method.
10	LOC	Calculates the number of lines of code in each method. Comments are counted for purposes of this metric, but whitespace is not.
11	N	Calculates the Halstead Length metric for a method. The Halstead length of a method is defined as the total number of operators and operands in a method. ( $N=N_1+N_2$ )
12	n	Calculates the Halstead Vocabulary metric for a method. The Halstead Vocabulary of a method is defined as the total number of distinct operators and operands in a method. ( $n=n_1+n_2$ )
13	NCLOC	Calculates the number of non-comment lines of code in each method. Comment and empty lines are not counted by this metric.
14	NP	Calculates the number of parameters for each method.

15	STAT	Calculates the total number of statements in each method.
16	TCOM_RAT	Calculates the ratio of lines of comments to total lines of source code in each method. Whitespace is not counted for purposes of this metric.
17	QCP_CRCT	Calculates the Quality Criteria Profile (Correctness) of a method. This is a synthetic metric, designed to estimate the difficulty of determining the correctness of given method. Lower scores are better. Quality Criteria Profile (Correctness) is defined as: $QCP\_CRCT = D + CONTROL + EXECUTABLE + (2*V(g))$
18	QCP_MAINT	Calculates the Quality Criteria Profile (Maintainability) of a method. This is a synthetic metric, designed to estimate the difficulty of maintenance for a given method. Lower scores are better. Quality Criteria Profile (Maintainability) is defined as: $QCP\_MAINT = (3*N) + EXEC + CONTROL + NEST + (2*V(g)) + BRANCH$
19	QCP_RLBTY	Calculates the Quality Criteria Profile (Reliability) of a method. This is a synthetic metric, designed to estimate the reliability of given method. Lower scores are better. Quality Criteria Profile (Correctness) is defined as: $QCP\_RLBTY = N + (2*NEST) + (3*V(g)) + BRANCH + CONTROL + EXEC$
20	V	Calculates the Halstead Volume metric for a method. The Halstead Volume is intended to correspond to the size of a method, and is defined as $N * \log(n)$ , where N is the Halstead Length metric for the method and n is the Halstead Vocabulary metric. ( $V=N*\log_2n$ )
21	V(G)	Calculates the cyclomatic complexity of each non-abstract method. Cyclomatic complexity is a measure of the number of distinct execution paths through each method. This can also be considered as the minimal number of tests necessary to completely exercise a method's control flow. In practice, this is 1 + the number of if's, while's, for's, do's, switch cases, catches, conditional expressions, &&'s and   's in the method.

- $N_1$  = the total number of operators
- $N_2$  = the total number of operands
- $\eta_1$  = the number of distinct operators
- $\eta_2$  = the number of distinct operands

### 1.3.2 Research Motivation

Many researchers interest in the software fault prediction to be the evolutionary research because of the importance of this topic in the software field. They build predictive models to predict the fault, defect or bug in the software for many reasons that useful for the software development life cycle. Most of the researches in this topic are examined on the class level and package level, but rarely found researches that examine the predictive models on the method level. Fault proneness prediction on the method level examined using some random forest technique combined with historical metrics (Hata, et al, 2012). But they didn't examine more than one machine learning techniques combined with object oriented metrics. This thesis aims to get new good results by using the proposed technique.

### 1.3.3 Research Questions

- Do object oriented metrics predict the software fault proneness on the method level effectively?
- Which category of metrics is the best for the fault proneness prediction on the method level?
- To which extent the studied metrics accomplish better performance in terms of software fault proneness prediction?
- Which fault proneness prediction techniques is the most suitable for the method level according the dataset applies on?

### 1.3.4 Research Significance

This thesis desires to study the process of enhancing the fault proneness prediction on the method level and producing high quality and reliability software, and how that can leads to decrease the cost and time of the maintenance and complexity when predicting the fault prone methods at the early stage of the software development life cycle.

### 1.3.5 Operational Definitions

- **Software defect:** “an imperfection or deficiency in a software product where the product does not meet its requirement or specifications and needs to be either repaired or replaced” (Hong, 2017).
- **Software testing:** is the procedure of executing the program with the purpose of detecting errors (Banitaan, et al, 2013).
- **Software fault proneness:** “the probability of fault detection in a class” (Malhotra and Jain, 2012).
- **Software fault proneness prediction model:** a model of classification the software design entities into two categories; fault prone and non-fault prone (Scanniello, et al, 2013).
- **Random Forest:** “ensemble classifier that manipulates its input features and uses decision trees as its base classifiers” (Hong, 2012).
- **Error Seeding:** “is one of the white box testing which is very fascinating to researcher due to its approach to improve quality of software” (Gupta, 2016).

## Chapter Two

### 2. Literature Review

#### 2.1 Software fault, bug and defect

Software defect as defined in (Hong, 2017) is “an imperfection or deficiency in a software product where the product does not meet its requirement or specifications and needs to be either repaired or replaced”. The model of software fault prediction based on the metrics received the modules or the classes that quantified as a metric vector and predicted the fault information. Classifications that are binary have an importance in the researches which are determine if the module is fault prone or not. Defect attributes such as priority or severity is not considered in the modules that predict the absence or presence of the fault and that is a main problem in it. Defect severity is defined as “A measure of the impact a defect has on a system and its users”. The capability of predicting the modules that fault proneness in different categories of severity like low, high and not fault prone is much better than binary classifications, because not all the defects have same severity. Through predicting the critical problem in the system, the fault prediction model of the severity enables the resource allocation, quality testing and refactoring with lower cost.

It is difficult to find bugs and fix them. Also, it is costly. And recently, there are several techniques and tools developed to find bugs automatically through analyzing the source code. (Rutar, et al, 2004) applied five different tools to find bugs, on various java programs especially Bandera, ESC/Java 2, FindBugs, JLint, and PMD, They used diversity of tools to be able to find warnings and bug reports. Their experimented results offer that

the tools didn't cross over another means that the tools almost detect non overlapping bugs. They discussed for each tool which techniques is based on, and they proposed the output of the tools that affected by each techniques. finally they proposed a meta tools that joins the output from the tools with each other's, by take into account set of standards that many tools alert about such as particular lined of code, classes and methods.

Models that have defects of course cause failures in the system, increase the cost of the maintenance and development and decrease the satisfaction of the customers. In order to improve quality assurance of the software and to help the developer to focus on the fault prone modules by applying the activities of the quality assurance on it, the fault prediction model is needed (Koru and Liu, 2005).

## **2.2 Software Fault Proneness prediction**

Different attributes can measure the software quality and one of them is the software fault proneness. Software fault proneness is considered as dependent variable and it defined as “the probability of fault detection in a class” (Malhotra and Jain, 2012).

Testing, software quality and software fault proneness become more important in recent years regarding of improving efficiency of the process and minimizing the cost. Software fault proneness estimating in the model is significant to minimize the cost and to improve the efficiency of the process of software testing. We cannot measure the software fault proneness of the software directly. We can estimate it by using the software metrics to provide descriptions of the attributes of the program, and that descriptions are quantitative.

Most of the studies work on finding the suitable software metrics that used in predicting the fault proneness (Gondra, 2008, Singh, et al, 2009).

According to (Hong, 2012) software fault proneness prediction model is a model of classification the software design entities into two categories; fault prone and non fault prone. The capability of predicting the modules that fault proneness in different categories of severity like low, high and not fault prone is much better than binary classifications, because not all the defects have same severity (Scanniello, et al, 2013).

Software fault proneness may be predicted using machine learning methods by using one metric or a more than one metric together (Malhotra and Jain, 2012, Gondra, 2008, Singh, et al, 2009, Rathore and Kumar, 2017). For example, decision table have been used to predict the fault proneness, while the artificial neural networks used in predicting value of fault proneness continuous measure (Malhotra and Jain, 2012).

In (Gondra, 2008), the author used the support vector machine (SVM) technique to support the software fault proneness and classified the module to be with errors or with no errors. In (Singh, et al, 2009), they found that SVM is achieving high accuracy in predicting the fault proneness of the software. They found the metrics that are related to fault proneness on the class level which are, SLOC (Source Lines of Codes), RFC (Response For Class) and CBO (Coupling Between Object). Also, they concluded that SVM model achieves the feasibility, adaptability to the object oriented systems and it is useful for the fault proneness prediction for the classes.

Malhotra, et al, 2010 used the object oriented metrics to predict the fault proneness and used the SVM machine learning technique to build the model of fault proneness, assessing the software quality and to decide the feasibility and adaptability of this study.



They use the ROC (Receiver Operating Characteristic) evaluation measure to validate the SVM results and consider the accuracy of the predicted results from the ROC curve. Also, they considered the accuracy of fault proneness predicting using object oriented metrics and classified the faults according the severity into, low severity, medium severity and high severity. The dataset used is KC1 NASA dataset (C++ dataset) to evaluate their work. They found that SVM is predicting classes with faults with high accuracy. The object oriented metrics that are related to fault proneness are SLOC, RFC and CBO while DIT (Depth of Inheritance Tree) and NOC (Number Of Children) are not related to fault proneness. The model that predicted with concern to the faults with high severity will has low accuracy. Therefore, the best result in fault proneness prediction is for the faults of medium severity.

The appropriate metrics for the software fault proneness prediction are process metrics such as; size, complexity, design features, performance, and quality level and product metrics such as; Mean Time to Failure, Defect Density, Customer Problems and Customer Satisfaction (Luo, et al, 2010).

The usefulness of the object oriented metrics for the software fault proneness was studied by (Yu, et al, 2002). They used a tool to collect the metrics of the software. They chose 5 attributes of object oriented software; reuse inheritance, cohesion, coupling, and size the software. And used 8 metrics; 2 of them are traditional (fan in and LOC) and the other 5 are Chidamber and Kemerer for object oriented metrics (CBO, DIT, NMC, RFC and NOC).

Singh, et al, 2009, compared the performance of the ANN predictive model with the DT, SVM and LR predictive models. They used 12 systems in java as a dataset and object oriented metrics of Chidamber and Kemerer to find the relationship between these metrics

with the fault proneness prediction on the class level they concluded that the LOC and RFC are the best metrics to the fault proneness prediction. And NOC and DIT metrics are not useful in the fault proneness prediction. Also, they concluded that DT, SVM and ANN are better than the LR model in the performance of predicting the fault proneness.

Rathore and Kumar, 2017, proposed a recommendation system for helping the researchers to select the suitable technique of fault prediction while building the predictive model on the base of decision table concept.

## **2.3 Software Fault Prediction using Machine Learning Techniques**

Machine learning is used in several domains such as, bioinformatics, retail companies, and financial institutions. The methods of machine learning used for predicting the accuracy of the predicted model such as, genetic programming, multilayer perceptron, support vector machine, adaboost, bagging and random forest (Malhotra and Jain, 2012).

A problem that attracted researchers which considered a challenge is fault prediction. Researchers proposed techniques of fault prediction and evaluated their performance using several datasets. They used the ensemble methods through applying three base learners which are the radial basis function neural network, artificial neural Network and logistic regression analysis. Also, they studied the fault prediction model applied on 45 projects on the class level and proposed a model for cost evaluation of the quality assurance of the software. They used the metrics of the source code as input for the fault prediction model. And these metrics are considered as independent variable, while the class category is considered as dependent variable. The dataset they used is from the

repository of PROMISE that contains 45 open source projects from real life. They proposed a framework to choose the suitable source code metrics that are effective in the fault prediction model. They concluded that the most relevant metrics are LCOM (Lack of cohesion of methods), WMC (Weighted Methods per Class), CBO and RFC (Kumar, et al, 2017).

Prediction is a significance process in the software development to avert the confusion in the process of the software, to enhance quality and to minimize the time complexity. The generality of the models of fault prediction is used the dataset from previous for predicting the faults. The prediction models is useful in improving the approach of the design through classify the alternative approach to the models of faults and to improve the quality. The process of software development can be identified whether if it in the right way or not through measuring the changes that happened. The quality is the major factor for the success of the software, on the basis of the accurate work of the software for the reason that was made. The authors used 4 classifiers (Lazy K-Star, Random forest, Naive Bayes and J48) and compared between them to make the prediction with high quality and by using the dataset of NASA and WEKA tool with the measures of F-measure, recall, precision, false positive rate and true positive rate to detect the accuracy in predictions. They found that the Naive Bayes was appropriate for the dataset which is small and the random forest was appropriate for the dataset which is large (Sathyaraj and Prabu, 2015).

One technique of the machine learning techniques is called ensemble learning which is combines more than one algorithm of machine learning and trained them for producing output better than the output of anyof them separately. There are two types of

ensemble machine learning techniques, heterogeneous and homogenous ensembles. For the heterogeneous ensemble, it develops each type of the base learner in different way by using several techniques of Machine learning. By merging each prediction of the base learner together, the dataset and the prediction are created. While for the homogeneous ensemble, it uses different subsets of the whole training dataset for each base learner. To produce satisfied conditions and to reach the suitable ensemble, there are two vital and primary conditions; which are the accuracy and diversity. The authors compared between 3 of measures of ensembles; boosting, bagging and stacking and evaluated the performance of them with 11 base learners for the software defects prediction on the module level using the NASA dataset. They found that boosting improves the performance better than the bagging method in the accuracy. While for the stacking, the random forest is the best classifier for improving the software defect prediction (Akour, et al, 2017).

For selecting the software metrics subsets that help in predicting the faulty classes, (Catal, 2011) chose eight OSS (open source software) systems that have twenty internal attributes and using the feature selection technique, then compared several classifications since the classification is the most popular technique in machine learning. It is known as “supervised statistical learning”. It trains the model using the data that class defined in advance. The data is useful for the training of learning algorithm, and that causes of creating the model that used in classifying the testing instances when the class labels value is unknown. He compared several classifiers; which are known as the fault predictors with high performance. The author compared between the classifiers based on two measures; AUC (Area under receiver) and F-measure (precision and recall). The most effective predictor for the accuracy in the field of software fault prediction is the AUC (Catal, 2011).

Many methods were proposed for developing the predictive models of the software fault proneness, the statistical techniques and machine learning and others. Machine learning techniques are used to find the most appropriate metrics that predicts with the errors. One of the machine learning techniques they used is the ANN and they used historical data which is NASA dataset and applied the sensitivity analysis to determine which software metric is important in the software fault proneness. Then they used the SVM technique to support the software fault proneness and classified the module to be with errors or with no errors. After the comparative experiment they found that the effectiveness of the SVM is better than the ANN in the classification mission. The concentration of machine learning field on the research of the algorithms which improve the performance of them at the task they do by their experience. Over fitting problem is when the training data has no errors and the function doesn't generate the values correct for the data which unseen previously. While, the generalization is when the function may generate the values correct for the data which is new and not in the training data. And the hardest problem is having a good generalization function. When the function complexity increases, the training errors decrease. But when the generalization error increases, the complexity increases. The technique that reduces the over fitting problem is the cross validation. Cross validation is the set of disjoint from training data that used for the selection of a model. The useful of using machine learning is to select the most appropriate software metrics that indicates the software fault proneness by using the sensitivity analysis and to implement that model of the fault proneness prediction on the base of these metrics (Gondra, 2008).

Software classifications used metrics of complexity as input vectors and used algorithms applied on sufficient training data on the base of statistical methods and

machine learning like case-based reasoning, support vector machines, logistic regression, discriminated analysis, fuzzy classification, Bayesian models, decision table, neural networks and genetic programming to build the predictive model. Based on researches used NASA data set, Random Forest technique is better in performance of prediction comparing with SVM and MLP. Random Forest is defined by(Hong, 2012) as “ensemble classifier that manipulates its input features and uses decision table as its base classifiers”. Ensemble techniques are techniques for classification to improve the accuracy of the classification through aggregating multiple classifiers predictions. They constructed the random forest model by using the WEKA tool which is a machine learning tool and used a dataset from previous research.

Software fault proneness is an example of the software quality attributes and predicted by using software metrics. Machine learning is used to predict the fault proneness probability. They evaluated the performance of the support vector machine technique by using one dataset of NASA which is KC1 and found the relation between the object oriented metrics and the models of fault proneness. Also, they evaluated the predicted model performance by using AUC, sensitivity, precision, specificity and completeness. They found that SVM is achieving high accuracy in predicting the fault proneness of the software. Support vector machine is a tool used for classifying the data. It used in a successful way in different applications such as, text classification, face identification, identification of organisms, Chinese character classification and pattern recognition. SVM separates the dataset to 2 categories. They found the metrics that are related to fault proneness which are, SLOC, RFC and CBO. Also, they found that SVM model achieves

the feasibility, adaptability to the object oriented systems and it is useful for the fault proneness prediction for the classes (Singh, et al, 2009).

## **2.4 Software Fault Prediction Using Metrics**

The appropriate metrics for the software fault proneness prediction are process and product metrics. The datasets took from MDP repositories (NASA Metrics Data) which contain 11 datasets. To produce more useful predictive model, metrics and attributes must mined to be more useful for the domain that applied on it and to avoid the metrics or attributes that are not useful or make noise in the analysis. They used 9 techniques of data reduction, and then used Naïve bays as a data miner and classifier to build the predictive model for evaluating the methods of metrics reduction. For feature selection, they used CFS as filter method and J48 as wrapper method. And to select the subset, they used the genetic algorithm and best first. PCA and DWT are two effective methods for feature extraction used in their study (Luo, et al, 2010).

Singh, et al, 2009, compared the performance of the ANN predictive model with the DT, SVM and LR predictive models. They used 12 systems in java as a dataset and object oriented metrics of Chidamber and Kemerer to find the relationship between these metrics with the fault proneness prediction on the class level they concluded that the LOC and RFC are the best metrics to the fault proneness prediction. And NOC and DIT metrics are not useful in the fault proneness prediction. Also, they concluded that DT, SVM and ANN are better than the LR model in the performance of predicting the fault proneness.

Sureshm, et al, 2014, assessed the impact of the Chidamber and Kemerer metrics on predicting the software fault prediction for the open source systems. They used machine learning techniques for predicting faults (radial basis function network, functional link artificial neural network and artificial neural Network) and classifying faults (probabilistic neural network). The dataset used contains of 965 classes. 5 parameters used to measure performance used (statistical analysis, precision, correctness, completeness, accuracy and  $R^2$  Statistic). They used MATLAB coding to support machine learning and statistical methods for fault prediction. They concluded that WMC is the best useful from the 6 CK metrics for fault prediction.

Boucher and Badri, 2016, focused on the effect of code metrics threshold, because the models that based on it will be understood and implemented by the programmers or software experts. Also, it can provide them with the reason of why the class is fault prone. Threshold cannot be used for all projects and not all thresholds are good for fault prediction. In their study, they compared two of the thresholds algorithms and considered them in predicting faults. They used 5 different systems as datasets (Eclipse JDT Core, JEdit, KC1, Apache IVY and Apache ANT) and Bayes Network as machine learning classifier for fault prediction to give good results. Also, they assess the validity of the method of Alves Rankings and ROC in software fault prediction and found that the method of Alves Rankings gives good results and considered them as threshold techniques that do a good job.

Jureczko and Spinellis, 2010, take into account object oriented metrics to build defect prediction models. the models evaluated on five application and eleven open source projects, a model built according to the data collected from version  $i$  of a project has been



estimated by predicting the bugs in version  $i+1$ . Experimented results displayed that after applying regression models with determined class size factor they can find 80% of bugs in 10.56% to 54.93% ( $\mu=36.086$ ;  $\sigma=10.435$ ) of the classes. They found that WMC and LOC are considered as factors for the class size in the models of fault detection.

Zimmermann and Nagappan, 2008, proposed a low level graph to enhance network analysis which these graph may be used to allocate resources effectively and enable the manager to recognize central program units that are more helpful to determine bugs. They evaluated their study on Windows Server 2003, they reached that the recall for models structured from complexity metrics is less 10% points than for models structured from network measures. also, network measures recognized that 60% of the binaries that developers of windows rated as critical , more than identified by complexity metrics.

## Chapter Three

### 3. Research Methodology

#### 3.1 Overall Research Design

The proposed methodology shown in figure 1 that used in this thesis is explained in this chapter with details about all phases. Our proposed technique is fault proneness prediction model that used the Random Forest, J48, SVM, Decision Table and Naïve Bays machine algorithm tested with the object oriented metrics to get the best results.

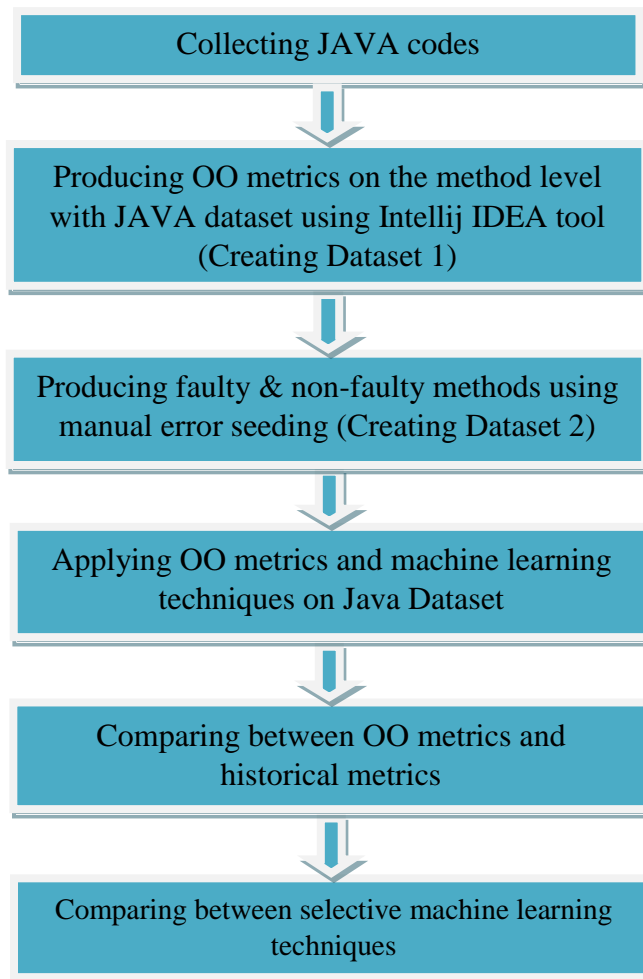
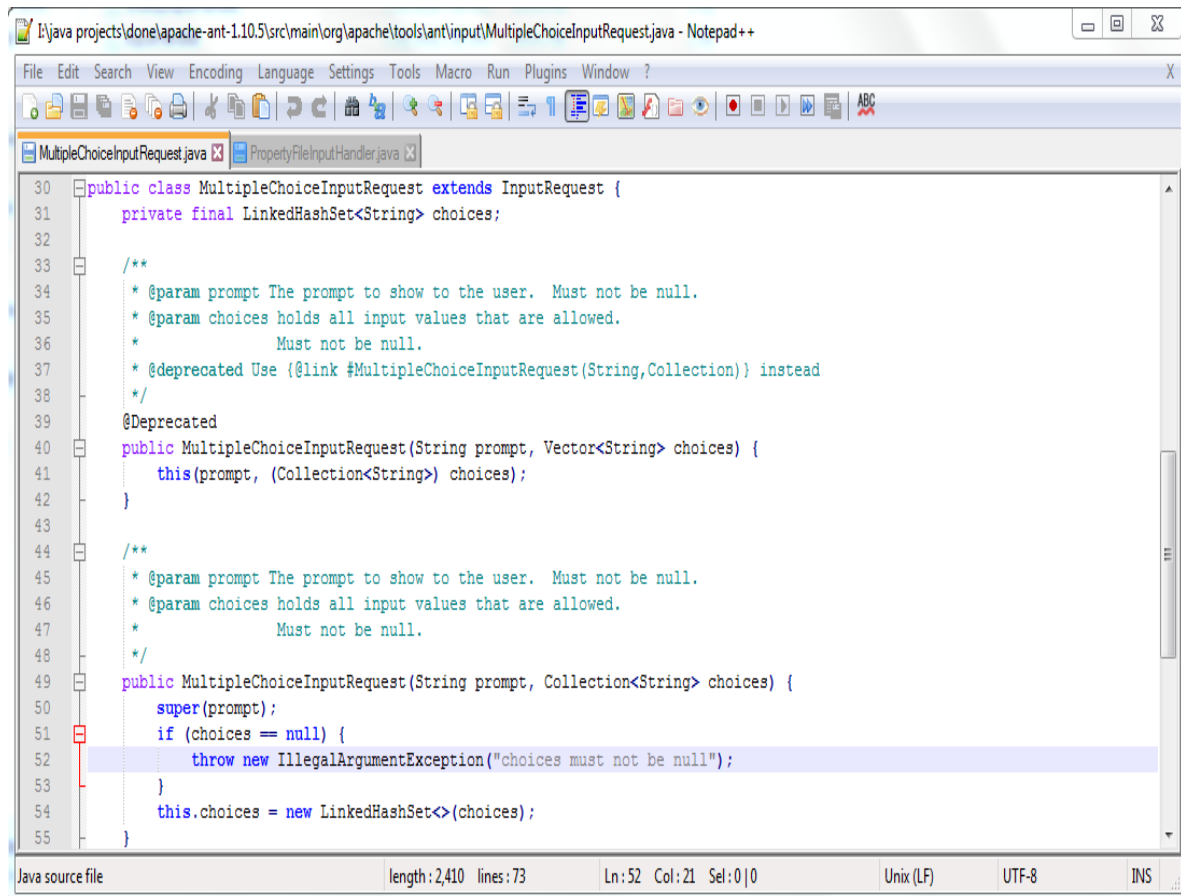


Figure 1: Research Methodology

## 3.2 Research Phases

### 3.2.1 Phase one: Collecting Java codes

Fifteen Java codes have been collected on the base of predicting faults on the method level using machine learning algorithms. Three codes of them (Ant, Cassandra and Wicket) are collected from (Hata, et al, 2011). The codes contain faults caused by manually mutation. Figure 2 shows an example of java source code.



```
30 public class MultipleChoiceInputRequest extends InputRequest {
31     private final LinkedHashSet<String> choices;
32
33     /**
34      * @param prompt The prompt to show to the user. Must not be null.
35      * @param choices holds all input values that are allowed.
36      *         Must not be null.
37      * @deprecated Use {@link #MultipleChoiceInputRequest(String,Collection)} instead
38      */
39     @Deprecated
40     public MultipleChoiceInputRequest(String prompt, Vector<String> choices) {
41         this(prompt, (Collection<String>) choices);
42     }
43
44     /**
45      * @param prompt The prompt to show to the user. Must not be null.
46      * @param choices holds all input values that are allowed.
47      *         Must not be null.
48      */
49     public MultipleChoiceInputRequest(String prompt, Collection<String> choices) {
50         super(prompt);
51         if (choices == null) {
52             throw new IllegalArgumentException("choices must not be null");
53         }
54         this.choices = new LinkedHashSet<>(choices);
55     }

```

Figure 2: Java source code

Apache Ant is “a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications” (ant.apache.org, 2018).

Apache Cassandra, “a top level Apache project born at Facebook and built on Amazon’s Dynamo and Google’s BigTable, is a distributed database for managing large amounts of structured data across many commodity servers, while providing highly available service and no single point of failure. Apache Cassandra offers capabilities that relational databases and other NoSQL databases simply cannot match such as: continuous availability, linear scale performance, operational simplicity and easy data distribution across multiple data centers and cloud availability zones” (cassandra.apache.org, 2016).

Wicket is “an open source Java component oriented web application framework that powers thousands of web applications and web sites for governments, stores, universities, cities, banks, email providers, and more. Wicket is one of the few survivors of the Java server side web framework wars of the mid 2000's. Wicket is an open source, component oriented, server side, Java web application framework” (wicket.apache.org, 2018).

Apache Log4j is “a Java-based logging utility. It was originally written by CekiGülcü and is part of the Apache Logging Services project of the Apache Software Foundation. Log4j is one of several Java logging frameworks” (logging.apache.org/log4j/2.x, 2018).

The goal of MARC4J is “to provide an easy to use Application Programming Interface (API) for working with MARC and MARCXML in Java. MARC stands for Machine Readable Cataloging and is a widely used exchange format for bibliographic data. MARCXML provides a loss-less conversion between MARC (MARC21 but also other formats like UNIMARC) and XML” ([github.com/marc4j/marc4j](https://github.com/marc4j/marc4j), 2018).

JTopas is “Java tokenizer and parser tools. The JTopas project provides a small, easy-to-use Java library for the common problem of parsing arbitrary text data. These data can come from a simple configuration file with a few comments, a HTML, XML or RTF stream, source code of various programming languages etc. Sometimes a text has to be parsed completely, sometimes only parts of it are important” ([openhub.net/p/jtopas](https://openhub.net/p/jtopas), 2018).

PureMVC is “a lightweight framework for creating applications based upon the classic Model-View-Controller design meta-pattern. This is a Java port of the AS3 reference implementation of the MultiCore Version. It supports modular programming through the use of Multiton Core actors instead of the Singletons used in the Standard Version” ([puremvc.org](http://puremvc.org), 2016).

JPacman is “like game used for teaching software testing. It exposes students to the use of git, maven, JUnit, and mockito. Parts of the code are well tested, whereas others are left untested intentionally. As a student in software testing, you can extend the test suite, or use the framework to build extensions in a test-driven way. As a teacher, you can use the framework to create your own testing exercises” ([github.com/SERG-Delft/jpacman-framework](https://github.com/SERG-Delft/jpacman-framework), 2018).

Commons-Lang is the “standard Java libraries fail to provide enough methods for manipulation of its core classes. The Commons Lang Component provides these extra methods. The Commons Lang Component provides a host of helper utilities for the java. Lang API, notably String manipulation methods, basic numerical methods, object reflection, creation and serialization, and System properties. Additionally it contains an inheritable enum type, an exception structure that supports multiple types of nested-Exceptions and a series of utilities dedicated to help with building methods, such as hash Code, to String and equals. With version of commons-lang 3.x, developers decided to change API and therefore created differently named artifact and jar files. This is the new version, while apache-commons-Lang is the compatibility package” (github.com/apache/commons-lang, 2018).

Apache Commons is “an Apache project focused on all aspects of reusable Java components. The Apache Commons source code repositories are writable for all ASF committers. While Apache Commons is a Commit-Then-Review community, we would consider it polite and helpful for contributors to announce their intentions and plans on the dev mailing list before committing code. All contributors should read our contributing guidelines” (commons.apache.org, 2018).

### 3.2.2 Phase Two: Producing OO metrics on the method level with JAVA dataset using IntelliJ IDEA tool (Creating Dataset 1)

To predict the software fault, several metrics such as historical and object oriented metrics used in different researches. The dataset used in researches may be private or public. Everyone can use the public dataset for a lot of applications. Unlike the private dataset that not everyone can access and in order to use these datasets, some procedures and issues should be followed. In this thesis, the datasets is created through collecting JAVA codes (large and small scales), and then import these codes into IntelliJ IDEA tool to extract the desired metrics (features) as shown in table 1 before.

Figure 3 below shows the extracted results from IntelliJ IDEA tool that contains methods as rows and each method has 44 metrics (features).

	A	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	Method	BRANCH	CALL	CALLED	CALLEDp	CALLEDT	CAST	CAUGHT	CDENS	CLOC	COM_RAT	CONTROL	D	E	ev(G)	EXEC	EXP	IF_NEST	IMP	iv(G)	
2	org.apache	0	0	1	0	0	0	0	0	6	0.6667	0	2	39	1	1	5	0	0		
3	org.apache	0	0	3	1	0	1	0	0	6	0.5455	0	4	282	1	2	23	0	0		
4	org.apache	0	0	1	0	0	0	0	0	6	0.6667	0	2	39	1	1	5	0	0		
5	org.apache	0	0	3	1	0	1	0	0	6	0.5455	0	4	282	1	2	23	0	0		
6	org.apache	0	0	1	1	1	0	0	0	3	0.5	0	1	4	1	1	3	0	0		
7	org.apache	0	0	3	0	0	0	0	0	9	0.6429	0	2	75	1	3	9	0	0		
8	org.apache	0	0	4	11	5	6	0	0.2	16	0.64	1	6	445	1	4	15	1	0		
9	org.apache	0	0	1	0	0	0	0	0	13	0.7222	0	2	91	1	3	10	0	0		
10	org.apache	0	0	3	1	1	0	0	0	11	0.6875	0	2	67	1	3	9	0	0		
11	org.apache	0	0	1	0	0	0	0	0	12	0.75	0	2	87	1	1	6	0	0		
12	org.apache	0	0	1	1	1	0	0	0	6	0.5455	0	1	33	1	3	10	0	0		
13	org.apache	0	0	3	2	2	0	0	1	0.2857	8	0.4	2	11	1278	1	5	29	0		
14	org.apache	0	0	0	0	0	0	0	0	7	0.7	0	2	49	1	1	5	0	0		
15	org.apache	0	0	2	0	0	0	0	0.2	5	0.3846	1	6	279	2	4	9	1	0		
16	org.apache	0	0	2	6	6	0	0	0	4	0.5714	0	0	0	1	1	7	0	0		
17	org.apache	0	0	2	3	3	0	0	0	9	0.75	0	3	90	1	1	13	0	0		
18	org.apache	0	0	2	2	2	0	0	0.3333	7	0.5385	1	5	126	2	2	8	1	0		
19	org.apache	0	0	4	2	2	0	0	1	0.25	9	0.5	1	7	646	1	3	15	0		
20	org.apache	0	1	27	2	2	0	0	0.3929	10	0.1852	11	18	8813	7	17	113	2	0		
21	org.apache	0	0	2	5	5	0	0	0	9	0.75	0	3	90	1	1	13	0	0		
22	org.apache	0	0	8	3	3	0	0	0	0	0	0	1	17	1	1	25	0	0		
23	org.apache	0	0	1	5	5	0	0	0	6	0.6667	0	0	0	1	1	2	0	0		
24	org.apache	0	0	0	3	3	0	0	0	6	0.75	0	0	0	1	0	0	0	0		
25	org.apache	0	0	4	21	18	3	0	0.3333	4	0.2857	2	10	685	1	4	20	1	0		

Figure 3: IntelliJ IDEA tool results (metrics)

### 3.2.3 Phase Three: Producing faulty & non-faulty methods using manual error seeding (Creating Dataset 2)

The class label as shown in figure 6 was created by using the manual error seeding as shown in figure 2 and 3 below. Through changing in the source code of the methods, that will cause faults. Then the method in the dataset will be faulty. And the rest will be not faulty. After adding faults in the source code of the method, prediction of real faults on the base of seeded faults is done.

```
public MultipleChoiceInputRequest(String prompt, Collection<String> choices) {
    super(prompt);
    if (choices == null) {
        throw new IllegalArgumentException("choices must not be null");
    }
    this.choices = new LinkedHashSet<>(choices);
}
```

Figure 4: Method before error seeding

```
public MultipleChoiceInputRequest(String prompt, Collection<String> choices) {
    super(prompt);
    if (choices != null) {
        throw new IllegalArgumentException("choices must not be null");
    }
    this.choices = new LinkedHashSet<>(choices);
}
```

Figure 5: Method after error seeding



The screenshot shows a Microsoft Excel spreadsheet titled 'apache-ant2 - Microsoft Excel'. The spreadsheet contains a dataset with 28 rows and 29 columns. The columns are labeled as follows: A (Method), B (CALL), C (CLOC), D (COM\_RAT), E (D), F (E), G (EXEC), H (EXP), I (iv(G)), J (LOC), K (N), L (n), M (NCLOC), N (NP), O (STAT), P (TCOM\_RAT), Q (STAT), R (QCP\_CRCT), S (QCP\_MAINT), T (QCP\_RLBT), U (STAT), V (TCOM\_RAT), W (V), X (V(G)), Y (class label). The 'class label' column contains the value 'TRUE' for all rows.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	X	Y	
Method	CALL	CLOC	COM_RAT	D	E	EXEC	EXP	iv(G)	LOC	N	n	NCLOC	NP	STAT	TCOM_RAT	STAT	QCP_CRCT	QCP_MAINT	QCP_RLBT	STAT	TCOM_RAT	V	V(G)	class label
org.apache	0	1	6	1	2	39	1	5	1	9	7	7	3	2	5	24	11	1	2	19	1	1	TRUE	
org.apache	0	3	6	1	4	282	2	23	1	11	18	14	5	3	10	60	26	2	1	68	2	1	TRUE	
org.apache	0	1	6	1	2	39	1	5	1	9	7	7	3	2	5	24	11	1	2	19	1	1	TRUE	
org.apache	0	3	6	1	4	282	2	23	1	11	18	14	5	3	10	60	26	2	1	68	2	1	TRUE	
org.apache	0	1	3	1	1	4	1	3	1	6	3	3	3	0	4	12	7	1	1	4	1	1	TRUE	
org.apache	0	3	9	1	2	75	3	9	1	14	10	8	5	3	7	35	16	3	2	30	1	1	TRUE	
org.apache	0	4	16	1	6	445	4	15	2	25	19	15	9	4	15	67	32	5	2	74	2	1	TRUE	
org.apache	0	1	13	1	2	91	3	10	1	18	11	10	5	2	7	38	17	3	3	36	1	1	TRUE	
org.apache	0	3	11	1	2	67	3	9	1	16	9	8	5	2	7	32	15	3	2	26	1	1	TRUE	
org.apache	0	1	12	1	2	87	1	6	1	16	11	9	4	3	5	36	15	1	3	34	1	1	TRUE	
org.apache	0	1	6	1	1	33	3	10	1	11	8	7	5	1	6	29	14	3	1	22	1	1	TRUE	
org.apache	0	3	8	0	11	1278	5	29	3	20	27	19	12	0	22	94	44	7	1	114	4	1	TRUE	
org.apache	0	0	7	1	2	49	1	5	1	10	7	7	3	0	5	24	11	1	2	19	1	1	TRUE	
org.apache	0	2	5	0	6	279	4	9	1	13	13	12	8	0	15	49	26	5	1	46	2	1	TRUE	
org.apache	0	2	4	1	0	0	1	7	1	7	4	4	3	0	3	15	8	1	1	8	1	1	TRUE	
org.apache	0	2	9	1	3	90	1	13	1	12	10	8	3	1	8	35	17	1	3	30	2	1	TRUE	
org.apache	0	2	7	1	5	126	2	8	1	13	9	7	6	1	12	35	20	3	1	25	2	1	TRUE	
org.apache	0	4	9	1	7	646	3	15	2	18	21	16	9	1	15	72	33	4	1	84	3	1	TRUE	
org.apache	0	27	10	0	18	8813	17	113	8	54	88	40	44	1	66	316	153	28	0	468	11	1	TRUE	
org.apache	0	2	9	1	3	90	1	13	1	12	10	8	3	1	8	35	17	1	3	30	2	1	TRUE	
org.apache	0	8	0	0	1	17	1	25	1	4	5	5	4	2	4	18	9	1	0	11	1	1	TRUE	
org.apache	0	1	6	1	0	0	1	2	1	9	4	4	3	1	3	15	8	1	2	8	1	1	TRUE	
org.apache	0	0	6	1	0	0	0	0	1	8	3	3	2	1	2	11	6	0	3	4	1	1	TRUE	
org.apache	0	4	4	0	10	685	4	20	3	14	18	14	10	0	22	67	35	6	0	68	3	1	TRUE	
org.apache	0	1	1	0	0	0	1	2	1	4	3	3	3	0	3	12	7	1	0	4	1	1	TRUE	
org.apache	0	10	13	0	9	1498	6	36	1	28	35	22	15	3	17	113	44	6	1	156	1	1	TRUE	
org.apache	0	6	12	0	16	3259	7	38	2	28	44	22	16	2	34	151	68	10	1	196	4	1	TRUE	

Figure 6: Class label

### 3.2.4 Phase Four: Applying OO metrics and machine learning techniques on Java Dataset

In this step, the object oriented metrics (B, CALL, CLOC, COM\_RAT, D, E, EXEC, EXP, IV(G), LOC, N, n, NCLOC, NP, STAT, TCOM\_RAT, , STAT, QCP\_CRCT, QCP\_MAINT, QCP\_RLBT, V, V(G)) are extracted from the java codes after the error seeding to get new values of features as shown in figure 7 below.

	A	C	E	L	M	O	P	R	S	V	X	AA	AB	AD	AE	AI	AJ	AK	AN	AO	AP	AQ	AR
1	Method	B	CALL	CLOC	COM_RAT	D	E	EXEC	EXP	iv(G)	LOC	N	n	NCLOC	NP	OCP_CRCT	OCP_MAIN	OCP_RLBT	STAT	TCOM_RA	V	v(G)	class label
2	org.apache	0	1	6	1	2	39	1	5	1	9	7	7	3	2	5	24	11	1	2	19	1	TRUE
3	org.apache	0	3	6	1	4	282	2	23	1	11	18	14	5	3	10	60	26	2	1	68	2	TRUE
4	org.apache	0	1	6	1	2	39	1	5	1	9	7	7	3	2	5	24	11	1	2	19	1	TRUE
5	org.apache	0	3	6	1	4	282	2	23	1	11	18	14	5	3	10	60	26	2	1	68	2	TRUE
6	org.apache	0	1	3	1	1	4	1	3	1	6	3	3	3	0	4	12	7	1	1	4	1	TRUE
7	org.apache	0	3	9	1	2	75	3	9	1	14	10	8	5	3	7	35	16	3	2	30	1	TRUE
8	org.apache	0	4	16	1	6	445	4	15	2	25	19	15	9	4	15	67	32	5	2	74	2	TRUE
9	org.apache	0	1	13	1	2	91	3	10	1	18	11	10	5	2	7	38	17	3	3	36	1	TRUE
10	org.apache	0	3	11	1	2	67	3	9	1	16	9	8	5	2	7	32	15	3	2	26	1	TRUE
11	org.apache	0	1	12	1	2	87	1	6	1	16	11	9	4	3	5	36	15	1	3	34	1	TRUE
12	org.apache	0	1	6	1	1	33	3	10	1	11	8	7	5	1	6	29	14	3	1	22	1	TRUE
13	org.apache	0	3	8	0	11	1278	5	29	3	20	27	19	12	0	22	94	44	7	1	114	4	TRUE
14	org.apache	0	0	7	1	2	49	1	5	1	10	7	7	3	0	5	24	11	1	2	19	1	TRUE
15	org.apache	0	2	5	0	6	279	4	9	1	13	13	12	8	0	15	49	26	5	1	46	2	TRUE
16	org.apache	0	2	4	1	0	0	1	7	1	7	4	4	3	0	3	15	8	1	1	8	1	TRUE
17	org.apache	0	2	9	1	3	90	1	13	1	12	10	8	3	1	8	35	17	1	3	30	2	TRUE
18	org.apache	0	2	7	1	5	126	2	8	1	13	9	7	6	1	12	35	20	3	1	25	2	TRUE
19	org.apache	0	4	9	1	7	646	3	15	2	18	21	16	9	1	15	72	33	4	1	84	3	TRUE
20	org.apache	0	27	10	0	18	8813	17	113	8	54	88	40	44	1	66	316	153	28	0	468	11	TRUE
21	org.apache	0	2	9	1	3	90	1	13	1	12	10	8	3	1	8	35	17	1	3	30	2	TRUE
22	org.apache	0	8	0	0	1	17	1	25	1	4	5	5	4	2	4	18	9	1	0	11	1	TRUE
23	org.apache	0	1	6	1	0	0	1	2	1	9	4	4	3	1	3	15	8	1	2	8	1	TRUE
24	org.apache	0	0	6	1	0	0	0	0	1	8	3	3	2	1	2	11	6	0	3	4	1	TRUE
25	org.apache	0	4	4	0	10	685	4	20	3	14	18	14	10	0	22	67	35	6	0	68	3	TRUE
26	org.apache	0	1	1	0	0	0	1	2	1	4	3	3	3	0	3	12	7	1	0	4	1	TRUE
27	org.apache	0	10	13	0	9	1498	6	36	1	28	35	22	15	3	17	113	44	6	1	156	1	TRUE
28	org.apache	0	6	12	0	16	3259	7	38	2	28	44	22	16	2	34	151	68	10	1	196	4	TRUE

Figure 7: metrics after error seeding

Then machine learning techniques are applied (Decision Table, J48, SVM, Naïve Bays and Random forest) on the dataset to build the predictive model on method level using python.

### 3.2.5 Phase Five: Comparing between OO metrics and historical metrics

In order to produce a better performance in predicting software fault proneness on the method level, the object oriented metrics that shown in table 1 before and historical metrics that shown in table 2 below, are being compared and decided which is the best set of metrics that improve the software quality and reduce the cost and time in determining the methods that contain faults.

Table 2: Historical Metrics Description (Hata, et al, 2012)

#	Metrics	Description
1	Code-Related Metrics	Churned LOC / Total LOC, and Deleted LOC / Total LOC
2	Process-Related Metrics	Changes, fixes, past bugs, Process complexity metrics
3	Organizational Metrics	Number of developers, Structure of organization, Network metrics
4	Geographical Metrics	locations

### 3.2.6 Phase Six: Comparing between machine learning techniques that used on the method level

As an enhancement step in the software quality, several machine learning techniques Decision Table, SVM, Naïve Bays, J48 and Random forest are compared to determine which is the best machine learning techniques using the object oriented metrics on method level.

#### 3.2.6.1 Naïve Bays Algorithm:

The Naïve Bayes classifier, presently experiencing a renaissance in machine learning, has long been a core technique in info retrieval. Number of the variations of Naïve Bayes models used for text retrieval and classification, specializing in the spacing assumptions created concerning word occurrences in documents (Lewis, D. D, 1998).

Naïve Bays is commonly used as a baseline in text classification as a result of its quick and straight forward to implement. Its serve assumptions create such potency potential, however additionally adversely has an effect on the standard of its results. With

Naïve Bays classifiers lead to a fast algorithmic program that's competitive with state of the-art text classification algorithms such as the Support Vector Machine.

The Naïve Bayes model could be a heavily simplified Bayesian likelihood model. The Naïve Bayes classifier operates on a powerful independence assumption; this implies that the likelihood of 1 attribute doesn't have an effect on the likelihood of the opposite. Given a series of attributes, the naïve Bayes classifier makes  $2^n$ ! Freeland assumptions. However, the results of the naïve Bayes classifier are often correct.

### **3.2.6.2 J48 Algorithm:**

J48 is an expansion of ID3. The further features of J48 are show cause for missing values, attribute value ranges, decision trees pruning and derivation of rules. J48 is an open source Java implementation algorithm; it generates based on particular identity of data and it is objective is gradually generalized of a decision tree till it gains balance of accuracy and flexibility (Kaur, G., et al. 2014)

J48 algorithm creates a decision tree based on the set of training instances. It depend on greedy-top-down approach to the build the decision tree; it starts with building a root node, where the attribute is considered as the best classifies all the training instances is the same process is reiterated for the rest of the attributes recursively till all the instances have been classified. In order to select the best instances, the data gained from each instances is calculated and the highest gained data is selected (Saravanan, N., et al. 2018).

All decision trees are most powerful technique in data processing implementation .A decision tree offers several benefits to data processing; it provides a straightforward understanding for the implementation It also proceed with flawed datasets or missing values and provides an improved prediction.J48 is capable of handling each Nominal and numeric data (Onik, A. et al. 2015).

J48 scans for a surveillance list in an incremental technique. It finds one run at any moment. Each time it finds a decision it adds it to finish the rundown standards, unhand the cases secured by that administer from the preparation in order to find another lead for the rest of the preparation cases. Classification of dengue dataset using J48 algorithm and ant colony based AJ48 algorithm (Saravanan, N. et al, 2017).

J48 based on the concept of information entropy and inspect the difference in entropy; this variation in entropy is called as normalized information. Attribute with highest normalized information is used to make decisions. J48 works very well with both discrete attributes and continuous attributes, also it gives an option for refining trees after creation (Bhargava, N. et al, 2017)

### **3.2.6.3 Decision Table Algorithm:**

The decision Table classifier (DTC) is one in all the doable approaches to multistage decision making. The main idea of any multistage approach is to split up a complex decision into unique several decisions, to get a final best solution obtained (Safavian, S. et al, 1991).

The ability of Decision tables is evaluated as a hypothesis for supervised learning algorithms. Decision tables are one among the only hypothesis areas attainable, and frequently they are straightforward to know. Decision tables show that on artificial and real-world domains containing solely separate options, and a lot of datasets employed in machine learning either don't need these options, or that these options have few values (Kohavi, R. 1995).

The advantages of decision table include robustness based on simultaneous usage of complementary recognition approaches and easy in dynamic adaptation. Decision tables are represented as ranking of a given class. They can be integrated by a many methods that reduce or the class set. These methods are acceptable regardless of the similarity between the individual classifiers; the effectivity and efficiency of the methods has been shown in many applications with real-world data. It is predicted that the decision tables are applicable to many problem domains.

Each decision in the decision table is corresponded to a, relation, variable or predicate whose probability values are within an alternatives. Each action in the decision table is a procedure to be performed; one of the uses of decision tables is to detect conditions under a certain input factor (Ho, T. et al, 1994).

One of the important approaches for decision-making and pattern recognition is a decision table, which is based on specific attribute selection. Attribute selection is a process of selecting the best subset of features by evaluating the performance of learning schemes depending on different attribute subsets.

Decision tables are significantly supreme to other models in terms of reliability, accuracy and response time. Decision tables have not been used in many fields and its results improved its high performance in classification (Chen, C., et al, 2016).

#### **3.2.6.4 Random Forest Algorithm:**

Random Forest (RF) could be a powerful machine learning classifier that's comparatively unknown in land remote sensing and has not been evaluated completely by the remote sensing community compared to a lot of typical pattern recognition techniques. Key benefits of RF include: their non-parametric nature; high classification accuracy; and capability to see variable importance. However, the split rules for classification are unknown, thus RF is thought of to be recording machine kind classifier. RF provides Associate in Nursing algorithmic rule for estimating missing values; and suppleness to perform many sorts of information analysis, as well as regression, classification, survival analysis, and unsupervised learning (Rodriguez-Galiano, V. et al, 2012).

Random Forests (RF), does not need reduction of the predictor before classification. To boot, RF yield variable importance measures for every candidate predictor. The effectiveness of RF variable is its importance measures in characteristic verity predictor among an oversized range of candidate predictors (Archer, K. et al, 2008). A Random Forest (RF) classifier is an associate ensemble classifier that produces multiple decision trees, employing a willy-nilly elite set of coaching samples and variables. This classifier has become in style inside the remote sensing community because

of the accuracy of its classifications. RF classifier handles high knowledge spatiality and multi co linearity, being quick and insensitive to over fitting. It is, however, sensitive to the sampling style. RF classifier has been extensively exploited in numerous situations, as an example to cut back the amount of dimensions of hyper spectral knowledge (Belgiu, M et al, 2016).

The random forest (RF) formula by Leo Breiman has become a customary information analysis tool in bioinformatics. It has shown glorious performance in settings wherever the quantity of variables is far larger than the quantity of observations, RF development on applications of bioinformatics and machine biology. Special attention is paid to sensible aspects like the choice of parameters, offered RF implementations, and vital pitfalls and biases of RF and its variable importance measures (Boulesteix, A. et al, 2012).

### **3.2.6.5 SVM Algorithm:**

Support vector machine is one in all the foremost powerful learning algorithms and is employed for a good range of real-world applications. The potency of SVM formula and its performance principally depends on the kernel kind and its parameters. Moreover, the feature set choice that's accustomed train the SVM model is another necessary issue that encompasses a major influence on the classification accuracy. The feature set choice could be an important step in machine learning, especially for managing high dimensional dataset. Most of the previous researches handled these necessary factors individually (Aljarah, I., et al, 2018).



Support vector machine (SVM) is thought as a robust methodology for resolution issues in nonlinear classification, perform estimation and density estimation. SVM has been introduced at intervals the context of applied mathematics learning theory and structural risk minimization. Least squares support vector machine (LS-SVM) is reformulations from normal SVM that cause resolution linear Karush-Kuhn-Tucker (KKT) systems. LS-SVM is closely associated with regularization networks and Gaussian processes to emphasize and exploits primal-dual interpretations (Mustafa, M. et al, 2012).

SVM is a theoretical machine learning classification technique that was adopted for structural risk minimization, authors show an empirical analysis that use SVM on the dataset with sound performance assessments. Therefore, authors have a tendency to utilize SVM for the benchmark classification rule to notice the accuracy rate of the feature subsets. SVM was 1st conferred at the Fifth Annual ACM Workshop on Computation Learning Theory (COLT). SVM preprocessing data patterns at a usually a lot of higher level than the initial feature subset. With associate acceptable non-linear mapping to the high-dimensional subset (Zhang, Y., et al, 2018).

Support Vector Machines is one of the techniques that are used for pattern classification and it is widely used in many application areas, kernel parameters is a major factor that impacts accuracy classification. The objective of this research is to optimize the best parameters and feature subset without degrading the SVM (Huang, C. et al, 2016).

### 3.3 Datasets

The datasets that used in this thesis is JAVA Open Source Projects as shown in table 3, (Malhotra and Jain, 2012, Hata, et al, 2012, Koru and Liu, 2005). The dataset consists of fourteen java projects; three of them were large scale and the remaining eleven were small scale. The overall extracted features were 44 features, but the number of features after processing was 21 features as shown in table 1. We exclude the features that make no difference on the results and have low variation that shown in table 6.

**Table 3: Details of Datasets**

Dataset	# of instances	# of Features	#of Features after Processing	Scale
Ant	14133	44	21	Large
Cassandra	15319	44	21	Large
Wicket	10310	44	21	Large
Apa	309	44	21	Small
apache-log4	4480	44	21	Large
cinema	326	44	21	Small
commons-codec	1321	44	21	Large
common-lang	5511	44	21	Large
iyad-marc4j	504	44	21	Small
jpacman	218	44	21	Small
jtopas1	433	44	21	Small
jtopas2	498	44	21	Small
puremvc	212	44	21	Small
realstate	483	44	21	Small

All projects of dataset are written in Java and have relatively object oriented properties and faults. The projects were chosen because they span varied application domains. Also, the open source projects are available for everyone in case of discovering anything that needs to change.

The datasets was normalized through rescaling attributes to the range -2 to 2 as shown in figure 4 (Singhal, S., & Jena, M. 2013). And preprocessed to gain better results by excluding the features in table 4 that have no effect on the dataset or have low variation as shown in figure 5 on the base of information gain.

Method	B	CALL	CLOC	COM_RAT	D	E	EXEC	EXP	iv(G)	LOC	N	n	AB	AD	AE	AN	AO	AR	AS	AT
'org.apache	-2	-1.98876	-1.84615	-1.75	-1.96875	-1.99971	-1.9785	-1.98208	-1.96552	-1.91262	-1.97422	-1.94595	-1.97619	-1.63636	-1.9845	-1.81818	-1.9885	-1.9697	TRUE	
'org.apache	-2	-1.96629	-1.84615	-1.75	-1.9375	-1.99787	-1.95699	-1.91756	-1.96552	-1.8932	-1.9337	-1.89189	-1.95238	-1.45455	-1.96899	-1.90909	-1.95885	-1.93939	TRUE	
'org.apache	-2	-1.98876	-1.84615	-1.75	-1.96875	-1.99971	-1.9785	-1.98208	-1.96552	-1.91262	-1.97422	-1.94595	-1.97619	-1.63636	-1.9845	-1.81818	-1.9885	-1.9697	TRUE	
'org.apache	-2	-1.96629	-1.84615	-1.75	-1.9375	-1.99787	-1.95699	-1.91756	-1.96552	-1.8932	-1.9337	-1.89189	-1.95238	-1.45455	-1.96899	-1.90909	-1.95885	-1.93939	TRUE	
'org.apache	-2	-1.98876	-1.92308	-1.75	-1.98438	-1.99997	-1.9785	-1.98925	-1.96552	-1.94175	-1.98895	-1.97683	-1.97619	-2	-1.9845	-1.90909	-1.99758	-1.9697	TRUE	
'org.apache	-2	-1.96629	-1.76923	-1.75	-1.96875	-1.99943	-1.93548	-1.96774	-1.96552	-1.86408	-1.96317	-1.93822	-1.95238	-1.45455	-1.95349	-1.81818	-1.98185	-1.9697	TRUE	
'org.apache	-2	-1.95506	-1.58974	-1.75	-1.90625	-1.99663	-1.91398	-1.94624	-1.93103	-1.75728	-1.93002	-1.88417	-1.90476	-1.27273	-1.92248	-1.81818	-1.95522	-1.93939	TRUE	
'org.apache	-2	-1.98876	-1.66667	-1.75	-1.96875	-1.99931	-1.93548	-1.96416	-1.96552	-1.82524	-1.95948	-1.92278	-1.95238	-1.63636	-1.95349	-1.72727	-1.97822	-1.9697	TRUE	
'org.apache	-2	-1.96629	-1.71795	-1.75	-1.96875	-1.99949	-1.93548	-1.96774	-1.96552	-1.84466	-1.96685	-1.93822	-1.95238	-1.63636	-1.95349	-1.81818	-1.98427	-1.9697	TRUE	
'org.apache	-2	-1.98876	-1.69231	-1.75	-1.96875	-1.99934	-1.9785	-1.9785	-1.96552	-1.84466	-1.95948	-1.9305	-1.96429	-1.45455	-1.9845	-1.72727	-1.97943	-1.9697	TRUE	
'org.apache	-2	-1.98876	-1.84615	-1.75	-1.98438	-1.99975	-1.93548	-1.96416	-1.96552	-1.8932	-1.97053	-1.94595	-1.95238	-1.81818	-1.95349	-1.90909	-1.98669	-1.9697	TRUE	
'org.apache	-2	-1.96629	-1.79487	-2	-1.82813	-1.99033	-1.89247	-1.89606	-1.89655	-1.80583	-1.90055	-1.85328	-1.86905	-2	-1.89147	-1.90909	-1.93101	-1.87879	TRUE	
'org.apache	-2	-2	-1.82051	-1.75	-1.96875	-1.99963	-1.9785	-1.98208	-1.96552	-1.90291	-1.97422	-1.94595	-1.97619	-2	-1.9845	-1.81818	-1.9885	-1.9697	TRUE	
'org.apache	-2	-1.97753	-1.8718	-2	-1.90625	-1.99789	-1.91398	-1.96774	-1.96552	-1.87379	-1.95212	-1.90734	-1.91667	-2	-1.92248	-1.90909	-1.97216	-1.93939	TRUE	
'org.apache	-2	-1.97753	-1.89744	-1.75	-2	-2	-1.9785	-1.97491	-1.96552	-1.93204	-1.98527	-1.96911	-1.97619	-2	-1.9845	-1.90909	-1.99516	-1.9697	TRUE	
'org.apache	-2	-1.97753	-1.76923	-1.75	-1.95313	-1.99932	-1.9785	-1.95341	-1.96552	-1.8835	-1.96317	-1.93822	-1.97619	-1.81818	-1.9845	-1.72727	-1.98185	-1.93939	TRUE	
'org.apache	-2	-1.97753	-1.82051	-1.75	-1.92188	-1.99905	-1.95699	-1.97133	-1.96552	-1.87379	-1.96685	-1.94595	-1.94048	-1.81818	-1.95349	-1.90909	-1.98487	-1.93939	TRUE	
'org.apache	-2	-1.95506	-1.76923	-1.75	-1.89063	-1.99511	-1.93548	-1.94624	-1.93103	-1.82524	-1.92265	-1.87645	-1.90476	-1.81818	-1.93798	-1.90909	-1.94917	-1.90909	TRUE	
'org.apache	-2	-1.69663	-1.74359	-2	-1.71875	-1.9333	-1.63441	-1.59498	-1.72414	-1.47573	-1.67588	-1.69112	-1.4881	-1.81818	-1.56589	-2	-1.71679	-1.66667	TRUE	
'org.apache	-2	-1.97753	-1.76923	-1.75	-1.95313	-1.99932	-1.9785	-1.95341	-1.96552	-1.8835	-1.96317	-1.93822	-1.97619	-1.81818	-1.9845	-1.72727	-1.98185	-1.93939	TRUE	
'org.apache	-2	-1.91011	-2	-2	-1.98438	-1.99987	-1.9785	-1.91039	-1.96552	-1.96117	-1.98158	-1.96139	-1.96429	-1.63636	-1.9845	-2	-1.99334	-1.9697	TRUE	
'org.apache	-2	-1.98876	-1.84615	-1.75	-2	-2	-1.9785	-1.99283	-1.96552	-1.91262	-1.98527	-1.96911	-1.97619	-1.81818	-1.9845	-1.81818	-1.99516	-1.9697	TRUE	
'org.apache	-2	-2	-1.84615	-1.75	-2	-2	-2	-2	-1.96552	-1.92233	-1.98895	-1.97683	-1.9881	-1.81818	-2	-1.72727	-1.99758	-1.9697	TRUE	
'org.apache	-2	-1.95506	-1.89744	-2	-1.84375	-1.99482	-1.91398	-1.92832	-1.89655	-1.86408	-1.9337	-1.89189	-1.89286	-2	-1.90698	-2	-1.95885	-1.90909	TRUE	

Figure 4: Normalized Dataset

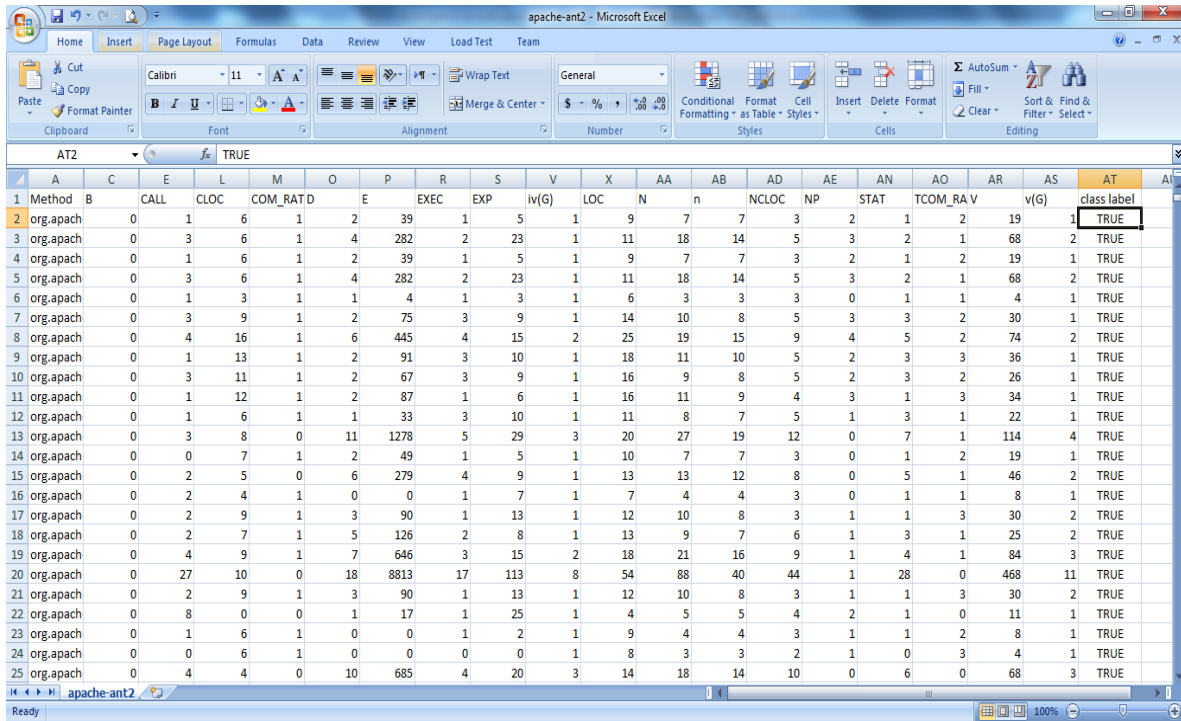


Figure 5: Processed Dataset

Table 4: Not used Object Oriented metrics (jetbrains.com/idea, 2018)

#	Metrics	Description
1	IF_NEST	Calculates the maximum depth of nesting of conditional (if) statements in each method.
2	CDENS	Calculates the ratio of control statements to all statements for each method.
3	ev(G)	Calculates the essential complexity of each non-abstract method. Essential complexity is a graph-theoretic measure of just how ill-structured a method's control flow is. Essential complexity ranges from 1 to v(G), the cyclomatic complexity of the method.
4	JLOC	Calculates the number of lines of javadoc comments in each method. Whitespace is not counted for purposes of this metric.
5	LOOP_NEST	Calculates the maximum depth of nesting of loop statements in each method. For, while, and do-while loops are counted.
6	NEST	Calculates the maximum nesting depth of each method.
7	TODO	Calculates the number of TODO comments in each method. The format of TODO comments is defined in the Settings   Editor   TODO configuration panel.
8	ASSERT	Calculates the total number of assert statements in each method.
9	BRANCH	Calculates the total number of non-structured branch statements in each method. Non-structured branch statements include continue statements and branch statements outside of switch statements.

10	CONTROL	Calculates the total number of control statements in each method. Control statements include if, for, while, do, try, break, continue, switch, and synchronized statements.
11	CAUGHT	Calculates the number of exception classes which are caught in each method.
12	THROWS	Calculates the number of exception classes each method declares in its "throws" clause.
13	IMP	Calculates the number of concrete implementation of each abstract method.
14	LOOP	Calculates the total number of loop statements in each method. For, while, and do-while loops are counted.
15	NULL	Calculates the number of comparisons with null in each method.
16	OVER	Calculates the number of times each non-abstract method is overridden.
17	RETURN	Calculates the total number of return points for each method. This includes any return statements as well as the implicit return at the end of constructors and methods returning void.
18	CALLED	Calculates the number places in the project at which each method may be called. This includes both calls to the method directly and calls to any method which it overrides.
19	CALLEDp	Calculates the number places in the product code of the project at which each method may be called. This includes both calls to the method directly and calls to any method which it overrides.
20	CALLEDt	Calculates the number places in the test code of the project at which each method may be called. This includes both calls to the method directly and calls to any method which it overrides.
21	CAST	Calculates the number of typecast or instance of expressions in each non-abstract method. Excessive use of typecasting may be a sign of an ill-structured program.
22	NTP	Calculates the total number of type parameters of each method.
23	RLOC	Calculates ratio of lines of code for a method to the lines of code for it's containing class. Methods which have high relative lines of code values may indicate poor abstraction.

### 3.4 Research Tools and Applications

In this thesis, the following tools used:

**IntelliJ Idea Tool:** “is a special programming environment or integrated development environment (IDE) largely meant for Java. This environment is used especially for the development of programs. It is developed by JetBrains, which was formally called IntelliJ. It is available in two editions: the Community Edition which is licensed by Apache 2.0, and a commercial edition known as the Ultimate Edition. Both of them can be used for creating software which can be sold. What makes IntelliJ IDEA so different from its counterparts is its ease of use, flexibility and its solid design.” (jetbrains.com/idea, 2018).

**WEKA Tool:** “WEKA is a workbench for machine learning that is designed to assist machine learning techniques to a diversity of real-world problems; it provides a working environment for the domain specialist, and it provides wealth interactive tools for data manipulation, result visualization, database linkage, and classification techniques.” (Holmes., et al., 1994).

**Python:** “is developed under an OSI-approved open source license, making it freely usable and distributable, even for commercial use. Python's license is administered by the Python Software Foundation” (python.org/about, 2018)

# Chapter Four

## 4. Experiment Setup

In order to evaluate the proposed methodology, dataset on method level is created by using IntelliJ IDEA tool which extracts 44 features through importing java projects; we decided to use 21 features of them as shown in table 1 after preprocessing the data using WEKA 2017 and excluding the recent 23 feature as shown in table 4 before. After that class label is determined by using error seeding manually.

Using python, datasets were normalized and machine learning algorithms are applied on them for software fault proneness prediction. To evaluate the results, evaluation measures are used in python.

### 4.1 Evaluation Measures

Six evaluation measures are used in this thesis, which are in the following table.

Table 5: Evaluation Measures

Evaluation Measure	equation
Accuracy	$TP+TN/(TN+FP+FN+TP)$
Error Rate	$FP+FN/(TN+FP+FN+TP)$
Precision	$TP/(TP+FP)$
Recall	$TP/(TP+FN)$
F-measure	$2 (\text{precision} * \text{recall})/(\text{precision}+\text{recall})$
Specificity	$TN/(TN+FP)$

TP: true positive    TN: true negative    FP: false positive    FN: false negative

True Positive is the correctly predicted positive result, while the False Positive is the incorrectly predicted positive result. (Sathyaraj and Prabu, 2015). True Negative is the correctly predicted negative result, while the False Negative is the incorrectly predicted negative result (Malhotra, R., 2015). The evaluation measures used to evaluate the results gained from applying machine learning algorithms.

#### **4.2 Experiment 1: Extracting Metrics**

Intellij IDEA tool is used to extract the features from java projects. Metrics reloaded plug-in is installed within the tool to get metrics values (features), then java projects are imported into the tool through import project tab. To get metrics values, from analyze tab, calculate metrics is chosen. After that metrics profiles are named and determined to be on method level. And finally, the results are exported to CSV file.

#### **4.3 Experiment 2: Error Seeding and Mutation**

Mutation testing is one of the white box testing which is very fascinating to researcher due to its approach to improve quality of software. In this testing technique the software is tested to check the completeness of test suite which in turns ensures the quality of software. In Mutation testing simple bugs are introduced in the program to check the adequacy of the test suite. If test suite fails to identify the seeded faults then effective test cases are added to it to make it sufficiently strong. The objective of mutation testing is to find the flaws of test suite and then modifying suite to ensure its reliability in finding errors.



Mutation testing is based on two assumptions: the competent programmer hypothesis and the coupling effect. The competent programmer hypothesis supposes that although program is written by skilled programmer but it may not be error free. It may contain very small error that may deviate program output of program from the intended one. The coupling effect is based on fact that detection of small errors may cause the identification of big faults. That is simple errors in a program may be associated with complex error (Khan, T., 2015).

Mutants are the key components of the mutation testing. A mutant is version of original program under test in which simple bug is added intuitively. Each mutant contains one simple error. The success of mutation testing depends on the number of mutants generated. On the basis of concept of mutant generation, Mutants can be of different types: Syntactical mutants: Mutants that are generated by making change in syntax of the program. These mutants can be detected by the compiler. For example:  $x=zy++$ . Minor mutants: Mutants that can be detected by any test case of the test suit. Equivalent mutant: These mutants are not detected by any test case because, these are not actually errors. These mutants produce the same output as that of original program. Value mutant: these mutants are generated by changing the value of constants and variable across the boundary values that is by replacing values to either too large or too small numbers. Condition Mutants: These mutants are generated to check the efficiency of test cases related to decision control statements accuracy. This can be done by replacing arithmetic, relational or logical operators in conditions. Statement mutants: In these mutants the statements are removed, replaced or duplicated to check the efficiency of test cases. From the point of testing the

value mutants, condition mutants and statement mutants are more useful as they help to find the inefficiency of test cases

Mutation testing involves generation of mutants, testing and analyzing the outcomes. The whole process can be implemented by following the given steps:

Step 1: Generation of mutants: For the program under test various mutants are generated. These mutants may be generated by introducing errors by replacing any operator, operand or statement of the program.

Step 2: Testing: In second step the original program as well as the generated mutant is tested against all test cases of test suite.

Step 3: Comparison of test outcome: Now the outcome of mutant program is tested with that of original program. If outcome is different, then mutant is killed that is it is not further tested with rest of the cases of test suite. It interprets that test suite is robust enough to handle the particular fault added in the killed mutant.

Step 4: Updating of test suite: In the previous step if the outcomes of mutant and original programs is same for all the test cases of test suite it may further have two interpretations. One the mutant is equivalent mutant of the original program. An equivalent mutant is a version of original program that has different syntax but same semantic as that of original program. Two, the test suite is not adequate to handle that particular fault so more effective test case is added to test suite so that one particular fault can be identified by testing. These steps are repeated for all mutants and for each mutant for all test cases in test suite. But this testing should be stopped if specified reliability of test suit has been achieved

or if pursuing further in testing is resulting in much testing cost as compared to benefit from it (Khan, T., 2015).

#### 4.4 Experiment 3: implementation

The libraries of machine learning algorithms are installed in MATLAB framework and python. The datasets are inserted into MATLAB and python, and five machine learning algorithms are applied on them to get the results. Each machine learning algorithm is repeated 100 times and 10 fold cross validation is used for training and testing datasets. Machine learning algorithms are applied on the dataset before preprocessing and after preprocessing.

The following tables show the result of applying the algorithms on the large scale dataset (processed and unprocessed):

**Table 6: Naive Bays Algorithm on Large Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	90.035	9.965	0.992	0.906	0.947	9.900	4.800	0.327	naive bays
Cassandra	92.752	7.248	0.991	0.935	0.962	26.620	5.580	0.173	
Wicket	92.849	7.151	0.991	0.936	0.963	15.130	4.170	0.216	

**Table 7: Naive Bays Algorithm on Large Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	88.607	11.393	0.994	0.891	0.939	8.150	6.550	0.446	naive bays
Cassandra	88.788	11.212	0.992	0.894	0.940	20.760	11.440	0.355	
Wicket	91.227	8.773	0.992	0.919	0.954	13.650	5.650	0.293	

**Table 8: J48 Algorithm on Large Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	98.958	1.042	0.990	1.000	0.995	14.700	0.000	0.000	J48
Cassandra	95.853	4.147	0.990	0.968	0.979	0.913	0.087	0.087	
Wicket	98.953	1.047	0.990	1.000	0.995	19.300	0.000	0.000	

**Table 9: J48 Algorithm on Large Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	98.944	1.056	0.990	1.000	0.995	14.690	0.010	0.001	J48
Cassandra	98.948	1.052	0.990	1.000	0.995	32.120	0.080	0.002	
Wicket	98.951	1.049	0.990	1.000	0.995	19.060	0.240	0.012	

**Table 10: Decision Table Algorithm on Large Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	98.960	1.040	0.990	1.000	0.995	14.700	0.000	0.000	Decision Table
Cassandra	97.899	2.101	0.979	1.000	0.989	32.000	0.000	0.000	
Wicket	98.953	1.047	0.990	1.000	0.995	19.300	0.000	0.000	

**Table 11: Decision Table Algorithm on Large Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	98.960	1.040	0.990	1.000	0.995	14.700	0.000	0.000	Decision Table
Cassandra	97.910	2.090	0.979	1.000	0.989	31.450	0.550	0.017	
Wicket	98.886	1.114	0.989	1.000	0.994	11.500	0.000	0.000	

**Table 12: Random Forest Algorithm on Large Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	99.013	0.987	0.991	1.000	0.995	13.300	1.400	0.095	Random Forest
Cassandra	97.943	2.057	0.980	0.999	0.990	29.860	2.140	0.067	
Wicket	98.924	1.076	0.990	0.999	0.995	10.520	0.980	0.085	

**Table 13: Random Forest Algorithm on Large Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	98.990	1.010	0.990	1.000	0.995	13.650	1.050	0.071	Random Forest
Cassandra	98.040	1.960	0.981	0.999	0.990	28.740	3.260	0.102	
Wicket	98.942	1.058	0.990	1.000	0.995	10.550	0.950	0.083	

**Table 14: SVM Algorithm on Large Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	98.957	1.043	0.990	1.000	0.995	14.700	0.000	0.000	SVM
Cassandra	97.911	2.089	0.979	1.000	0.989	32.000	0.000	0.000	
Wicket	98.884	1.116	0.989	1.000	0.994	11.500	0.000	0.000	

**Table 15: SVM Algorithm on Large Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Ant	98.960	1.040	0.990	1.000	0.995	14.700	0.000	0.000	SVM
Cassandra	97.911	2.089	0.979	1.000	0.989	32.000	0.000	0.000	
Wicket	98.886	1.114	0.989	1.000	0.994	11.500	0.000	0.000	

The following tables show the result of applying the algorithms on the small scale dataset (processed and unprocessed):

**Table 16: Naive Bays Algorithm on small Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	86.528	13.472	0.997	0.867	0.925	0.100	0.030	0.231	Naïve Bays
apache-log4	92.758	7.242	0.990	0.936	0.962	4.230	0.470	0.100	
cinema	42.536	57.464	0.983	0.428	0.585	0.300	0.100	0.250	
commons-codec	93.015	6.985	0.987	0.940	0.960	1.410	0.290	0.171	
common-lang	23.194	76.806	0.991	0.226	0.368	1.120	4.580	0.804	
iyad-marc4j	29.061	70.939	0.995	0.286	0.429	0.100	0.400	0.800	
jpacman	90.024	9.976	0.991	0.909	0.946	0.200	0.000	0.000	
jtopas1	63.401	36.599	0.997	0.633	0.765	0.110	0.390	0.780	
jtopas2	81.028	18.972	0.992	0.815	0.894	0.320	0.180	0.360	
puremvc	91.171	8.829	0.995	0.916	0.952	0.100	0.200	0.667	
realstate	66.652	33.348	0.991	0.667	0.789	0.210	0.290	0.580	

**Table 17: Naive Bays Algorithm on small Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	88.318	11.682	0.994	0.888	0.936	0.180	0.220	0.550	Naive Bays
apache-log4	87.437	12.563	0.990	0.882	0.924	3.920	0.780	0.166	
cinema	59.077	40.923	0.988	0.595	0.735	0.270	0.130	0.325	
commons-codec	91.439	8.561	0.988	0.924	0.952	1.350	0.350	0.206	
common-lang	25.658	74.342	0.994	0.250	0.400	0.850	4.850	0.851	
iyad-marc4j	35.576	64.424	0.990	0.354	0.513	0.200	0.300	0.600	
jpacman	93.439	6.561	0.991	0.943	0.965	0.200	0.000	0.000	
jtopas1	86.429	13.571	0.998	0.865	0.926	0.100	0.400	0.800	
jtopas2	87.220	12.780	0.994	0.876	0.931	0.270	0.230	0.460	
puremvc	88.554	11.446	0.995	0.889	0.937	0.100	0.200	0.667	
realstate	85.303	14.697	0.995	0.856	0.919	0.200	0.300	0.600	

**Table 18: J48 Algorithm on small Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	98.710	1.290	0.987	1.000	0.993	0.400	0.000	0.000	J48
apache-log4	98.939	1.061	0.989	1.000	0.995	4.700	0.000	0.000	
cinema	98.788	1.212	0.988	1.000	0.994	0.400	0.000	0.000	
commons-codec	98.712	1.288	0.987	1.000	0.994	1.700	0.000	0.000	
common-lang	98.966	1.034	0.990	1.000	0.995	5.700	0.000	0.000	
iyad-marc4j	99.012	0.988	0.990	1.000	0.995	0.500	0.000	0.000	
jpacman	99.091	0.909	0.991	1.000	0.995	0.200	0.000	0.000	
jtopas1	98.848	1.152	0.988	1.000	0.994	0.500	0.000	0.000	
jtopas2	99.000	1.000	0.990	1.000	0.995	0.500	0.000	0.000	
puremvc	98.593	1.407	0.986	1.000	0.993	0.300	0.000	0.000	
realstate	98.971	1.029	0.990	1.000	0.995	0.500	0.000	0.000	

**Table 19: J48 Algorithm on small Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	98.710	1.290	0.987	1.000	0.993	0.400	0.000	0.000	J48
apache-log4	98.939	1.061	0.989	1.000	0.995	4.700	0.000	0.000	
cinema	98.788	1.212	0.988	1.000	0.994	0.400	0.000	0.000	
commons-codec	98.712	1.288	0.987	1.000	0.994	1.700	0.000	0.000	
common-lang	98.966	1.034	0.990	1.000	0.995	5.700	0.000	0.000	
iyad-marc4j	99.012	0.988	0.990	1.000	0.995	0.500	0.000	0.000	
jpacman	99.091	0.909	0.991	1.000	0.995	0.200	0.000	0.000	
jtopas1	98.848	1.152	0.988	1.000	0.994	0.500	0.000	0.000	
jtopas2	99.000	1.000	0.990	1.000	0.995	0.500	0.000	0.000	
puremvc	98.593	1.407	0.986	1.000	0.993	0.300	0.000	0.000	
realstate	98.825	1.175	0.990	0.999	0.994	0.500	0.000	0.000	

**Table 20: Decision Table Algorithm on small Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	98.710	1.290	0.987	1.000	0.993	0.400	0.000	0.000	Decision Table
apache-log4	98.939	1.061	0.989	1.000	0.995	4.700	0.000	0.000	
cinema	98.788	1.212	0.988	1.000	0.994	0.400	0.000	0.000	
commons-codec	98.712	1.288	0.987	1.000	0.994	1.700	0.000	0.000	
common-lang	98.966	1.034	0.990	1.000	0.995	5.700	0.000	0.000	
iyad-marc4j	99.012	0.988	0.990	1.000	0.995	0.500	0.000	0.000	
jpacman	99.091	0.909	0.991	1.000	0.995	0.200	0.000	0.000	
jtopas1	98.848	1.152	0.988	1.000	0.994	0.500	0.000	0.000	
jtopas2	99.000	1.000	0.990	1.000	0.995	0.500	0.000	0.000	
puremvc	98.593	1.407	0.986	1.000	0.993	0.300	0.000	0.000	
realstate	98.971	1.029	0.990	1.000	0.995	0.500	0.000	0.000	



**Table 21: Decision Table Algorithm on small Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	98.710	1.290	0.987	1.000	0.993	0.400	0.000	0.000	Decision Table
apache-log4	98.939	1.061	0.989	1.000	0.995	4.700	0.000	0.000	
cinema	98.788	1.212	0.988	1.000	0.994	0.400	0.000	0.000	
commons-codec	98.712	1.288	0.987	1.000	0.994	1.700	0.000	0.000	
common-lang	98.966	1.034	0.990	1.000	0.995	5.700	0.000	0.000	
iyad-marc4j	99.012	0.988	0.990	1.000	0.995	0.500	0.000	0.000	
jpacman	99.091	0.909	0.991	1.000	0.995	0.200	0.000	0.000	
jtopas1	98.848	1.152	0.988	1.000	0.994	0.500	0.000	0.000	
jtopas2	99.000	1.000	0.990	1.000	0.995	0.500	0.000	0.000	
puremvc	98.593	1.407	0.986	1.000	0.993	0.300	0.000	0.000	
realstate	98.971	1.029	0.990	1.000	0.995	0.500	0.000	0.000	

**Table 22: Random Forest Algorithm on small Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	98.710	1.290	0.987	1.000	0.993	0.400	0.000	0.000	Random Forest
apache-log4	98.939	1.061	0.989	1.000	0.995	4.700	0.000	0.000	
cinema	98.788	1.212	0.988	1.000	0.994	0.400	0.000	0.000	
commons-codec	98.712	1.288	0.987	1.000	0.994	1.700	0.000	0.000	
common-lang	98.966	1.034	0.990	1.000	0.995	5.700	0.000	0.000	
iyad-marc4j	99.012	0.988	0.990	1.000	0.995	0.500	0.000	0.000	
jpacman	99.091	0.909	0.991	1.000	0.995	0.200	0.000	0.000	
jtopas1	98.848	1.152	0.988	1.000	0.994	0.500	0.000	0.000	
jtopas2	99.000	1.000	0.990	1.000	0.995	0.500	0.000	0.000	
puremvc	98.545	1.455	0.986	1.000	0.993	0.300	0.000	0.000	
realstate	98.971	1.029	0.990	1.000	0.995	0.500	0.000	0.000	

**Table 23: Random Forest Algorithm on small Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	98.710	1.290	0.987	1.000	0.993	0.400	0.000	0.000	Random Forest
apache-log4	98.804	1.196	0.990	0.998	0.994	4.460	0.240	0.051	
Cinema	98.788	1.212	0.988	1.000	0.994	0.400	0.000	0.000	
commons-codec	98.712	1.288	0.987	1.000	0.994	1.700	0.000	0.000	
common-lang	98.786	1.214	0.990	0.998	0.994	5.690	0.010	0.002	
iyad-marc4j	99.012	0.988	0.990	1.000	0.995	0.500	0.000	0.000	
Jpacman	99.091	0.909	0.991	1.000	0.995	0.200	0.000	0.000	
jtopas1	98.709	1.291	0.988	0.999	0.993	0.500	0.000	0.000	
jtopas2	99.000	1.000	0.990	1.000	0.995	0.500	0.000	0.000	
Puremvc	98.545	1.455	0.986	1.000	0.993	0.300	0.000	0.000	
Realstate	98.950	1.050	0.990	1.000	0.995	0.500	0.000	0.000	

**Table 24: SVM Algorithm on small Scale Processed Dataset**

processed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	98.710	1.290	0.987	1.000	0.993	0.400	0.000	0.000	SVM
apache-log4	98.939	1.061	0.989	1.000	0.995	4.700	0.000	0.000	
cinema	98.788	1.212	0.988	1.000	0.994	0.400	0.000	0.000	
commons-codec	98.712	1.288	0.987	1.000	0.994	1.700	0.000	0.000	
common-lang	98.966	1.034	0.990	1.000	0.995	5.700	0.000	0.000	
iyad-marc4j	99.012	0.988	0.990	1.000	0.995	0.500	0.000	0.000	
jpacman	99.091	0.909	0.991	1.000	0.995	0.200	0.000	0.000	
jtopas1	98.848	1.152	0.988	1.000	0.994	0.500	0.000	0.000	
jtopas2	99.000	1.000	0.990	1.000	0.995	0.500	0.000	0.000	
puremvc	98.593	1.407	0.986	1.000	0.993	0.300	0.000	0.000	
realstate	98.971	1.029	0.990	1.000	0.995	0.500	0.000	0.000	

**Table 25: SVM Algorithm on small Scale Unprocessed Dataset**

unprocessed data	Accuracy	Error Rate	Precision	Recall	F-measure	False positive	True Negative	Specificity	Algorithm
Apa	98.710	1.290	0.987	1.000	0.993	0.400	0.000	0.000	SVM
apache-log4	98.939	1.061	0.989	1.000	0.995	4.700	0.000	0.000	
cinema	98.788	1.212	0.988	1.000	0.994	0.400	0.000	0.000	
commons-codec	98.712	1.288	0.987	1.000	0.994	1.700	0.000	0.000	
common-lang	98.966	1.034	0.990	1.000	0.995	5.700	0.000	0.000	
iyad-marc4j	99.012	0.988	0.990	1.000	0.995	0.500	0.000	0.000	
jpacman	99.091	0.909	0.991	1.000	0.995	0.200	0.000	0.000	
jtopas1	98.848	1.152	0.988	1.000	0.994	0.500	0.000	0.000	
jtopas2	99.000	1.000	0.989	0.964	0.970	0.500	0.000	0.000	
puremvc	98.593	1.407	0.986	1.000	0.993	0.300	0.000	0.000	
realstate	98.971	1.029	0.990	1.000	0.995	0.500	0.000	0.000	

## Chapter Five

### 5. Results Discussion

A few researches discuss the software fault proneness prediction at method level. One of these few researches discussed the method level fault prediction, but it used historical metrics which related to control version (Hata, et al, 2012). The dataset for this paper is gained through contacting the authors but without details about how it was built depending on class label. This ambiguous dataset motivate us to find the projects of it and create new dataset of these projects with different metrics type and discover how the class label is built.

They used the historical metrics on method level and applied one machine learning algorithm, this motivate us to use object oriented metrics on method level and apply more machine learning algorithms.

The result in our developed approach we gained from applying python on the dataset is compared with (Hata, et al, 2012) depending on error rate. It is obvious that the result in our developed approach using Random Forest is better than their result as shown in Table 28. In this thesis another 4 classifiers used to predict faults on the method level.

**Table 26: Comparison of Error Rate**

Dataset	Error Rate	Error Rate (Hata, et al, 2012)	Algorithm
Ant	1.010	1.600	Random Forest
Cassandra	1.960	6.300	
Wicket	1.058	0.800	

Based on comparison between processed and unprocessed datasets for the large scale as shown in the tables before, it is obvious that the processed data is a little bit better than the unprocessed data for the large scale datasets.

While based on comparison between processed and unprocessed data for the small scale datasets, it is obvious that there is no effect of preprocessing the dataset to get better results for all algorithms.

Based on the tables before of the evaluation measures the results show that for the accuracy and error rate the best algorithm is Random Forest and the descending order for the algorithms is: Random Forest, Decision Table, J48, SVM and the last one is Naïve Bays. While for the precision the descending order is Naïve Bays, Random Forest, J48, SVM and Decision Table. For the recall the order is Decision Table, SVM, J48, Naïve Bays and Random Forest. The false-positive order is Decision Table, SVM, J48, Random Forest and Naïve Bays. F-measure order is Random Forest, SVM, Decision Table, J48 and Naïve Bays. Finally, the Specificity and True-negative order is Naïve Bays, Random Forest, J48, Decision Table and SVM.

## Chapter Six

### 6. Conclusion and Future Work

Software fault proneness prediction on the method level was done using building a predictive model that use five machine learning algorithms which are Random Forest, J48, Naïve Bays, Decision Table and SVM and selected object oriented metrics mentioned before.

The result in our developed approach is compared with (Hata, et al, 2012) depending on error rate. It is obvious that the result in our developed approach using Random Forest is better than their results.

Based on comparison between processed and unprocessed datasets for the large scale as, it is obvious that the processed data is a little bit better than the unprocessed data for the large scale datasets. While based on comparison between processed and unprocessed data for the small scale datasets as, it is obvious that there is no effect of preprocessing the dataset to get better results for all algorithms.

The evaluation measures the results show that for the accuracy and Error rate the best algorithm is Random Forest and the descending order for the algorithms is: Random Forest, Decision Table, J48, SVM and the last one is Naïve Bays. While for the precision the descending order is Naïve Bays, Random Forest, J48, SVM and Decision Table. For the recall the order is Decision Table, SVM, J48, Naïve Bays and Random Forest. The false-positive order is Decision Table, SVM, J48, Random Forest and Naïve Bays. F-measure

order is Random Forest, SVM, Decision Table, J48 and Naïve Bays. Finally, the Specificity and True-negative order is Naïve Bays, Random Forest, J48, Decision Table and SVM.

In this thesis three out of rate projects are used for the software fault proneness prediction. As a future work, five other projects will be used.

## 7. References

Akour, M., Alsmadi, I. and Alazzam, I., 2017. Software fault proneness prediction: a comparative study between bagging, boosting, and stacking ensemble and base learner methods. *International Journal of Data Analysis Techniques and Strategies*, 9(1), pp.1-16.

Alenezi, M., Banitaan, S. and Obeidat, Q., 2014. Fault-proneness of open source systems: An empirical analysis. *The International Arab Conference on Information Technology (ACIT2014) Synapse*, 1, p.256.

Aljarah, I., Ala'M, A. Z., Faris, H., Hassonah, M. A., Mirjalili, S., and Saadeh, H. 2018. Simultaneous feature selection and support vector machine optimization using the grasshopper optimization algorithm. *Cognitive Computation*, pp. 1-18.

Announcing Apache Wicket 8: Write Less, Achieve More. 2018. Retrieved from <http://wicket.apache.org/>.

Apache Log4j 2. 2018. Retrieved from <https://logging.apache.org/log4j/2.x/>.

Apache. 2018. Apache/commons-lang. Retrieved from <https://github.com/apache/commons-lang>.

Archer, K. J., and Kimes, R. V. 2008. Empirical characterization of random forest variable importance measures. *Computational Statistics & Data Analysis*, 52(4), pp. 2249-2260.

Banitaan, S., Alenezi, M., Nygard, K. and Magel, K., 2013, April. Towards test focus selection for integration testing using method level software metrics. In *Information Technology: New Generations (ITNG), 2013 Tenth International Conference on* (pp. 343-348). IEEE.

Belgiu, M., and Drăguț, L. 2016. Random forest in remote sensing: A review of applications and future directions. *ISPRS Journal of Photogrammetry and Remote Sensing*, 114, pp. 24-31.



Bhargava, N., Sharma, S., Purohit, R., and Rathore, P. S. 2017, October. Prediction of recurrence cancer using J48 algorithm. In *Communication and Electronics Systems (ICCES), 2017 2nd International Conference on* (pp. 386-390). IEEE.

Boucher, A. and Badri, M., 2016, December. Using Software Metrics Thresholds to Predict Fault- Classes in Object-Oriented Software. In *Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD), 2016 4th Intl Conf on* (pp. 169-176). IEEE.

Boulesteix, A. L., Janitza, S., Kruppa, J., and König, I. R. 2012. Overview of random forest methodology and practical guidance with emphasis on computational biology and bioinformatics. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(6), pp. 493-507.

Catal, C., 2011. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4), pp.4626-4636.

Chen, C., Zhang, G., Yang, J., and Milton, J. C. 2016. An explanatory analysis of driver injury severity in rear-end crashes using a decision table/Naïve Bayes (DTNB) hybrid classifier. *Accident Analysis & Prevention*, 90, pp. 95-107.

Giger, E., D'Ambros, M., Pinzger, M. and Gall, H.C., 2012, September. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement* (pp. 171-180). ACM.

Gondra, I., 2008. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2), pp.186-195.

Gupta, D., October, 2016. Mutation Testing: An Error Seeding Software Testing Technique. *International Journal of Advance Research in Science and Engineering*, 5(10).

Gupta, D.L. and Malviya, A.K., 2011. Observations on Fault Proneness Prediction Models of Object-Oriented System to Improve Software Quality. *International Journal of Advanced Research in Computer Science*, 2(2).

Hall, C. 2018. The PureMVC Framework Code at the Speed of Thought. Retrieved from <http://puremvc.org/>.

Hata, H., Mizuno, O. and Kikuno, T., 2012, June. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering* (pp. 200-210). IEEE Press.

Hata, H., Mizuno, O. and Kikuno, T., 2011, September. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution* (pp. 96-100). ACM.

Ho, T. K., Hull, J. J., and Srihari, S. N. 1994. Decision combination in multiple classifier systems. *IEEE transactions on pattern analysis and machine intelligence*, 16(1), pp. 66-75.

Holmes, G., Donkin, A., and Witten, I. H. 1994. "Weka: A machine learning workbench. In Intelligent Information Systems". *Proceedings of the 1994 Second Australian and New Zealand Conference on* (pp. 357-361). IEEE.

Hong, E., 2012, October. Software Fault-proneness Prediction using Random Forest. *International Journal of Smart Home*, 6(4).

Hong, E., 2017. Software Fault-proneness Prediction using Module Severity Metrics. *International Journal of Applied Engineering Research*, 12(9), pp.2038-2043.

Hovemeyer, D. and Pugh, W., 2004. Finding bugs is easy. *ACM Sigplan Notices*, 39(12), pp.92-106.

Huang, C. L., and Wang, C. J. 2006. A GA-based feature selection and parameters optimization for support vector machines. *Expert Systems with applications*, 31(2), pp. 231-240.

JetBrains.com/idea, 2018. Jet Brains Co. IntelliJ®IDEA 7.0 retrieved from <http://www.jetbrains.com/idea/>

Jureczko, M. and Spinellis, D., 2010. Using object-oriented design metrics to predict software defects. *Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej*, pp.69-81.

Kaur, G., and Chhabra, A. 2014. Improved J48 classification algorithm for the prediction of diabetes. *International Journal of Computer Applications*, 98(22).

Khan, T. A., Muzammal, M., and Ijaz, A. 2015, December. On effectiveness of fault-seeding using interaction patterns. In *Frontiers of Information Technology (FIT), 2015 13th International Conference on* (pp. 119-124). IEEE.

Kohavi, R. 1995, April. The power of decision tables. In *European conference on machine learning* (pp. 174-189). Springer, Berlin, Heidelberg.

Koru, A.G. and Liu, H., 2005. Building effective defect-prediction models in practice. *IEEE software*, 22(6), pp.23-29.

Kumar, L., Rath, S. and Sureka, A., 2017. Using Source Code Metrics and Ensemble Methods for Fault Proneness Prediction. *arXiv preprint arXiv:1704.04383*.

Lewis, D. D. 1998, April. Naive Bayes at forty: The independence assumption in information retrieval. In *European conference on machine learning* (pp. 4-15). Springer, Berlin, Heidelberg.

Luo, Y., Ben, K. and Mi, L., 2010. Software metrics reduction for fault-proneness prediction of software modules. *Network and Parallel Computing*, pp.432-441.

MacNeill, C., and Bodewig, S. 2018. Welcome. Retrieved from <http://ant.apache.org/>.

Malhotra, R. and Jain, A., 2012. Fault prediction using statistical and machine learning methods for improving software quality. *Journal of Information Processing Systems*, 8(2), pp.241-262.

Malhotra, R., 2015. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27, pp.504-518.

Malhotra, R., Kaur, A. and Singh, Y., 2010. Empirical validation of object-oriented metrics for predicting fault proneness at different severity levels using support vector machines. *International Journal of System Assurance Engineering and Management*, 1(3), pp.269-281.

Manage massive amounts of data, fast, without losing sleep. 2018. Retrieved from <http://cassandra.apache.org/>.

Marc4j. 2018. Marc4j/marc4j. Retrieved from <https://github.com/marc4j/marc4j>.

Moukhafi, M., El Yassini, K., and Bri, S. 2018. A novel hybrid GA and SVM with PSO feature selection for intrusion detection system. *IJASRE*, 4.'

Mukherjee, S., and Sharma, N. 2012. Intrusion detection using naive Bayes classifier with feature reduction. *Procedia Technology*, 4, pp. 119-128.

Mustafa, M. W., Sulaiman, M. H., Khalid, S. A., and Shareef, H. 2012. Hybrid Genetic Algorithm-Support Vector Machine Technique for Power Tracing in Deregulated Power Systems. In *Real-World Applications of Genetic Algorithms*. InTech.

Onik, A. R., Haq, N. F., Alam, L., and Mamun, T. I. 2015. An analytical comparison on filter feature extraction method in data mining using J48 classifier. *International Journal of Computer Applications*, 124(13).

Openhub.net. 2018. JTopas. Retrieved from <http://www.openhub.net/p/jtopas>.

Python.org, 2018, [online], puthon.org retrieved from <https://www.python.org/about/>

Rathore, S.S. and Kumar, S., 2017. A decision tree logic based recommendation system to select software fault prediction techniques. *Computing*, 99(3), pp.255-285.

Rennie, J. D., Shih, L., Teevan, J., and Karger, D. R. 2003. Tackling the poor assumptions of naive bayes text classifiers. In *Proceedings of the 20th international conference on machine learning (icml-03)* (pp. 616-623).

Rish, I. 2001, August. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence* 3(22), pp. 41-46. New York: IBM.

Rodriguez-Galiano, V. F., Ghimire, B., Rogan, J., Chica-Olmo, M., and Rigol-Sanchez, J. P. 2012. An assessment of the effectiveness of a random forest classifier for land-cover classification. *ISPRS Journal of Photogrammetry and Remote Sensing*, 67, pp. 93-104.

Rutar, N., Almazan, C.B. and Foster, J.S., 2004, November. A comparison of bug finding tools for Java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on* (pp. 245-256). IEEE.

Safavian, S. R., and Landgrebe, D. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3), pp. 660-674.

Saravanan, N., and Gayathri, V. 2017, November. Classification of dengue dataset using J48 algorithm and ant colony based AJ48 algorithm. In *Inventive Computing and Informatics (ICICI), International Conference on* (pp. 1062-1067). IEEE.

Sathyaraj, R. and Prabu, S., 2015. An approach for software fault prediction to measure the quality of different prediction methodologies using software metrics. *Indian Journal of Science and Technology*, 8(35).

Satyanarayana, N., Ramadevi, Y., and Chari, K. K. 2018, January. High blood pressure prediction based on AAA using J48 classifier. In *Signal Processing And Communication Engineering Systems (SPACES), 2018 Conference on* (pp. 121-126). IEEE.

Scanniello, G., Gravino, C., Marcus, A. and Menzies, T., 2013, November. Class level fault prediction using software clustering. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (pp. 640-645). IEEE Press.

SERG-Delft. 2018. SERG-Delft/jpacman-framework. Retrieved from <https://github.com/SERG-Delft/jpacman-framework>.

Singh, Y., Kaur, A. and Malhotra, R., 2009. Comparative analysis of regression and machine learning methods for predicting fault proneness models. *International journal of computer applications in technology*, 35(2-4), pp.183-193.

Singh, Y., Kaur, A. and Malhotra, R., 2009, July. Software fault proneness prediction using support vector machines. In *Proceedings of the world congress on engineering* (Vol. 1, pp. 1-3).

Singhal, S., and Jena, M. 2013. A study on WEKA tool for data preprocessing, classification and clustering. *International Journal of Innovative technology and exploring engineering (IJITEE)*, 2(6), pp. 250-253.

Suresh, Y., Kumar, L. and Rath, S.K., 2014. Statistical and machine learning methods for software fault prediction using CK metric suite: a comparative analysis. *ISRN Software Engineering*, 2014.

Team, A. C. 2018. Apache Commons – Apache Commons. Retrieved from <http://commons.apache.org/>.

Yu, P., Systa, T. and Muller, H., 2002. Predicting fault-proneness using OO metrics. An industrial case study. In *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on* (pp. 99-107). IEEE.

Zhang, Y., Song, W., Li, S., Fu, L., and Li, S. 2018. Risk Detection of Stroke using a Feature Selection and Classification Method. *IEEE Access*.