EMPIRICAL ASSESSMENT OF CODE SMELLS FOR PREDICTING SOFTWARE CHANGE PRONENESS

A dissertation submitted in the partial fulfilment for the award of Degree of

Master of Technology

in

Software Engineering

By

Nakul Pritam

Roll No. 09/SE/2010

Under the esteemed guidance of

Dr. Ruchika Malhotra

Department of Computer Engineering

Delhi Technological University, New Delhi

2011-2012

Certificate

Delhi Technological University

(Govt. of National Capital Territory of Delhi)

Bawana Road, Delhi – 110042

Date: _____

This is to certify that the thesis entitled **'Empirical Assessment of Code Smells for Predicting Software Change Proneness'** done by **Nakul Pritam (09/SE/2010),** for the partial fulfillment of the requirements for the award of the degree of Masters of Technology in Software Engineering, is an authentic work carried out by him under my guidance. The matter embodied in this thesis has not been submitted earlier for the award of any degree or diploma to the best of my knowledge and belief.

Project Guide:

Dr. Ruchika Malhotra

Assistant Professor, Department of Software Engineering

Delhi Technological University, Delhi 110042

Acknowledgement

If you want to succeed you should strike out on new paths, rather than travel the worn paths of accepted success.

- John D. Rockefeller

While it is always possible to travel a new path by yourself, having someone hold your hand while you do so is better. As soon as I wrote that a lot of names came to my mind which I would like to honor before I present to you my work.

I would first hand like to thank Delhi Technological University for giving me the opportunity and the platform for showcasing my abilities and simultaneously molding my career in a shape no other place possibly can. It always felt like home.

I would like express my gratitude to **Prof P. B. Sharma**, Vice Chancellor, DTU, Delhi for giving me permission to undertake the project work at DTU, Delhi for partial fulfillment for the degree of Master of Technology.

I also express my gratitude towards **Prof. Daya Gupta**, Dept. of Computer Engineering, Delhi Technological University, for extending her support & valuable guidance.

The major force behind this work is **Dr. Ruchika Malhotra**, Assistant Professor, Department of Software Engineering. I thank her profusely for her constant encouragement, guidance and moral support during the course of project work. The way she instilled confidence and allowed me to

make my own choices at critical points is something I will always remember. I could call her anytime of the day and talk to her like I talk to my friends.

I would also extend my thanks to the Ph.D scholars of my department for their support. Ms. Ankita Bansal for helping me while I was learning software tools, Ms. Poonam Goel for her constant advice and Ms. Shruti Jaiswal for her laptop.

I would also like to thank my sister Shikha who helped me in preparing all the datasets, my mother for serving food at my table and my father for getting prints of research papers.

Every member of my institution and all faculty members have contributed to my work. The lab assistants who tolerated me while working late, guys at Nescafe for serving me at 5:00 PM and all of my wonderful classmates without whom this project would have been a distant reality.

Nakul Pritam

09/SE/2010

M.Tech (Software Engineering)

4th Semester

ABSTRACT

Poor design choices called anti-patterns manifest themselves in the source code as code smells. Code smell is a synonym for bad implementation and is assumed to make maintenance tasks difficult to perform.

In our previous study we validated the fact that it is possible to determine the degree of change-proneness for a class on the basis of certain code smells in an object-oriented system. The data used for the assessment was source code of Quartz, an open source job scheduler, from two versions 1.5.2 and 1.6.6. A total of 79 classes were examined and the results suggested a clear relationship between code smells and change proneness of a class.

The dataset we used was very small to reach a strong conclusion so we extended our previous work by examining a dataset consisting of 4120 classes spanning 14 software systems. The dataset is created by preprocessing the class files that included removal of classes not common to both versions of the systems used. This was followed by assessment of code smells which was done on the basis of metrics. The dataset finally derived was then analyzed using Machine Learning Methods and the results suggest that code smells can classify a change prone class with a probability of .7 or more and a not change prone class with a probability of .67 or more using Multilayer Perceptron model.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Change in software is one of the most unpredictable elements that a maintenance team encounters over the lifespan of the system. Changing requirements, adaptation to new environment, corrective maintenance and a host of other reasons trigger change in software.

1.1 Background

Research over the years has been able to quantify many attributes of a software system by using patterns and metrics. By using these measures we can easily quantify good and bad aspects of the software and are able to predict stuff that otherwise cannot be predicted. This includes the work done by [3], [6] and [7] in presenting metric suites, each having its own domain of application and speciality.

Even more recently, a quality factor called change proneness has emerged and is used to quantify the amount of change a particular software system has undergone over two successive releases. The quantification can be done at class level and hence the exact change a class went through over two successive releases can be computed.

The most unpredictable component about change is that it is very much tied to software design and the theories we have usually aim at suggesting best practices rather than specifying exactly how a design must be made. Researchers have tried to club change proneness to various other attributes like design patterns, code smells and metrics. [26] analysed the ability of object-oriented metrics to predict change proneness of a class, [27] has established a relationship between change proneness and anti-patterns while [22] has

empirically assessed the influence of patterns on the same. The results however are still in the experiment phase.

1.2 Motivation

Using the work already done in this area, in our last study, we used the conclusions provided by [1] and validated the ability of code smells to predict the degree of change proneness a class exhibits. We used an open-source task scheduler called Quartz as the subject of our study to find the error in classifications that occur in predictions of change proneness made using code smells. Our dataset had 79 classes and the results showed that code smells have more than 70% accuracy in such predictions. The motivation for this study comes from two facts,

- The size of the data set was very small to come to a concrete conclusion.

- The study we conducted did not use any machine learning techniques.

1.3 Statement of Work

In this study, we extend our previous work by examining 14 software systems written in JAVA programming language. The dataset is sufficiently large (contains in excess of 4,000 classes) to prove weather code smells have the ability of predicting the change a class could undergo in subsequent releases.

1.4 Organisation of Thesis

This report is organised in the following manner.

- Chapter 2 summarises the work done by us and other researchers in the field of study.
- Chapter 3 provides the detailed description of the research method. It starts off by highlighting the dependent and independent variables and then moves on to the process used to capture the empirical data used in the study.
- Chapter 4 explains our work in detail with each and every step shown conceptually and diagrammatically.
- Chapter 5 summarises the results.
- Chapter 6 is devoted to conclusions and future work possible in this area.
- Finally, we list down the references for this work.

Chapter 2

RELATED WORK

Change proneness of a class is the odds of it undergoing change in the subsequent version. Changes in a class can occur due to multiple reasons like requirements, adaptive maintenance, corrective maintenance, detected or undetected faults, performance enhancement, etc.

Usually change in a class in measured manually by comparing two versions of the software but [1] has conducted an exploratory study and linked class change proneness to certain code smells. Code smells [5] are bad implementation choices. Mostly, the roots of a code smell lie in the design phase but only in the implementation do they manifest themselves completely. Good implementation choices are called design patterns [8] while bad choices are called anti-patterns. Apart from change proneness, code smells have also been used to study software evolvability.

The first description of anti-patterns was given by [9]. In [10], Fowler defined 22 code smells and suggests the areas where refactoring should be applied. And [11], [12] and [13] all define different classifications of smells and anti-patterns.

A lot of existing work has focussed on detection of smells. Moha et al. proposed DECOR[4] for specification and detection of smells. Many other techniques exist for this purpose, ranging from manual approaches [14], to heuristic based [15] and [16] and many others [17], [18], [19] and [20].

In [21] code smells have been used to identify poorly evolvable structures in software. The term software evolvability means “the ease of further developing a software element” [1].

The results of [1] provide a good foundation to explore further in the direction because if a developer is able to estimate the degree to which a class is change prone the designs can be rectified. The study conducted by [1] was empirically investigated by us in our previous work and found to be correct with a probability of 0.70 or more.

The structure of a class can be analysed by studying the metric values it produces. The earliest and most fundamentally strong metric suite was proposed by [3] where a total of 6 metrics were proposed. This was followed by [6] where a set of metrics that predict maintainability were proposed. In [7] the author proposed a set of design quality metrics which can be used directly at system level. Hence, a large number of metrics has been proposed till date and each set has its own importance.

There has been a great deal of work in change proneness estimation by using object-oriented metrics. Moha et al. have presented a method called DECOR in [4] which specifies and detects code smells.

While it is possible to determine the presence of some code smells without using thresholds like, HasChildren and ClassOneMethod, it is not possible to examine a class for all smells without using a threshold. For example, the smell ChildClass cannot be determined without using a threshold. It needs thresholds for three metrics, namely NOM, LOC and No. of Variables.

Chapter 3

RESEARCH BACKGROUND

In this chapter we will discuss what code smells are and follow it up with the description of smells we have used in our study. Second, we will touch upon the concept of metrics and describe the various metrics selected for this study in detail. Third, we talk about code smells and their relationship with object oriented metrics. We also explain how to use metrics to estimate the presence of code smells in a class. Fourth, we explain the process of data collection and highlight the important aspects of the dataset used in this study. Finally, we point down the dependent and independent variables.

3.1 Code Smells

The number of code smells proposed till date is significantly high but only a few of those are actually needed to do predictions in the domain of software maintainability. We will now look at thirteen popular code smells in this regard. Table 3.1 summarizes the code smells and the criteria for their presence in a class.

Smell	Criteria for Presence
ClassOneMethod	A class with one method only
ChildClass	A class which declares large number of attributes & methods
HasChildren	A class which has a large number of children
LargeClass	A class with large measure of LOC

LowCohesion	A class which lacks cohesion
ComplexClassOnly	A class which declares highly complex methods
FewMethod	A class which declares few methods
ManyAttributes	A class which declares a large number of attributes
OneChildClass	A class having only one direct ancestor
NoInheritance	A class which uses little inheritance
DataClass	A class which holds data but doesn't do a lot of processing
TwoInheritance	A class with a inheritance depth of two or more
NotComplex	A class which is not doing much

Table 3.1 : Code Smells and Criteria for Their Presence

3.1.1 ClassOneMethod

This smell is present in classes which declare only one method. Presence of this code smell means that the class is not doing much. Such a class in future could be modified to contain more methods or could be completely removed from the system if possible. In other words, the class is not complex enough to be part of a software system and could undergo change.

3.1.2 ChildClass

This smell is an indication of poor decomposition. ChildClass smell is present in classes which declare a very high number of attributes and contains a high number of methods. This means that the class should have been broken down further and does much more than what a

single class is supposed to do in a system. Such a class might be broken down into simpler classes in future releases.

3.1.3 HasChildren

This code smell suggests that class has a large number of children. Such a class could limit the maintainability of the system due to large number of classes directly depending on it. Changing such a class could mean changing its child classes. The effect of such a smell could propagate to all its children. The developers may avoid updates to such a class but in case a bug is identified, the amount of changes needed could be huge.

3.1.4 LargeClass

This smell is present in classes with a high count of LOC. In other words, large sized classes exhibit this smell. It is an indication of poor decomposition and high complexity. There is every possibility that such class be decomposed in future versions, if not these classes will see changes more frequently than others.

3.1.5 LowCohesion

This odour suggests lack of cohesion in the class. Low cohesion among classes is an indicator of serious design flaws. The performance of such systems tends to be on the lower side.

3.1.6 ComplexClassOnly

This smell is present in classes which declare methods which are highly complex in nature along with a large set of attributes. This indicates poor decomposition. Classes which are more complex than other classes tend to more buggy than classes not so complex. This is because of the fact that such classes do more than what a class is supposed to do.

3.1.7 FewMethod

This smell is present in classes which declare very few methods. Declaring few methods means that the class is not doing as much as it should. Having a large number of methods points to very high complexity of the class while having a small number of methods corresponds to low complexity.

3.1.8 ManyAttributes

Just like the number of methods is a useful indicator of design time issues so is the number of attributes a class has. A large number of attributes means that the class is holding a lot of data and carries the ManyAttribute odour.

3.1.9 OneChildClass

This smell is present in classes which have only one direct ancestor. It is an indicator of improper use of inheritance. Sometimes it can also indicate use of inheritance where it was not necessary.

3.1.10 NoInheritance

NoInheritance smell is present in classes which have an inheritance depth of zero. It can be an indicator of lack of inheritance use.

3.1.11 DataClass

DataClass smell is present in classes which declare a large number of attributes while declaring relatively few methods. Essentially this means that the class holds a lot of data but doesn't do much with it.

3.1.12 TwoInheritance

This odour is present in classes with a maximum inheritance depth of two or more.

3.1.13 NotComplex

This smell is present in classes which do very little. Presence of this smell means that the class is not complex enough to be part of a software system. Such a class could be removed in the subsequent versions or could undergo change to increase its use in the system.

3.2 Metrics Selected for Study

The class-level metrics are used to indicate the internal properties of a class (LOC, NOM, etc) and the association between classes (CBO, NOC, etc).

In our last study we used only C & K metrics [3] given their simplicity and the amount of information these 6 metrics can alone provide. This time however we use the metrics provided by Understand. Understand is a static analysis tool for maintaining, measuring, & analyzing critical or large code bases [29]. By using Understand we can calculate up to 22 complexity metrics [30], 42 count metrics [31] and 29 object-oriented metrics [32].

Out of these 93 metrics only 7 are required to examine a software system for the smells listed above. Table 3.2 summarizes the metrics selected for this study and is followed by a brief description of each metric.

Metric	Estimation Made
CountDeclMethod	Number of Local Methods Declared in Class
CountLineCode	Lines of Code in the Class
CountDeclInstanceVariable	Number of Instance Variables Declared

CountClassDerived	Number of Direct Successors of a Class
MaxInheritanceTree	Maximum Inheritance Depth, aka DIT
AvgCyclomatic	Average Cyclomatic Complexity of Class
PercentLackOfCohesion	Lack of Cohesion, aka LCOM

Table 3.2 : Object Oriented Metrics Selected for Study

3.2.1 CountDeclMethod

This metric estimates the total number of local methods in a class. The metric is essentially identical to NOM (number of methods) metric proposed by [6]. The NOM value of a class indicates its operational property.

3.2.2 CountLineCode

Number of lines containing source code, including inactive regions. It is essentially same as counting the LOC in a class.

3.2.3 CountDeclInstanceVariable

It is the count of the total number of variables/attributes declared by a class.

3.2.4 CountClassDerived

CountClassDerived corresponds to the number of immediate subclasses for a particular class. The metric CountClassDerived is the same as NOC proposed by C & K [3]. A high value for CountClassDerived implies a high level of reuse for the class as inheritance is also reuse.

3.2.5 MaxInheritanceTree

MaxInheritanceTree corresponds to the maximum depth of inheritance tree for the class. It is essentially the same as DIT proposed by C & K [3]. A high value of MaxInheritanceTree

signifies a very complex inheritance structure while a small value points to poor use of inheritance.

3.2.6 AvgCyclomatic

AvgCyclomatic give us an estimate of the total class complexity. It is the average Cyclomatic complexity of all nested methods for a class [30].

3.2.7 PercentLackOfCohesion

PercentLackOfCohesion is the same as LCOM. It estimates the degree of cohesion for a class. PrecentLackOfCohesion is 100% minus the average cohesion for package entities [32]. If LCOM is low, the class is not very cohesive and vice versa.

3.3 Code Smells & Their Relation to Metrics

Essentially, code smells are just presence of certain metric values over or below a particular threshold. For example, HasChildren code smell for a class can be estimated by taking into account the value of NOC metric for that class. If NOC > 0 for the class under test, it carries the HasChildren smell.

Once a suitable threshold is selected we can simply check the class for metric values. Every smell has one or more metrics associated to its presence. Table 3.3 shows the odours along with the metrics associated to it. All that is to be done is to check the corresponding metric(s) beyond the specified threshold. If it is present, the odour exists, otherwise not.

Smell	**Metrics Used to Investigate Their Presence**
ClassOneMethod	CountDeclMethod
ChildClass	CountDeclMethod, CountLineCode, CountDeclInstanceVariable

HasChildren	CountClassDerived
LargeClass	CountLineCode
LowCohesion	PercentLackOfCohesion
ComplexClassOnly	AvgCyclomatic
FewMethod	CountDeclMethod
ManyAttributes	CountDeclInstanceVariable
OneChildClass	CountClassDerived
NoInheritance	MaxInheritanceTree
DataClass	CountDeclMethod, CountDeclInstanceVariable
TwoInheritance	MaxInheritanceTree
NotComplex	AvgCyclomatic

Table 3.3: Code Smells and Related Metrics

3.4 Empirical Data Collection

We investigate the results obtained over a set of 14 software systems.

3.4.1 AOI

Art of Illusion is a free, open source 3D modelling and rendering studio. Many of its capabilities rival hose found in commercial programs. Highlights include subdivision surface based modelling tools, skeleton based animation, and a graphical language for designing procedural textures and materials. (artofillusion.com)

We considered versions 2.0 and 2.9 for this study which consisted of 249 common classes. 202 out of these exhibited change while 47 did not.

3.4.2 CheckStyle

CheckStyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard. (checkstyle.sourceforge.net)

We considered versions 5.2 and 5.5 for this study which consisted of 693 common classes. 145 out of these exhibited change while 548 did not.

3.4.3 FreePlane

FreePlane is free and open source software to support thinking, sharing information and getting things done at work, in school and at home. The core of the software consists of functions for mind mapping, also called concept mapping or information mapping, and tools for using mapped information. FreePlane runs on any operating system on which a current version of Java is installed and from USB. (freeplane.sourceforge.net)

We considered versions 1.1.1 and 1.1.3 for this study which consisted of 572 common classes. 29 out of these exhibited change while 543 did not.

3.4.4 jKiwi

The aim of the jKiwi project is to bring to the open source community a software that simply does not exist for free; that is an application capable of doing virtual makeup (concealer paint, eye shadows, blush, contact lenses for eye colours, change lip colours, etc.) and virtual hair styler (try different hair cuts in different colours), by using a given user's photo. (jkiwi.com)

We considered versions 0.91 and 0.95 for this study which consisted of 45 common classes. 23 out of these exhibited change while 22 did not.

3.4.5 Joda

Joda-Time provides a quality replacement for the Java date and time classes. The design allows for multiple calendar systems, while still providing a simple API. The 'default' calendar is the ISO8601 standard which is used by XML. The Gregorian, Julian, Buddhist, Coptic, Ethiopic and Islamic systems are also included, and we welcome further additions. Supporting classes include time zone, duration, and parsing. (joda-time.sourceforge.net)

We considered versions 1.0 and 2.1 for this study which consisted of 135 common classes. 103 out of these exhibited change while 32 did not.

3.4.6 jStock

JStock makes it easy to track your stock investment. It provides well organized stock market information, to help you decide your best investment strategy. (jstock.sourceforge.net)

We considered versions 1.0.5 and 1.0.6 for this study which consisted of 207 common classes. 108 out of these exhibited change while 99 did not.

3.4.7 jText

Schoolprogramm for learning ten-finger-typing. The program is made for a class test. It sends the text, written by the pupil, to the teacher, and checks the text for mistakes. (sourceforge.net/projects/jtext)

We considered versions 5.0 and 5.1 for this study which consisted of 314 common classes. 181 out of these exhibited change while 133 did not.

3.4.8 LWJGL

The Lightweight Java Game Library (LWJGL) is a solution aimed directly at professional and amateur Java programmers alike to enable commercial quality games to be written in Java. LWJGL provides developers access to high performance cross platform libraries such as OpenGL (Open Graphics Library), OpenCL (Open Computing Language) and OpenAL (Open Audio Library) allowing for state of the art 3D games and 3D sound. Additionally LWJGL provides access to controllers such as Gamepads, Steering wheel and Joysticks. All in a simple and straight forward API. (lwjgl.org)

We considered versions 1.0 and 2.8 for this study which consisted of 31 common classes. 26 out of these exhibited change while 5 did not.

3.4.9 ModBus

A high-performance and ease-of-use implementation of the ModBus protocol written in Java by Serotonin Software. Supports ASCII, RTU, TCP, and UDP transports as slave or master, automatic request partitioning and response data type parsing. (sourceforge.net/projects/modbus4j)

We considered versions 1.01 and 1.02 for this study which consisted of 86 common classes. 69 out of these exhibited change while 17 did not.

3.4.10 openGTS

OpenGTS™ ("Open GPS Tracking System") is the first available open source project designed specifically to provide web-based GPS tracking services for a "fleet" of vehicles. To date, OpenGTS™ has been downloaded and put to use in over 100 countries around the world to track many 1000's of vehicles/assets around all 7 Continents. The types of vehicles

and assets tracked include taxis, delivery vans, trucks/trailers, farm equipment, personal vehicles, service vehicles, containers, ships, ATVs, personal tracking, cell phones, and more. (opengts.sourceforge.net)

We considered versions 2.1.6 and 2.4.0 for this study which consisted of 161 common classes. 131 out of these exhibited change while 30 did not.

3.4.11 openRocket

OpenRocket is a free, fully featured model rocket simulator that allows you to design and simulate your rockets before actually building and flying them. The main features include six-degree-of-freedom flight simulation, automatic design optimization, real-time simulated altitude, velocity and acceleration display, staging and clustering support, cross-platform. (openrocket.sourceforge.net)

We considered versions 1.1.6 and 12.03 for this study which consisted of 83 common classes. 34 out of these exhibited change while 49 did not.

3.4.12 Quartz

Quartz is a full-featured, open source job scheduling service that can be integrated with, or used alongside virtually any Java EE or Java SE application - from the smallest stand-alone application to the largest e-commerce system. Quartz can be used to create simple or complex schedules for executing tens, hundreds, or even tens-of-thousands of jobs; jobs whose tasks are defined as standard Java components that may execute virtually anything you may program them to do. The Quartz Scheduler includes many enterprise-class features, such as JTA transactions and clustering. (quartz-scheduler.org)

We considered versions 1.5.2 and 1.6.6 for this study which consisted of 93 common classes. 83 out of these exhibited change while 10 did not.

3.4.13 Spring

The dominant application framework for Java, Spring solves core enterprise development and runtime problems, offering configuration via Dependency Injection; declarative services via AOP; and packaged enterprise services. (sourceforge.net/projects/springframework)

We considered versions 1.2 and 1.2.9 for this study which consisted of 1333 common classes. 588 out of these exhibited change while 745 did not.

3.4.14 SubSonic

Subsonic is a free, web-based media streamer, providing ubiquitous access to your music. Use it to share your music with friends, or to listen to your own music while at work. You can stream to multiple players simultaneously, for instance to one player in your kitchen and another in your living room. Subsonic is designed to handle very large music collections (hundreds of gigabytes). Although optimized for MP3 streaming, it works for any audio or video format that can stream over HTTP, for instance AAC and OGG. By using transcoder plug-ins, Subsonic supports on-the-fly conversion and streaming of virtually any audio format, including WMA, FLAC, APE, Musepack, WavPack and Shorten. (subsonic.org)

We considered versions 2.8 and 4.6 for this study which consisted of 121 common classes. 95 out of these exhibited change while 26 did not.

The data collection for this study is twofold.

Firstly, we downloaded two stable releases of each system listed above. Each set of source files is then pre-processed. Pre-processing included removing all Java files in either version

that are not present in the other version. In totality, all 14 systems combined had approximately 10000 classes in two versions of each system out of which a little more than 8200 were left after pre-processing. This essentially means that we are left with around 4100 unique classes spanning over 14 software systems.

The next step in this process is to calculate the exact change a class has gone through in the two stated versions. For this purpose, we use an open-source tool named CLOC. CLOC examines two versions of the same file and gives as output the following data,

1. The number of lines unchanged.
2. The number of lines added to the prior version.
3. The number of lines deleted from the prior version.
4. The number of lines modified over the two versions.

These outputs are then used to calculate the amount of change as follows,

Total Change = No. of lines added + No. of lines deleted + 2*No. of lines modified

The number of lines modified is multiplied by 2 because modification is the same as deleting one line and adding one line. Figure 3.1 shows the distribution of change in our dataset.

Figure 3.1: Distribution of change-prone and not change-prone classes

Table 3.1 summarizes the dataset we will use for this study,

S.No	Name	Ver. 1	Ver. 2	P/L Used	Total LOC	Total Classes	Classes Exhibiting Change	Classes Without Change
1	AOI	2.0	2.9	Java	58260	249	202	47
2	CheckStyle	5.2	5.5	Java	50461	693	145	548
3	FreePlane	1.1.1	1.1.3	Java	58286	572	29	543
4	jKiwi	0.91	0.95	Java	8851	45	23	22
5	Joda	1.0	2.1	Java	34705	135	103	32
6	jStock	1.0.5	1.0.6	Java	35205	207	108	99
7	jText	5.0	5.1	Java	67875	314	181	133
8	LWJGL	1.0	2.8	Java	2813	31	26	5
9	ModBus	1.01	1.02	Java	4212	86	69	17
10	openGTS	2.1.6	2.4.0	Java	60593	161	131	30
11	openRocket	1.1.6	12.03	Java	9279	83	34	49
12	Quartz	1.5.2	1.6.6	Java	19123	93	83	10
13	Spring	1.2	1.2.9	Java	111665	1333	588	745
14	SubSonic	2.8	4.6	Java	9162	121	95	26

Table 3.1: Summary of the Dataset used

Once we have calculated the exact change for each class we need to examine them for odours of code smells discussed in the previous chapter. To do this we examine the classes for the values of metrics. We use a commercial tool called Understand [29] for this purpose. Understand allows us to estimate the metric values for each system and export the results as a comma-separated-value list.

We then apply the thresholds selected for metrics and mark the truth value of each smell in each class. The thresholds for all metrics along with the corresponding smell are given in Appendix B.

3.5 Dependent & Independent Variables

The dependent variable in this study is change proneness. Our objective is to empirically investigate the relationship between change proneness of a class and the code smells it carries. The dataset contains 14 attributes per tuple, 13 of these are independent variables,

i.e. code smells while the 14th attribute is dependent, i.e. change. The code smells which act as independent variables have been discussed in section 3.1.

Chapter 4

RESEARCH METHODOLOGY

4.1 Methodology

Figure 4.1 provides an outline of the methodology used in this study.

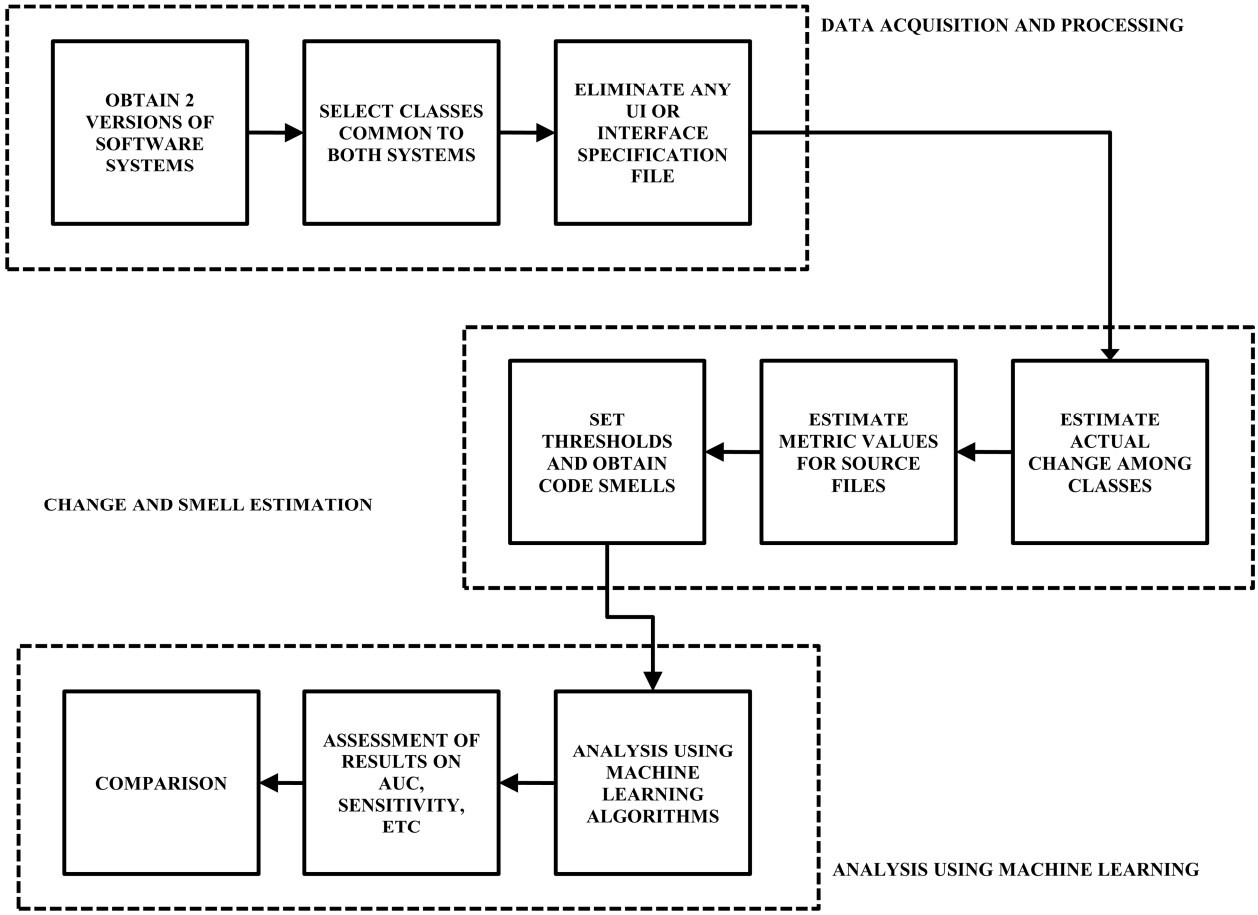


Figure 4.1: Outline of Research Methodology

The entire process can be divided into three parts,

1. Data acquisition and processing.
2. Change and smell estimation.
3. Analysis using Machine Learning Methods.

4.1.1 Data acquisition and processing

In this step the empirical data is collected and processed. All the unnecessary files (files not present in both versions of system, interface files) during this step.

4.1.2 Change and smell estimation

In this step the actual change a class undergoes in two versions is calculated. The details of this calculation are already stated in chapter 3. Following this, we obtain the metric values for each class in each system. These metrics are then used to analyse each class for code smells.

4.1.3 Analysis using Machine Learning Methods

The data obtained after step 2 is then used with six machine learning algorithms (described in section 4.3) to assess the power of code smells in predicting change proneness.

4.2 Class Selector – A Tool to Pre-Process Source Files for Analysis of Change

The first step to estimate change is the selection of common classes in two versions of a software system. Doing so manually is a tiresome process. While conducting this study we found that it is important that we have a tool that automatically discards classes which are not of interest so that the process can be quickened and the chances of error be minimized.

Figure 4.2 shows the home screen of the tool.

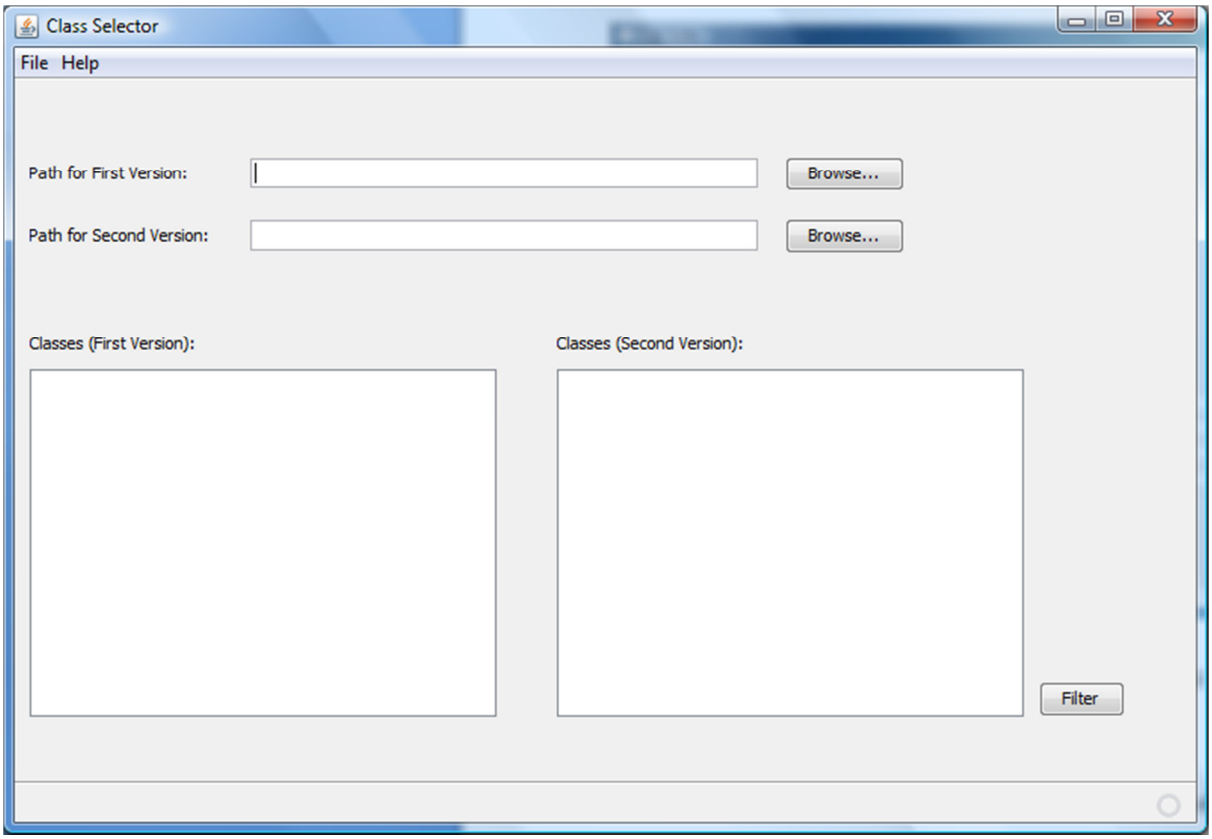


Figure 4.2: Snapshot of Tool

4.2.1 Module 1 – FilesLoader

The FilesLoader module interacts with the directories holding source files for the two versions of the software system under examination. It gives as output two arrays holding the file names of all .java files in the source directories specified.

4.2.2 Module 2 – DataProcessor

The DataProcessor module takes as input the arrays supplied by the FilesLoader module and removes from each iteratively checks each file in each array for presence in the other array. If the file is present in the other array it is left as it is, otherwise it is deleted using native system calls.

Since the system calls used in this tool are native to the Windows OS, it should not be used on UNIX or Linux even when Java provides portability.

4.3 Machine Learning Algorithms for Analysis

4.3.1 Naive Bayes Classifier

Naïve Bayes classifier is a simple probabilistic classifier which is based on the Bayesian theorem which represents a supervised learning method. It naively assumes independence, it is only valid to multiply probabilities when the events are independent [38]. Given a class variable, a Naive Bayes classifier assumes that the presence of a particular feature of a class is not related to the presence of any other feature. Given the set of variables $X = \{x_1, x_2, x_3, \dots\ x_n\}$, a probabilistic classifier can be defined as $p(C|x_1, x_2, x_3, \dots\ x_n)$

Where, C is a dependent class variable with a set of possible outcomes conditional on several variables.

Using Bayes Theorem,

$$\mathrm{p(C)|p}(x_1, x_2, x_3, \dots\ x_n) = \frac{\mathrm{p(C)\ p}(x_1, x_2, x_3, \dots\ x_n\,|C)}{\mathrm{p}(x_1, x_2, x_3, \dots\ x_n)}$$

Thus, we want to construct the posterior probability of the event C. Thus, the equation can be written as:

$$Posterior = \frac{Prior * likelihood}{Evidence}$$

Naïve Bayes algorithm is quite accurate and very fast and therefore, is a popular technique for classification. It is said that Naïve Bayes outperforms more sophisticated classifiers on many datasets, achieving impressive results [37].

4.3.2 Multilayer Perceptron

A Multilayer Perceptron is a feed forward artificial neural network model that maps different input data instances onto a set of appropriate output. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Each node in all the layers is a neuron associated with a nonlinear activation function except for the input nodes. MLP utilizes a supervised learning technique called back-propagation for training the network. MLP is a modification of the standard linear perceptron, which can distinguish data that is not linearly separable.

Fig. 4.3 shows the architecture of Multilayer Perceptron which contains one input layer, two hidden layers and one output layer.

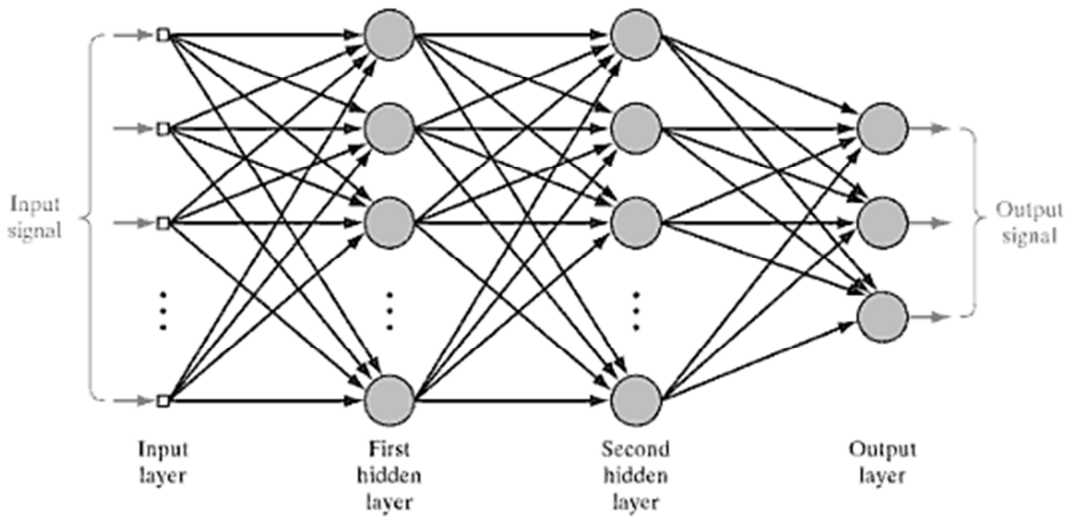


Figure 4.3: Architecture of Multilayer Perceptron with 2 hidden layers

4.3.2.1 The Algorithm

The training of MLP proceeds in 2 phases:

- In the forward phase, the synaptic weights are fixed and the values in the input pattern are propagated through the network layer by layer until it reaches the output.

- In the backward phase, an error is generated by comparing the observed output of the network with the target response. The resulting error is propagated through the network, layer by layer in the backward direction. In this phase successive adjustments are applied to the synaptic weights.

4.3.2.2 Weight Training Calculation in Backward phase

Let the input pattern be E. Let the target and observed response for node ' i' be ti(E) and oi(E) respectively. Let wij to specify weight between node i and node j

1. The Error Term for output unit k is calculated first as:

$$\delta_{O_k} = o_k(E)(1 - o_k(E))(t_k(E) - o_k(E))$$

2. The Error Term for hidden unit k is:

$$\delta_{H_k} = h_k(E)(1 - h_k(E)) \sum_{i \epsilon \, outputs} w_{ki}\delta_{O_i}$$

3. For each weight wij between input node i and hidden node j, calculate

$$\Delta_{ij} = \eta \delta_{H_j} x_i$$

where, x_i is the input to the network to the input node i for input pattern E and η is learning rate.

4. For each weight wij between hidden node i and output node j, calculate:

$$\Delta_{ij} = \eta \delta_{o_j} h_i(E)$$

where, hi(E) is the output from hidden node i for E.

5. Finally, add on each Δ_{ij} on to w_{ij}

$$w_{ij} = w_{ij} + \Delta_{ij}$$

6. In this way, the error is propagated back through the MLP.

4.3.3 LogitBoost

Like AdaBoost, LogitBoost is also a boosting scheme which was proposed by Jerome Friedman, Trevor Hastie and Robert Tibshirani. Boosting is a process of applying a classification algorithm to the training instances, reweighting them again and again, and then taking a majority vote of the number of classifiers thus produced. LogitBoost algorithm takes AdaBoost algorithm as a additive model and applies the cost functional of logistic regression [36]. LogitBoost is suitable for problems involving two class situations.

4.3.4 Bagging

Bagging is an acronym for Bootstrap Aggregating. It was proposed by Leo Breiman [35] in 1994 to improve the classification by combining classifications of randomly generated training sets. It is a machine learning ensemble method to improve machine learning and statistical classification of regression models in terms of stability and classification accuracy. Bagging is a meta-algorithm which is based on averaging the results of various bootstrap samples. It is usually applied to decision tree models, but it can be used with any type of model.

$$\text{Bagging} = \text{Bootstrapping} + \text{Aggregation}$$

A learning set of L consists of data $\{(y_n, x_n)$, n = 1, 2, . . , N$\}$ where the y's are either class labels or a numerical response. Bagging is a procedure for using this learning set to form a predictor $\varphi(\boldsymbol{x}, L)$ — if the input is $\boldsymbol{x}$ we predict y by $\varphi(\boldsymbol{x}, L)$.

4.3.4.1 Aggregation

Suppose we are given a sequence of learning sets $\{L_k\}$ each consisting of N independent observations from the same underlying distribution as L. Our mission is to use the $\{L_k\}$ to get a better predictor than the single learning set predictor $\varphi(\boldsymbol{x}, L)$. The restriction is that all we are allowed to work with is the sequence of predictors $\{\varphi(\boldsymbol{x}, L)\}$. If y is numerical, $\varphi(\boldsymbol{x}, L)$ is replaced by average of $\varphi(\boldsymbol{x}, L_k)$ over k.

$$\varphi_A(\boldsymbol{x}) = E_L\, \varphi(\boldsymbol{x}, L)$$

where E_L denotes the expectation over L, and the subscript A in φ_A denotes aggregation.

4.3.5 Random Forest

Random forest is an ensemble classifier that is made up of many decision trees and outputs the class that is the mode of the class's output by individual trees [33]. The algorithm for inducing a random forest was developed by Leo Breiman and Adele Cutler in 1999. The term "Random Forest" came from "randomized decision forests" that was first proposed by Tin Kam Ho of Bell Labs in 1995. The method combines idea of bagging and the random selection of features, introduced independently by Ho and Amit and Geman in order to construct a collection of decision trees with controlled variation. Breiman [34] defines random forest as follows:

"A random forest is a classifier consisting of a collection of tree-structured classifiers $\{h(x,\Theta_k), k = 1, \ldots\}$ where the Θ_k are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input x."

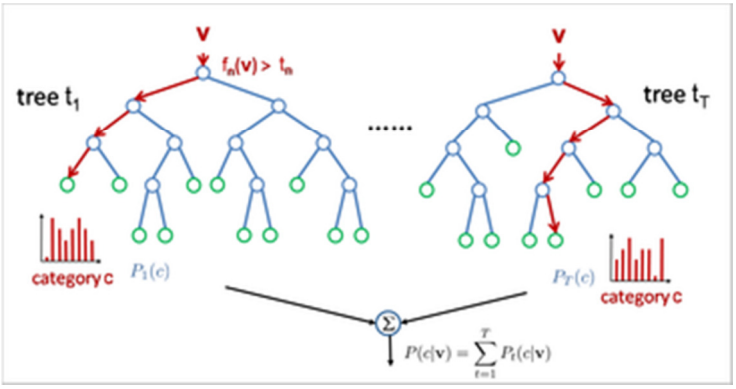


Figure 4.4: A Forest of Trees

4.3.5.1 The Algorithm

The Random Forest algorithm for both classification and regression can be described as follows:

1. Choose T—number of trees to grow.
2. Choose m—number of variables used to split each node. m ≪ M, where M is the number of input variables. m is hold constant while growing the forest.
3. Grow T trees. When growing each tree do the following:
    a. Construct a bootstrap sample of size n sampled from S_n with replacement and grow a tree from this bootstrap sample.
    b. At each node, rather than choosing the best split among all predictor variables, select m variables at random and use them to find the best split.
4. Grow the tree to a maximal extent. There is no pruning.

    (Bagging: special case of random forests obtained when m, number of randomly sampled variables = M, total number of variables)

5. Predict the new data by aggregating the predictions of the T trees (i.e., majority votes for classification, average for regression).

In standard decision trees, each node is split on the basis of the best split among all variables. In a random forest, each node is split using the best among a subset of predictors randomly chosen at that node. This counterintuitive strategy turns out to perform very well when compared to many other classifiers, including discriminant analysis, support vector machines and neural networks, and is robust against over fitting.

4.3.6 Decision Tree

Decision tree learning uses a decision tree as a predictive model whose goal is to create a model that predicts the value of a target variable based on several input variables or attributes.

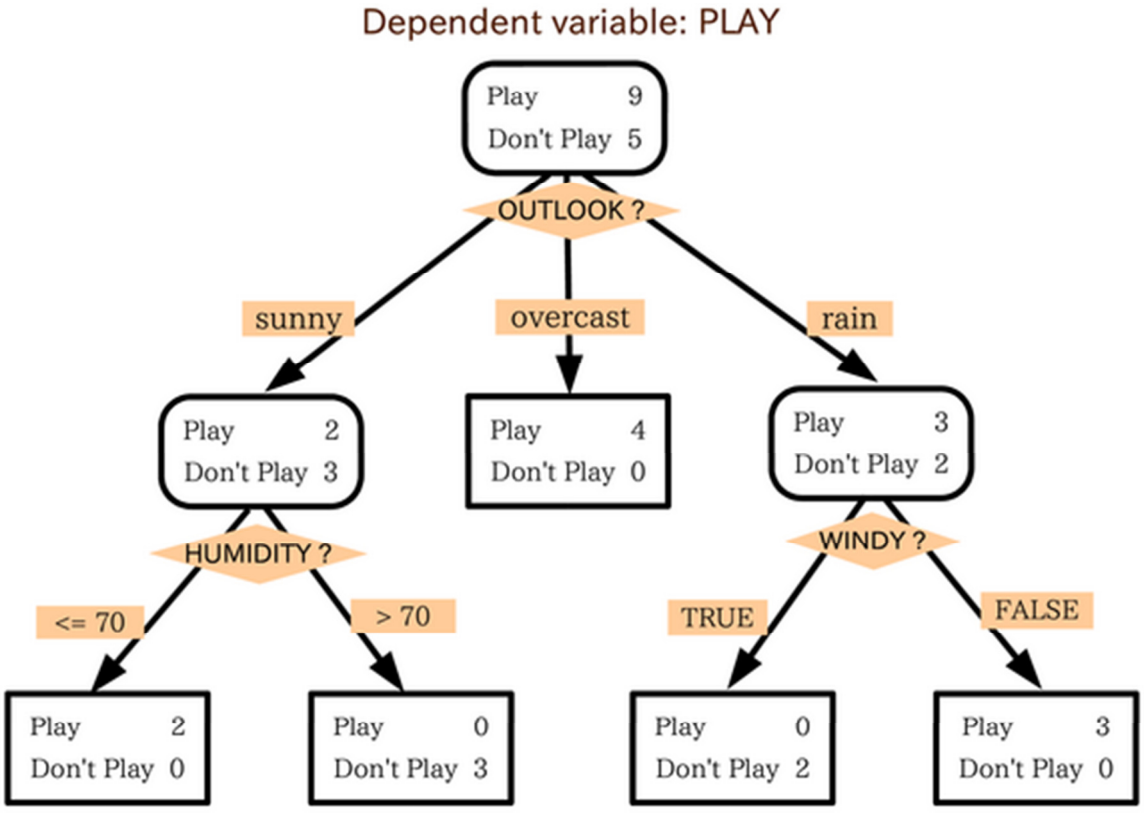


Figure 4.5: A Decision Tree

A Decision Tree is a tree-structured plan of a set of attributes to test in order to predict the output. In these tree structures, leaves represent class labels and branches represent conjunctions of attributes that lead to that class labels. ID3 is one of the decision tree algorithms that we have used for our data analysis.

4.3.6.1 The Algorithm

The ID3 algorithm can be summarized as follows:

1. Take all unused attributes and count their entropy concerning test samples
2. Choose attribute for which entropy is minimum (or, equivalently, information gain is maximum)
3. Make node containing that attribute

The algorithm is as follows:

1. Create a root node for the tree
2. If all examples are positive, Return the single-node tree Root, with label = +.
3. If all examples are negative, Return the single-node tree Root, with label = -.
4. If number of predicting attributes is empty, then Return the single node tree Root, with label = most common value of the target attribute in the examples.
5. Otherwise Begin
    a. A = The Attribute that best classifies examples.
    b. Decision Tree attribute for Root = A.
    c. For each possible value, , of A,
        i. Add a new tree branch below Root, corresponding to the test A = .
        ii. Let Examples() be the subset of examples that have the value for A
        iii. If Examples() is empty

1. Then below this new branch add a leaf node with label = most common target value in the examples

iv. Else below this new branch add the subtree ID3 (Examples(), Target_Attribute, Attributes – {A})

6. End
7. Return Root

Chapter 5

RESULTS

In this chapter, we analyze the effectiveness of code smells in predicting weather a class will undergo change in the subsequent versions or not. The analysis is done on results of a dataset containing 4120 classes from 14 software systems; we have employed 6 machine learning algorithms explained in the previous chapter to predict the model best suited for evaluation the change in software. The measures are used to evaluate the performance of each predicted model are given below:

1. **Sensitivity and Specificity:** The sensitivity and specificity predict the correctness of the model. The percentage of classes correctly predicted to undergo change is called the sensitivity (True Positive Rate i.e. TPR) of the model. The percentage of classes correctly predicted not to change is called specificity (False Positive Rate i.e. FPR) of the model. Ideally, both the sensitivity and specificity should be high.
2. **Receiver Operating Characteristic (ROC) analysis:** The performance of the outputs of the predicted models are evaluated using ROC analysis. It is an efficient method for evaluation of the performance of models.

   The ROC curve is defined as a plot of sensitivity (on the y-coordinate) versus its 1-specificity (on the x coordinate). It is also known as a Relative Operating Characteristic curve, because it is a comparison of two operating characteristics (TPR & FPR). The construction of ROC curves enables us to select cutoff points between 0 and 1, and to determine sensitivity and specificity at each cut off point. The optimal cutoff point is the

one that maximizes both sensitivity and specificity. This point can be selected from the ROC curve.

The accuracy of the model can be determined by applying it to the different data sets. We therefore, performed a 10-cross validation of the models [33]. In 10-cross validation, each dataset is divided into 10 equal subsets. One of the subsets is used as the test set and the other 9 subsets are used to form a training set.

5.1 Random Forest Analysis

For each of the software systems, a random forest of 10 trees is constructed and each constructed while considering 4 random independent variables at each node. Table 5.1 shows the 10-cross validation results of all the 14 software systems.

	Sensitivity	**Specificity**	**Cutoff Point**	**AUC**
AOI	0.809	0.851	0.301500	0.893
CheckStyle	0.531	0.889	0.187000	0.769
FreePlane	0.724	0.670	0.034000	0.740
jKiwi	0.739	0.682	0.346000	0.786
Joda	0.719	0.738	0.162000	0.691
jStock	0.889	0.667	0.350000	0.777
jText	0.915	0.602	0.379500	0.786
LWJGL	1.000	0.885	0.226500	0.915
ModBus	0.647	0.797	0.251000	0.756
openGTS	0.700	0.908	0.425000	0.790
openRocket	0.824	0.837	0.622000	0.866
Quartz	0.800	0.795	0.143000	0.747
Spring	0.701	0.636	0.560500	0.732
SubSonic	0.731	0.958	0.649000	0.848

Table 5.1: 10-Cross validation results for Random Forest

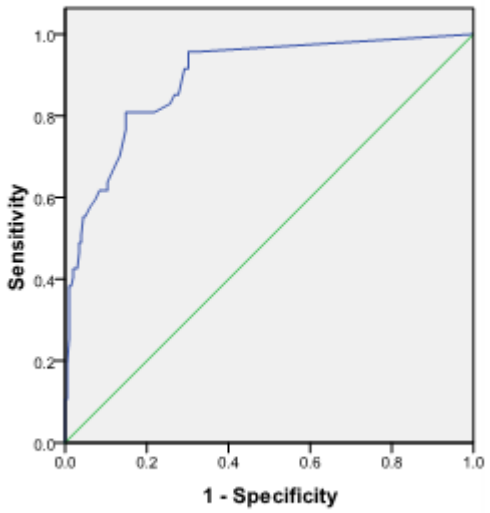


Fig 5.1: ROC Curve for AOI

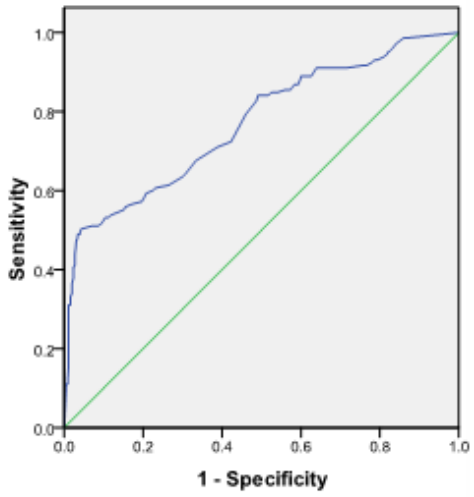


Fig 5.2: ROC Curve for CheckStyle

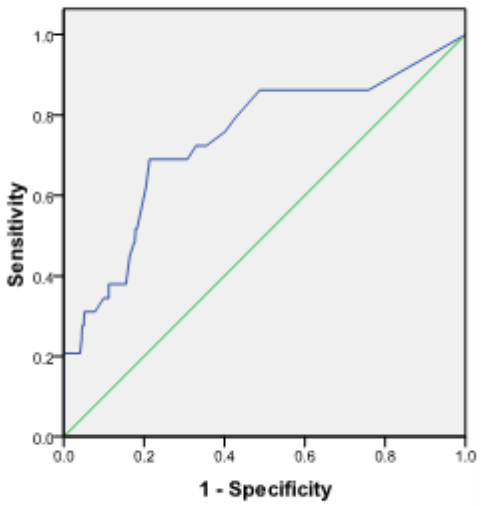


Fig 5.3: ROC Curve for FreePlane

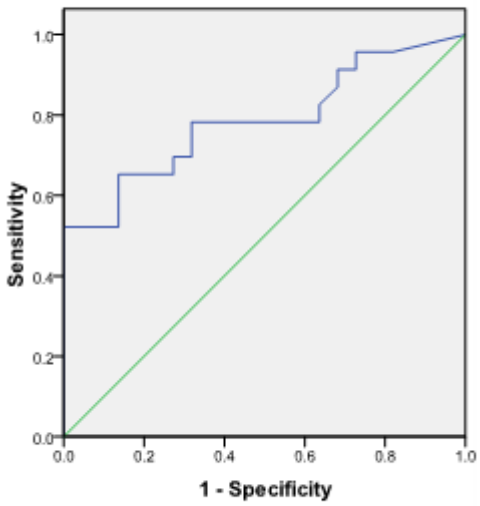


Fig 5.4: ROC Curve for jKiwi

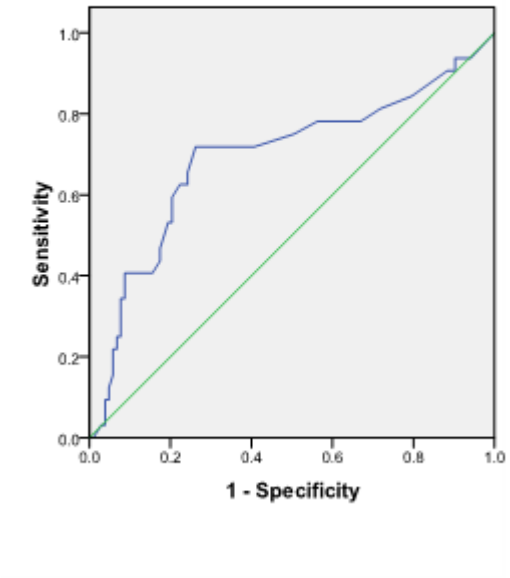


Fig 5.5: ROC Curve for Joda

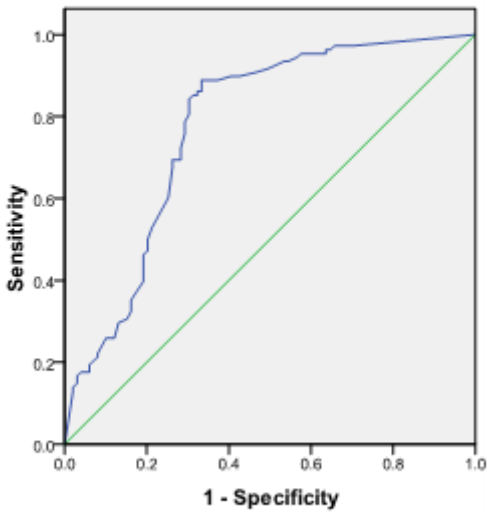


Fig 5.6: ROC Curve for jStock

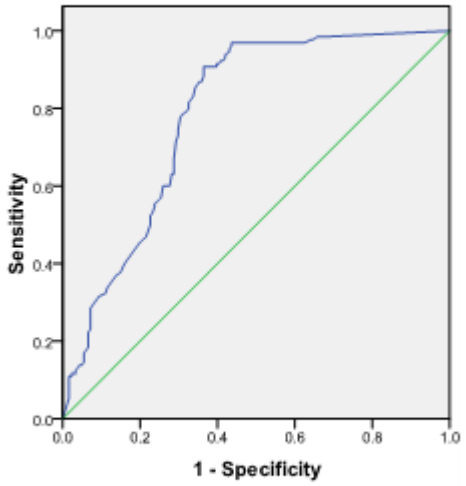


Fig 5.7: ROC Curve for jText

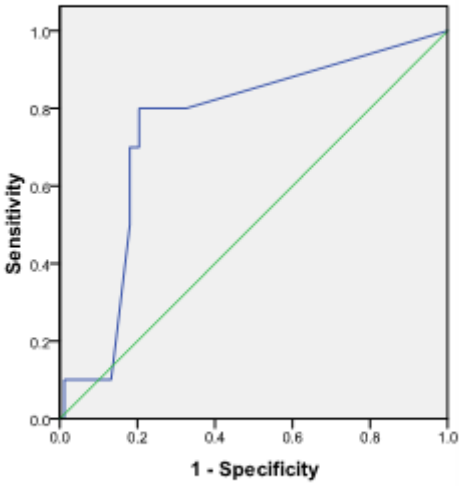


Fig 5.8: ROC Curve for Quartz

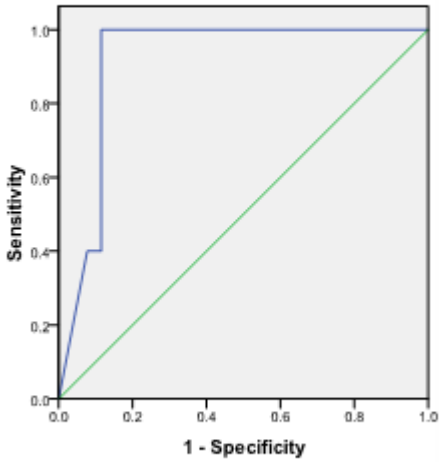


Fig 5.9: ROC Curve for LWJGL

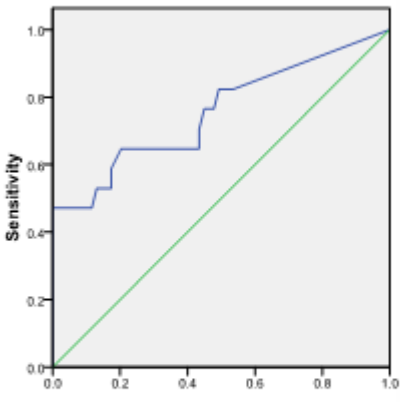


Fig 5.10: ROC Curve for ModBus

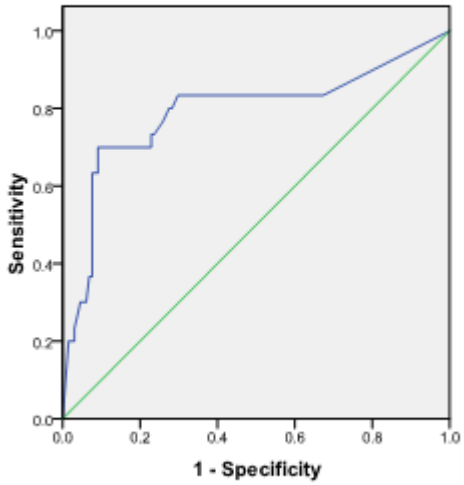


Fig 5.11: ROC Curve for openGTS

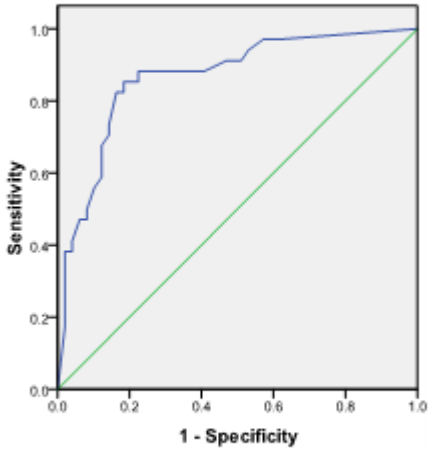


Fig 5.12: ROC Curve for openRocket

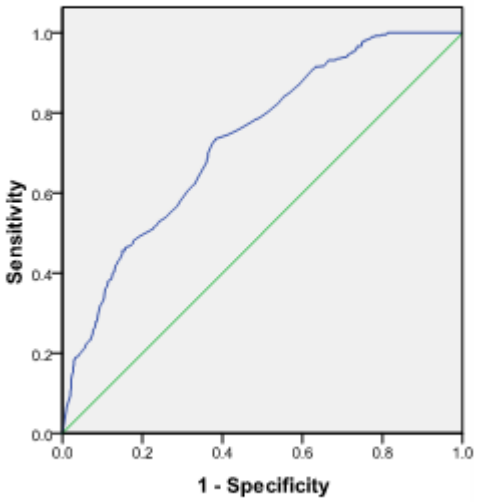


Fig 5.13: ROC Curve for Spring

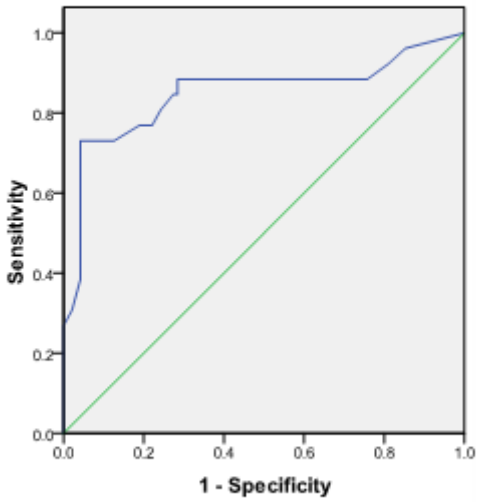


Fig 5.14: ROC Curve for SubSonic

5.2 Naïve Bayes Analysis

Table 5.2 shows the 10-cross validation results of Naïve Bayes classifier for all the 14 software systems.

	Sensitivity	Specificity	Cutoff Point	AUC
AOI	0.830	0.856	0.345000	0.902
CheckStyle	0.538	0.863	0.185000	0.757
FreePlane	0.759	0.746	0.043000	0.784
jKiwi	0.826	0.727	0.259000	0.834
Joda	0.813	0.641	0.156000	0.738
jStock	0.889	0.667	0.338500	0.758
jText	0.931	0.564	0.531500	0.703
LWJGL	1.000	0.885	0.292500	0.912
ModBus	0.765	0.754	0.158000	0.690
openGTS	0.767	0.931	0.403500	0.836
openRocket	0.824	0.816	0.521500	0.874
Quartz	0.900	0.687	0.735000	0.796
Spring	0.650	0.639	0.552500	0.715
SubSonic	0.731	0.937	0.586000	0.862

Table 5.2: 10-cross validation results for Naïve Bayes Classifier

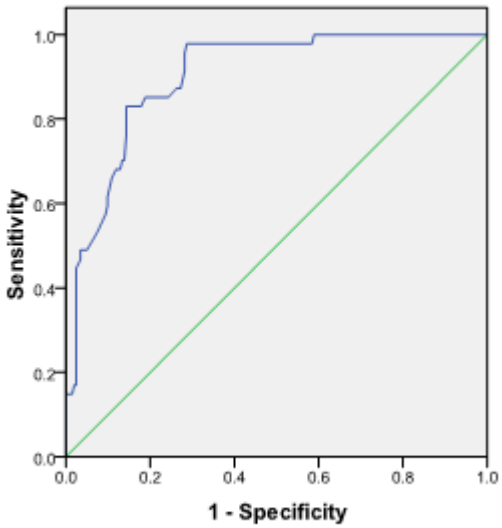


Fig 5.15: ROC Curve for AOI

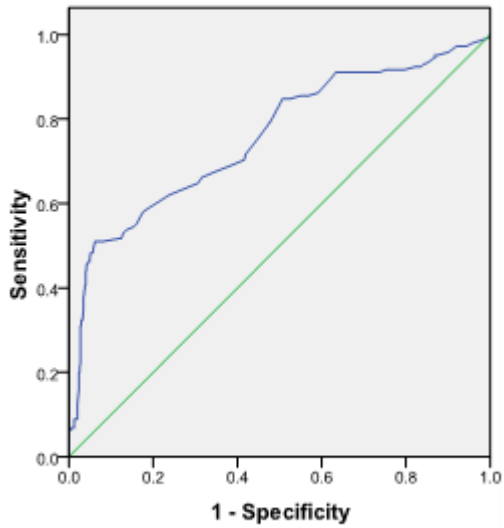


Fig 5.16: ROC Curve for CheckStyle

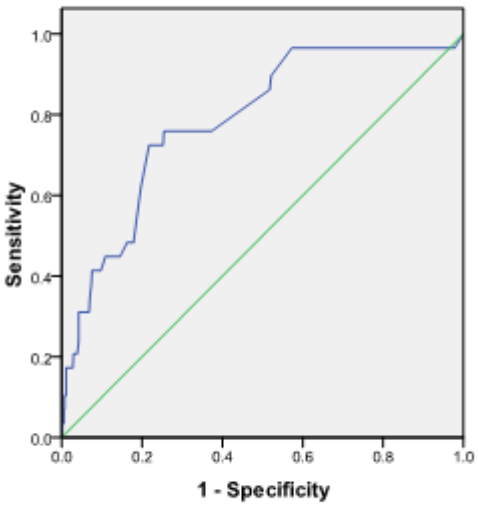


Fig 5.17: ROC Curve for FreePlane

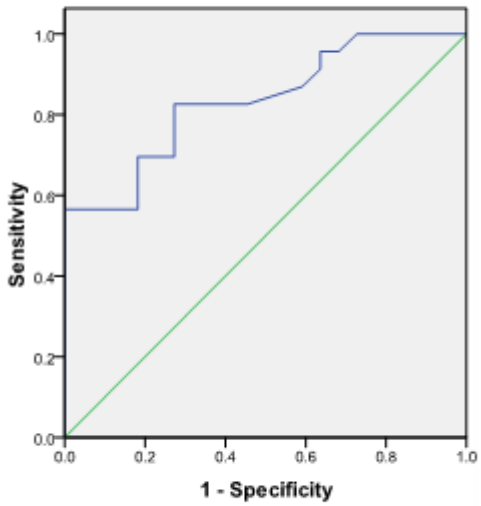


Fig 5.18: ROC Curve for jKiwi

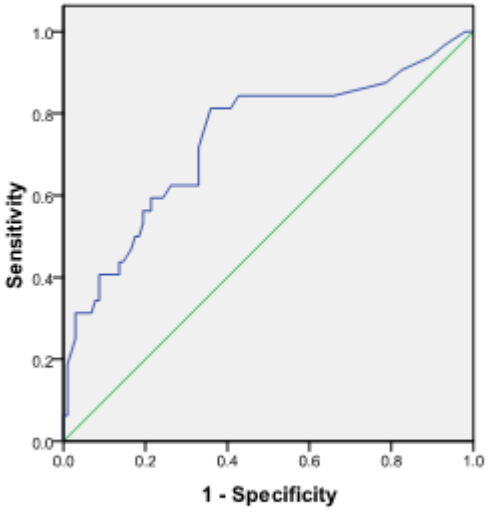


Fig 5.19: ROC Curve for Joda

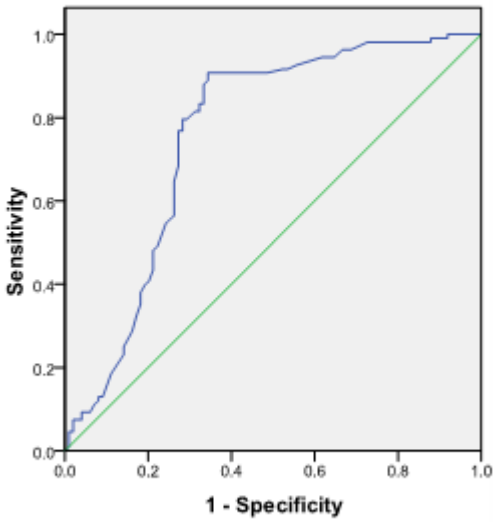


Fig 5.20: ROC Curve for jStock

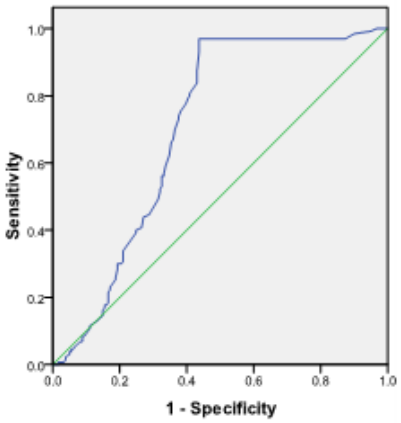


Fig 5.21: ROC Curve for jText

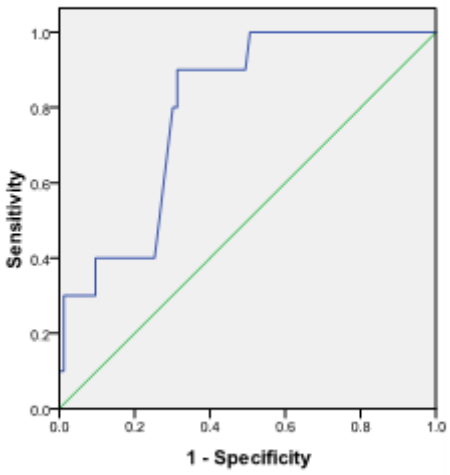


Fig 5.22: ROC Curve for Quartz

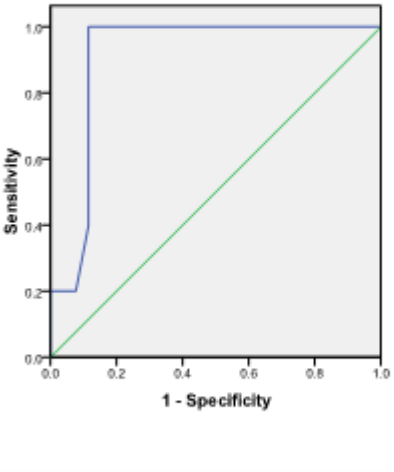


Fig 5.23: ROC Curve for LWJGL

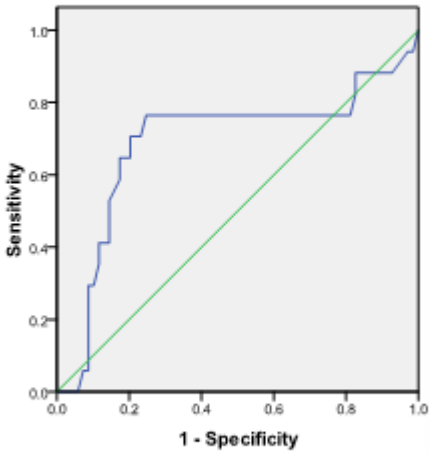


Fig 5.24: ROC Curve for ModBus

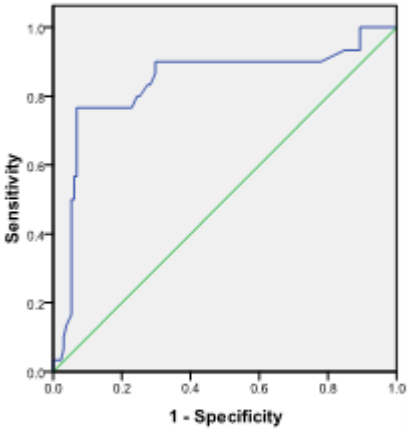


Fig 5.25: ROC Curve for openGTS

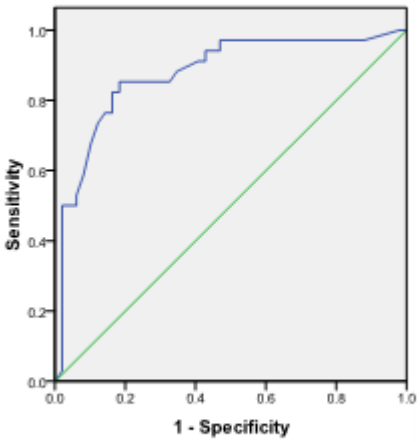


Fig 5.26: ROC Curve for openRocket

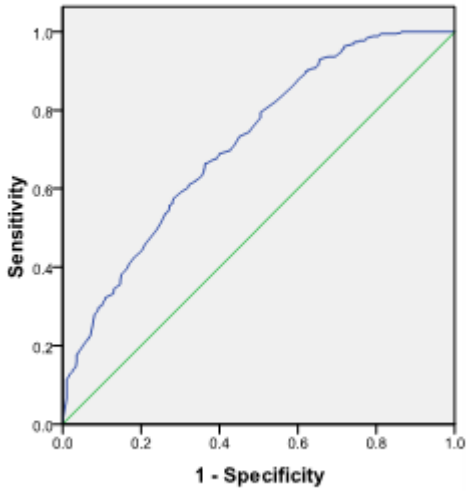


Fig 5.27: ROC Curve for Spring

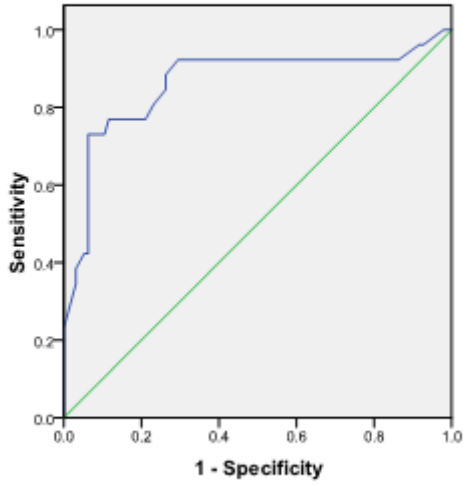


Fig 5.28: ROC Curve for SubSonic

5.3 Bagging Analysis

Table 5.3 shows the 10-cross validation result of bagging for all the 14 software systems.

	Sensitivity	Specificity	Cutoff Point	AUC
AOI	0.830	0.837	0.287500	0.903
CheckStyle	0.510	0.949	0.221500	0.750
FreePlane	0.517	0.836	0.048500	0.689
jKiwi	0.826	0.818	0.378000	0.821
Joda	0.688	0.689	0.201500	0.711
jStock	0.907	0.687	0.329500	0.766
jText	0.931	0.591	0.514000	0.759
LWJGL	1.000	0.985	0.145000	0.908
ModBus	0.647	0.783	0.150000	0.655
openGTS	0.767	0.931	0.346500	0.799
openRocket	0.882	0.857	0.504500	0.833
Quartz	0.400	0.783	0.112000	0.543
Spring	0.723	0.628	0.520500	0.732
SubSonic	0.731	0.937	0.383000	0.791

Table 5.3: 10-cross validation results for Bagging

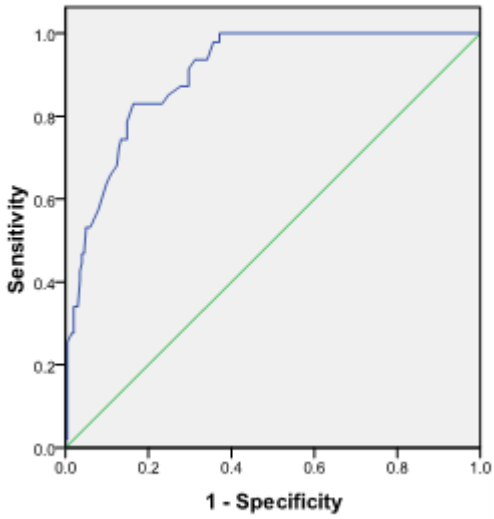


Fig 5.29: ROC Curve for AOI

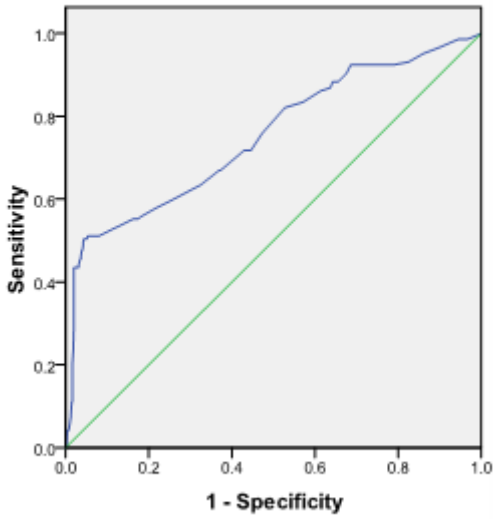


Fig 5.30: ROC Curve for CheckStyle

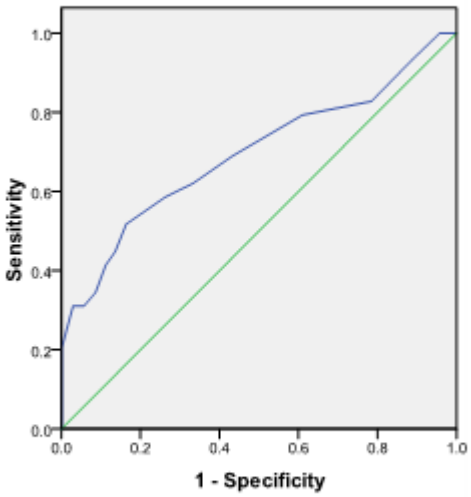


Fig 5.31: ROC Curve for FreePlane

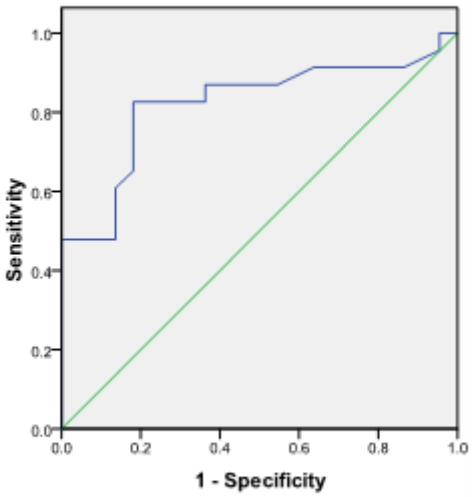


Fig 5.32: ROC Curve for jKiwi

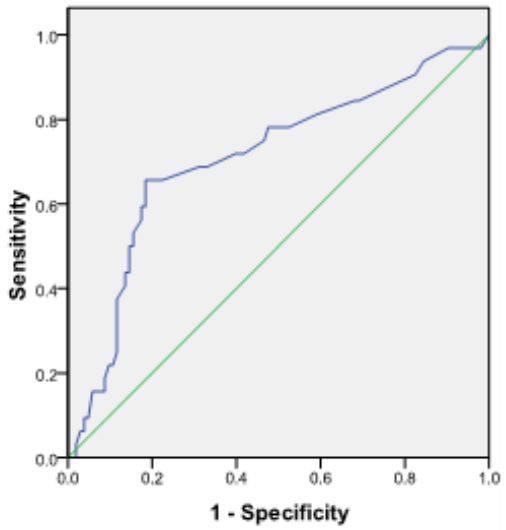


Fig 5.33: ROC Curve for Joda

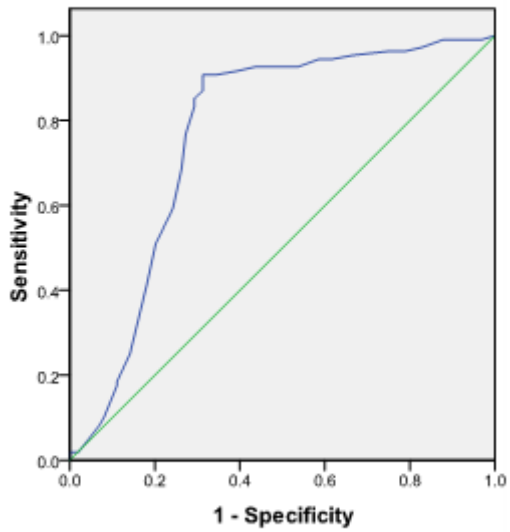


Fig 5.34: ROC Curve for jStock

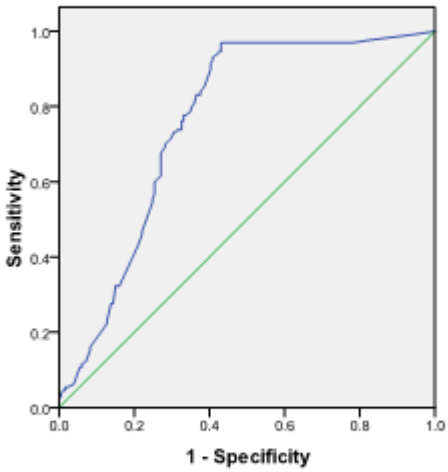


Fig 5.35: ROC Curve for jText

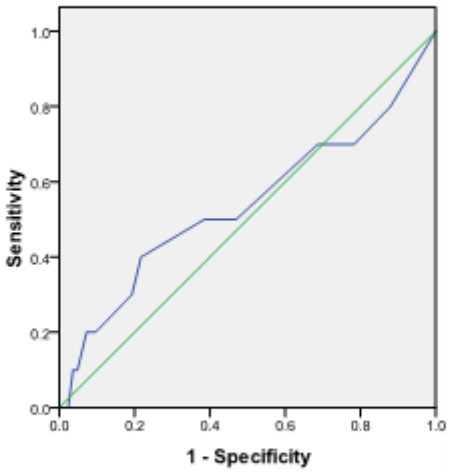


Fig 5.36: ROC Curve for Quartz

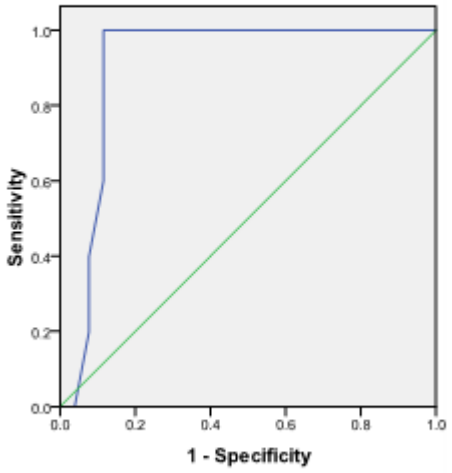


Fig 5.37: ROC Curve for LWJGL

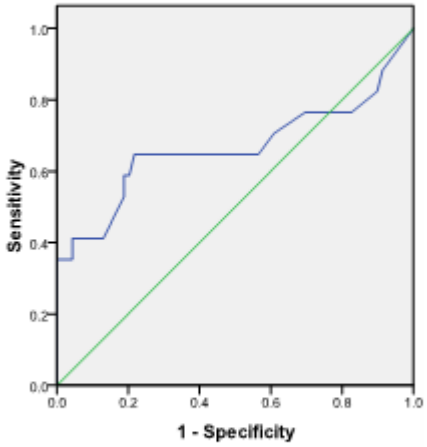


Fig 5.38: ROC Curve for ModBus

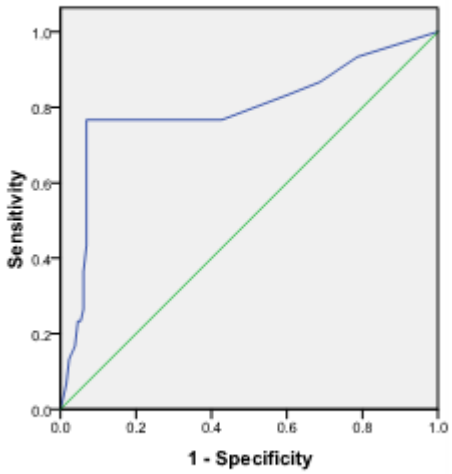


Fig 5.39: ROC Curve for openGTS

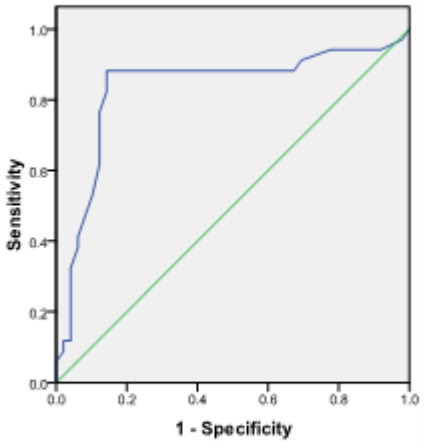


Fig 5.40: ROC Curve for openRocket

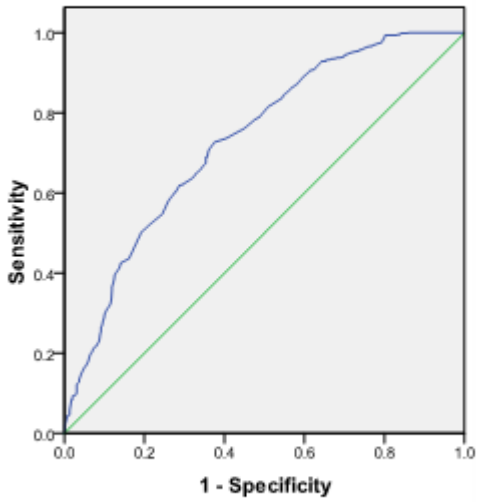


Fig 5.41: ROC Curve for Spring

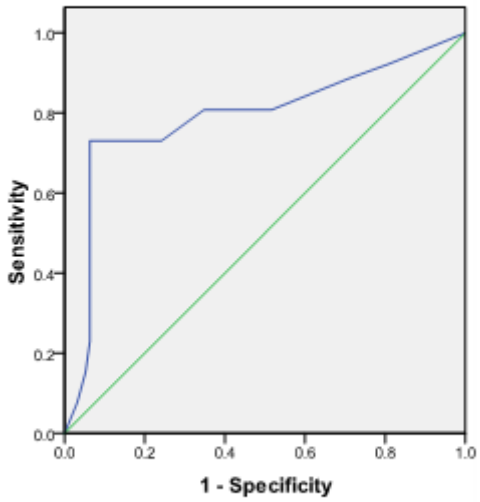


Fig 5.42: ROC Curve for SubSonic

5.4 Decision Tree Analysis

In the Decision Tree method, an independent variable is selected at each node of the tree. The tree is traversed during classification from the root until a leaf node is reached. Each leaf node is associated with a decision or classification. ID3 algorithm is used to create the decision tree. Table 5.4 shows the 10-cross validation results of all the 14 systems.

	Sensitivity	Specificity	Cutoff Point	AUC
AOI	0.809	0.856	0.296000	0.889
CheckStyle	0.510	0.878	0.185000	0.752
FreePlane	0.586	0.820	0.091500	0.741
jKiwi	0.654	0.864	0.583500	0.802
Joda	0.699	0.687	0.882000	0.650
jStock	0.889	0.697	0.370500	0.752
jText	0.923	0.619	0.387500	0.790
LWJGL	1.000	0.885	0.250000	0.904
ModBus	0.647	0.739	0.225000	0.743
openGTS	0.700	0.924	0.550000	0.766
openRocket	0.853	0.857	0.568000	0.858
Quartz	0.600	0.819	0.198500	0.658

Spring	0.622	0.725	0.438000	0.731
SubSonic	0.731	0.937	0.471000	0.840

Table 5.4: 10-cross validation results for Decision Tree

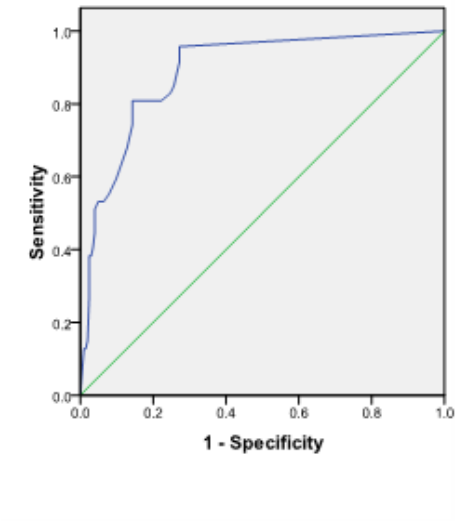


Fig 5.43: ROC Curve for AOI

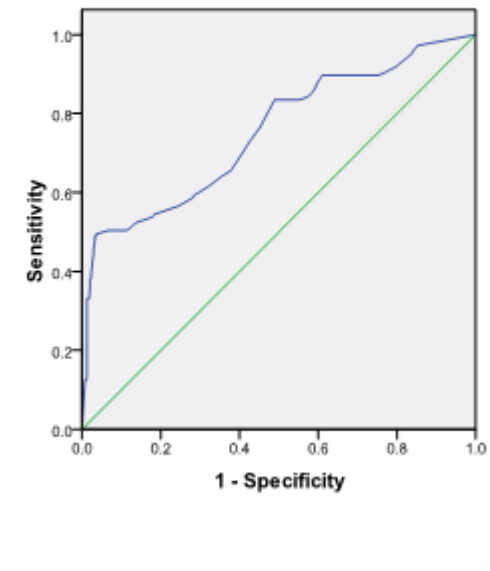


Fig 5.44: ROC Curve for CheckStyle

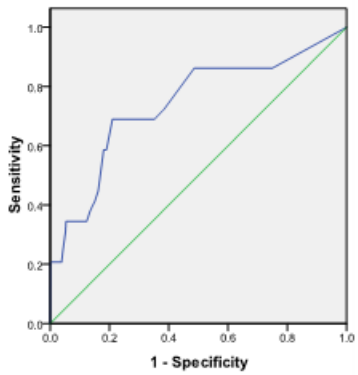


Fig 5.45: ROC Curve for FreePlane

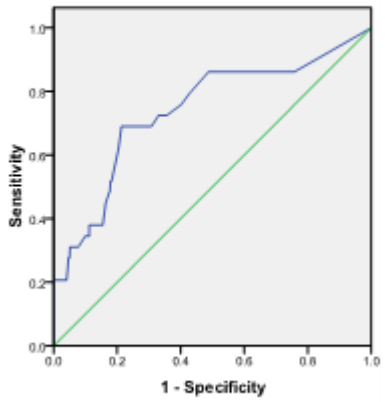


Fig 5.46: ROC Curve for jKiwi

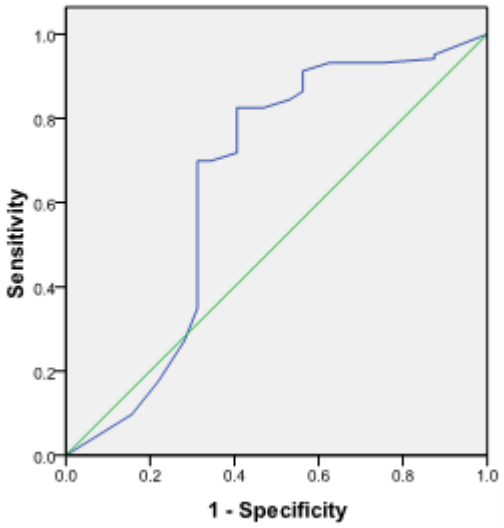


Fig 5.47: ROC Curve for Joda

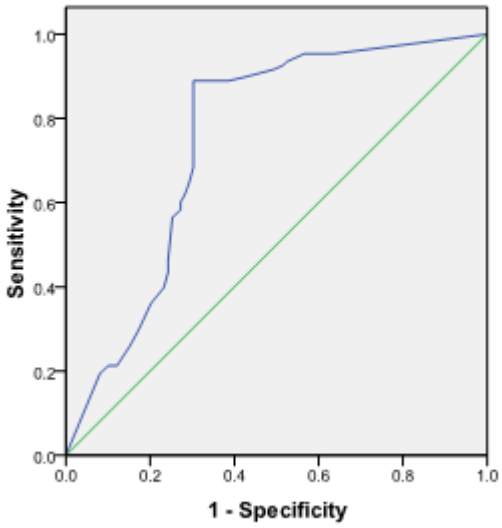


Fig 5.48: ROC Curve for jStock

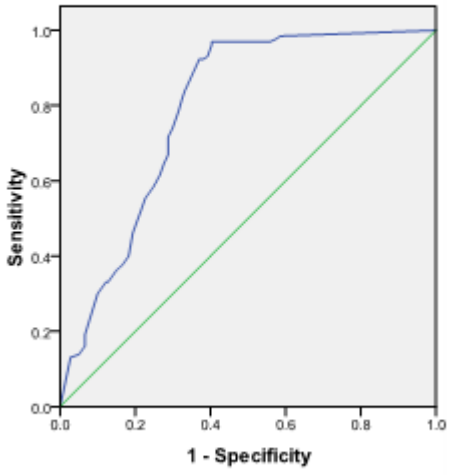


Fig 5.49: ROC Curve for jText

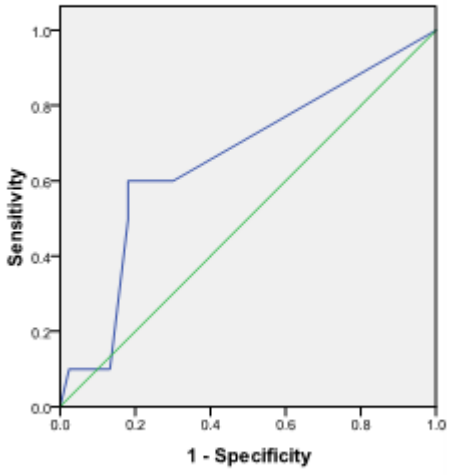


Fig 5.50: ROC Curve for Quartz

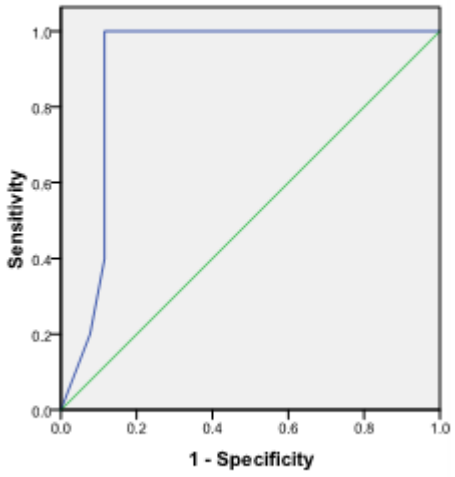


Fig 5.51: ROC Curve for LWJGL

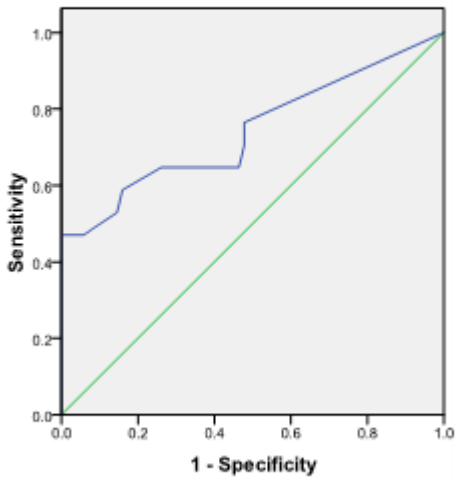


Fig 5.52: ROC Curve for ModBus

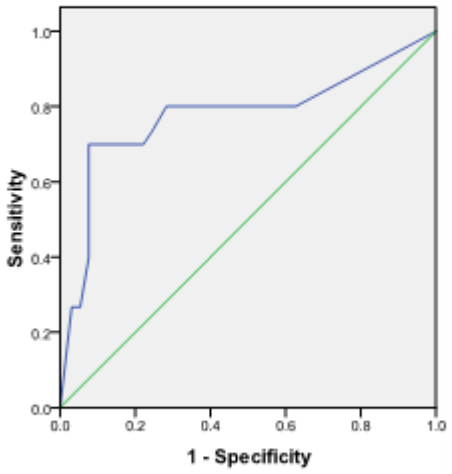


Fig 5.53: ROC Curve for openGTS

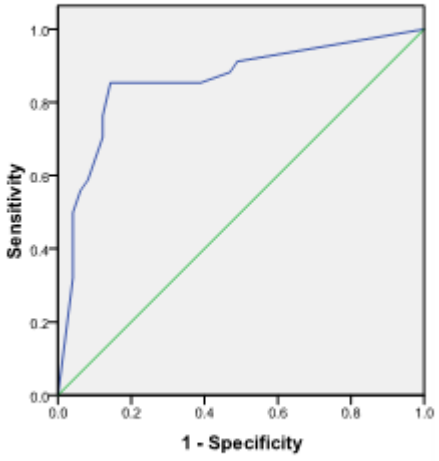


Fig 5.54: ROC Curve for openRocket

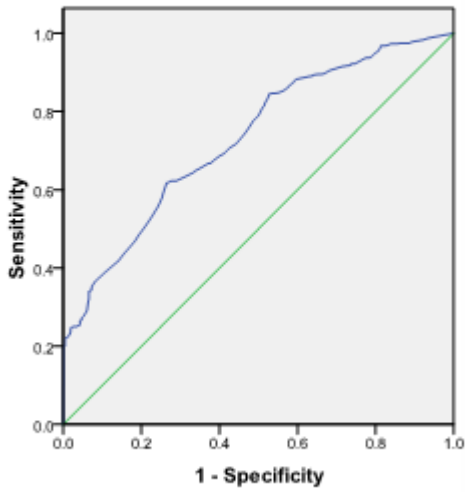


Fig 5.55: ROC Curve for Spring

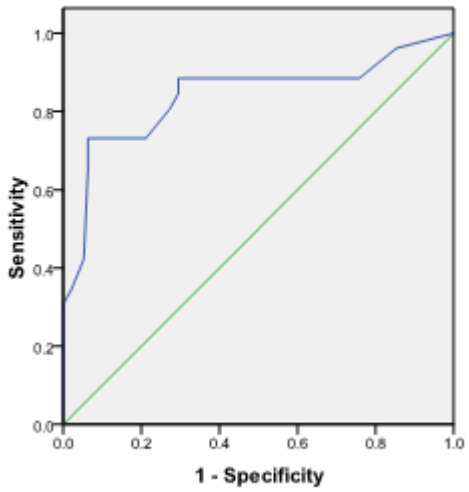


Fig 5.56: ROC Curve for SubSonic

5.5 LogitBoost Analysis

Table 5.5 shows the 10-cross validation results of all the 14 systems.

	Sensitivity	Specificity	Cutoff Point	AUC
AOI	0.809	0.847	0.318000	0.907
CheckStyle	0.531	0.876	0.175500	0.756
FreePlane	0.724	0.790	0.085500	0.759
jKiwi	0.783	0.682	0.232500	0.848
Joda	0.813	0.660	0.123000	0.752
jStock	0.907	0.677	0.442500	0.749
jText	0.923	0.409	0.473000	0.756
LWJGL	1.000	0.885	0.254500	0.900
ModBus	0.647	0.174	0.249000	0.646
openGTS	0.767	0.901	0.209500	0.833
openRocket	0.794	0.878	0.676500	0.878
Quartz	0.900	0.819	0.135000	0.797
Spring	0.619	0.704	0.604500	0.722
SubSonic	0.731	0.937	0.229500	0.866

Table 5.5: 10-cross validation results for LogitBoost

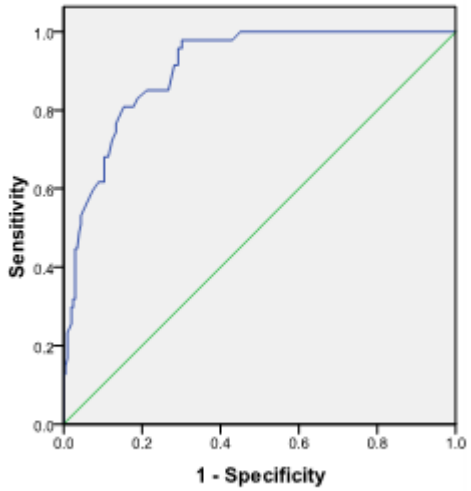


Fig 5.57: ROC Curve for AOI

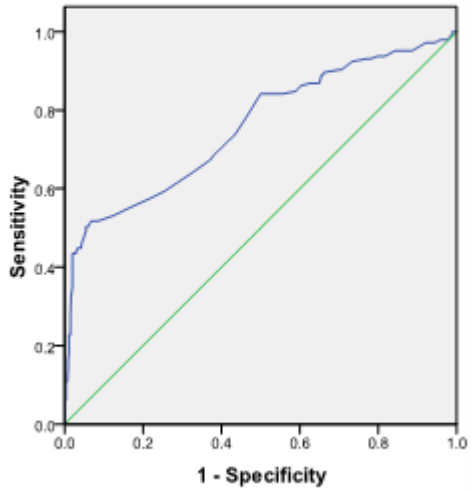


Fig 5.58: ROC Curve for CheckStyle

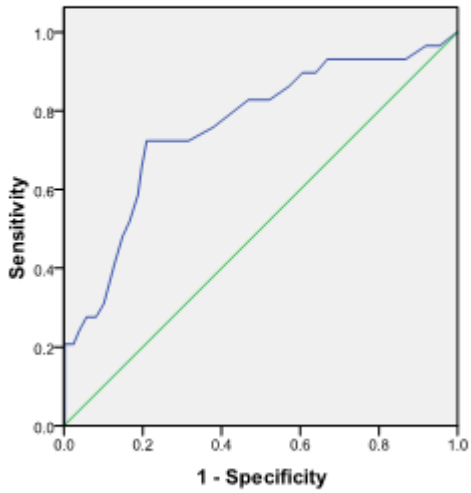


Fig 5.59: ROC Curve for FreePlane

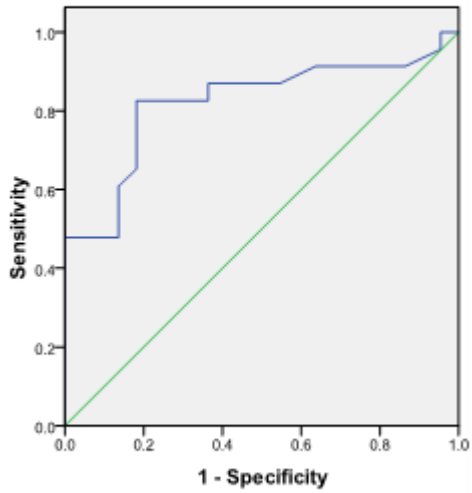


Fig 5.60: ROC Curve for jKiwi

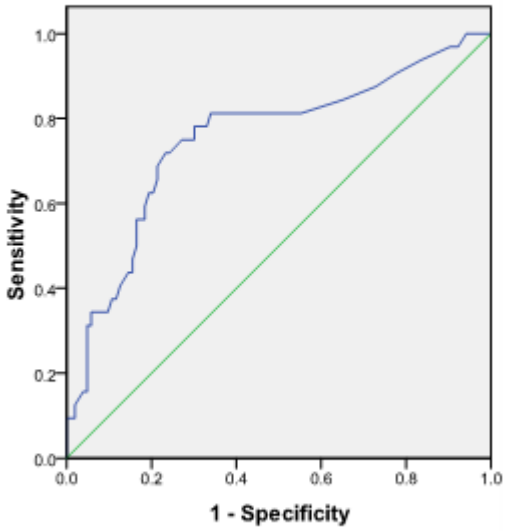


Fig 5.61: ROC Curve for Joda

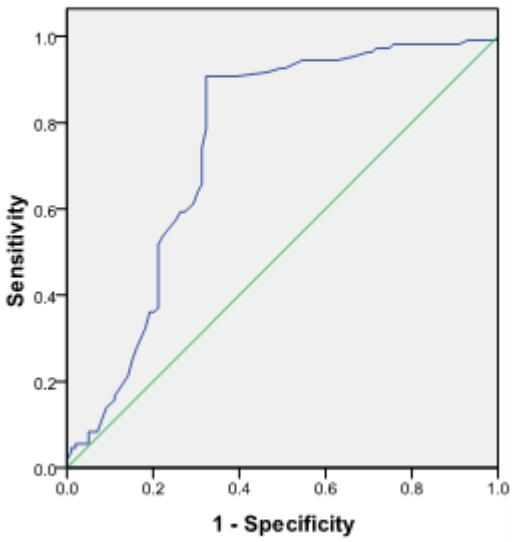


Fig 5.62: ROC Curve for jStock

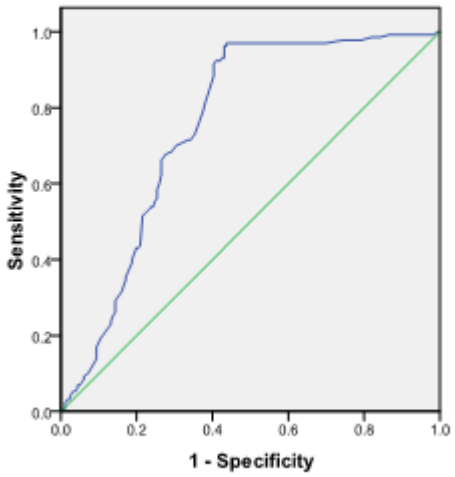


Fig 5.63: ROC Curve for jText

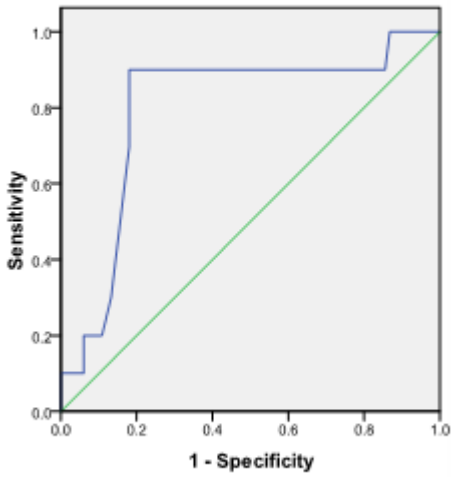


Fig 5.64: ROC Curve for Quartz

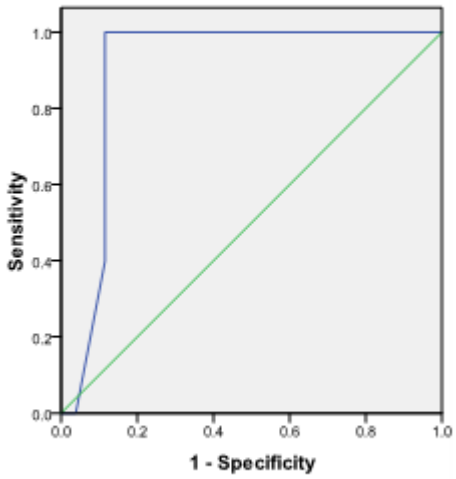


Fig 5.65: ROC Curve for LWJGL

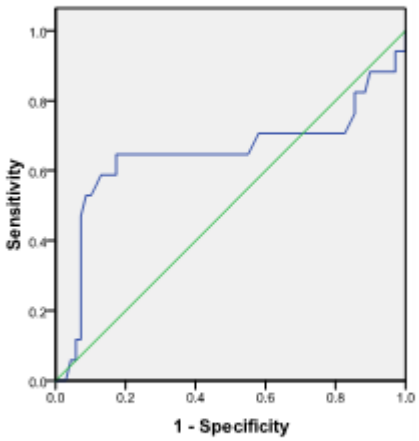


Fig 5.66: ROC Curve for ModBus

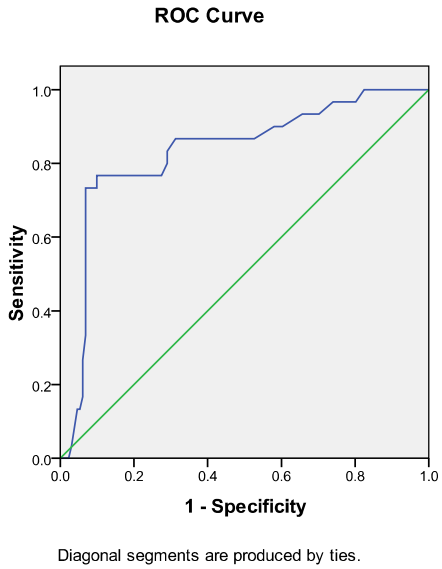


Fig 5.67: ROC Curve for openGTS

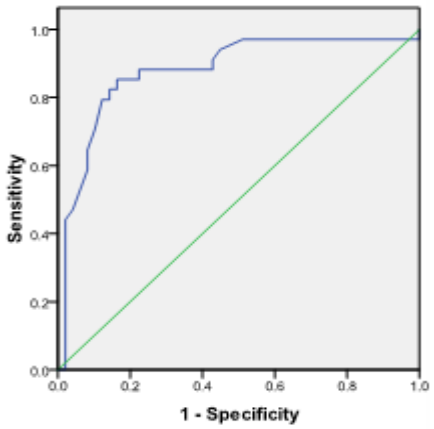


Fig 5.68: ROC Curve for openRocket

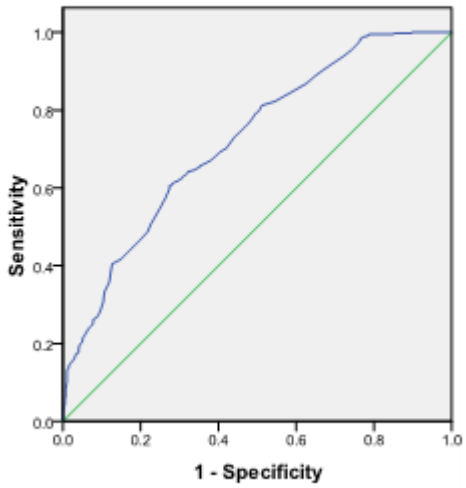


Fig 5.69: ROC Curve for Spring

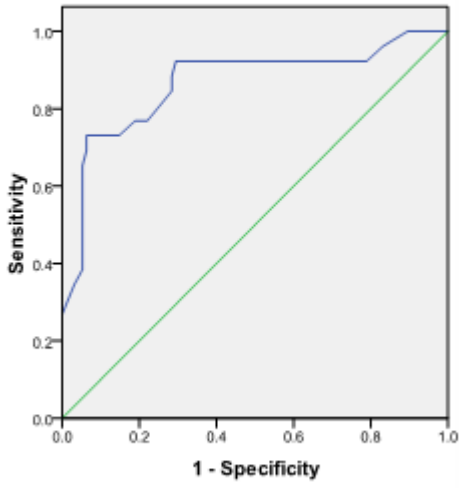


Fig 5.70: ROC Curve for SubSonic

5.6 Multilayer Perceptron Analysis

The result of 10-cross validation over multilayer perceptron technique over the data is shown below. We have used only 1 hidden layer. There is only 1 output node in output layer whose value greater than a threshold (cutoff point) shows weather the class undergoes change or not.

	Sensitivity	Specificity	Cutoff Point	AUC
AOI	0.830	0.748	0.213000	0.888
CheckStyle	0.531	0.870	0.189500	0.767
FreePlane	0.724	0.807	0.051000	0.790
jKiwi	0.609	0.955	0.738500	0.790
Joda	0.750	0.689	0.103500	0.725
jStock	0.889	0.677	0.424000	0.757
jText	0.900	0.624	0.439000	0.778
LWJGL	1.000	0.885	0.277500	0.896
ModBus	0.588	0.884	0.308000	0.749
openGTS	0.700	0.916	0.507000	0.827
openRocket	0.853	0.857	0.548000	0.847
Quartz	0.800	0.819	0.169000	0.797
Spring	0.650	0.651	0.552000	0.728
SubSonic	0.731	0.947	0.516000	0.852

Table 5.6: 10-cross validation results for Multilayer Perceptron

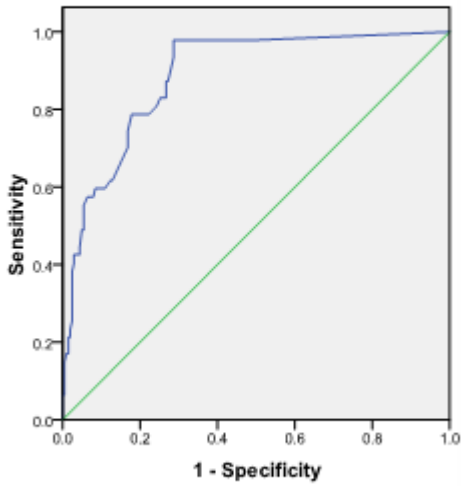


Fig 5.71: ROC Curve for AOI

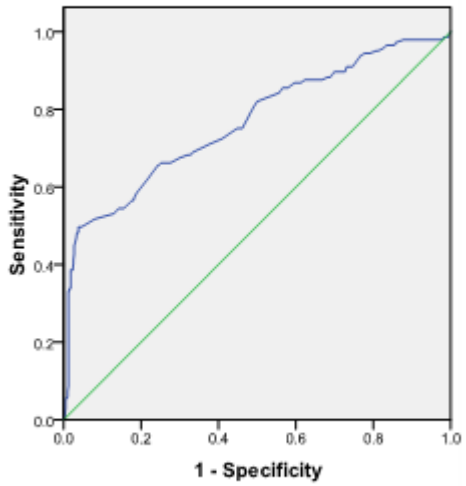


Fig 5.72: ROC Curve for CheckStyle

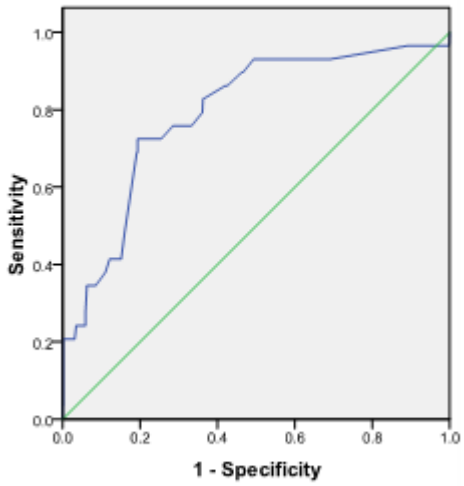


Fig 5.73: ROC Curve for FreePlane

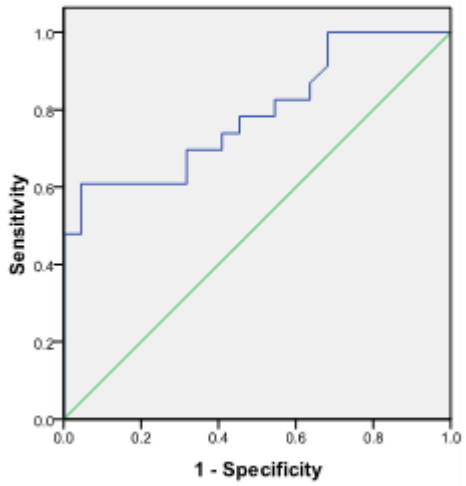


Fig 5.74: ROC Curve for jKiwi

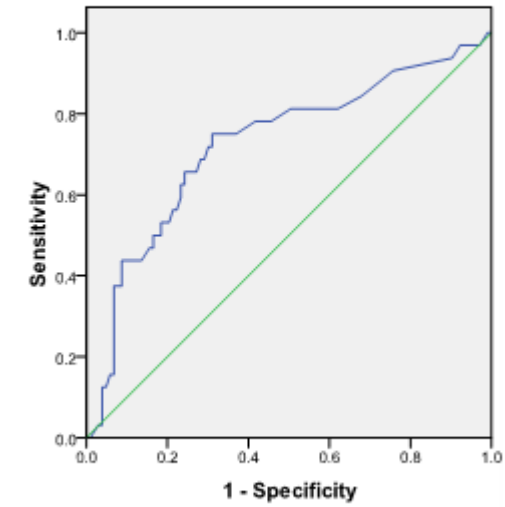


Fig 5.75: ROC Curve for Joda

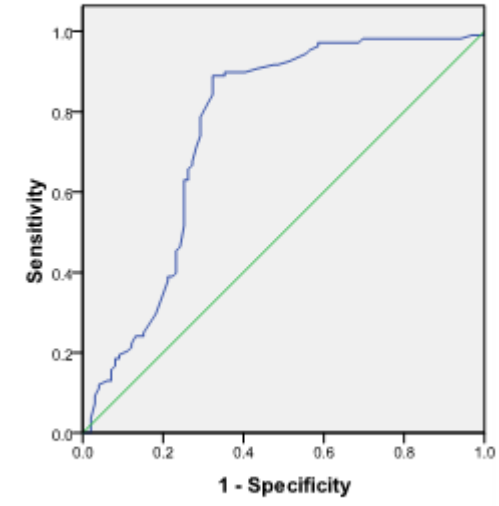


Fig 5.76: ROC Curve for jStock

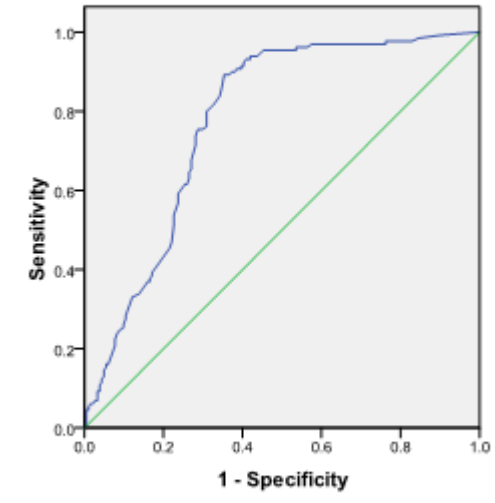


Fig 5.77: ROC Curve for jText

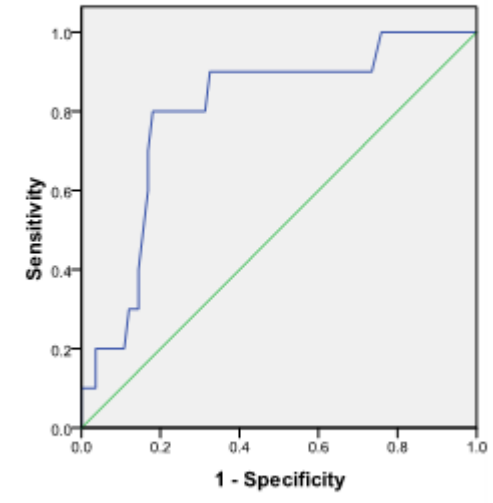


Fig 5.78: ROC Curve for Quartz

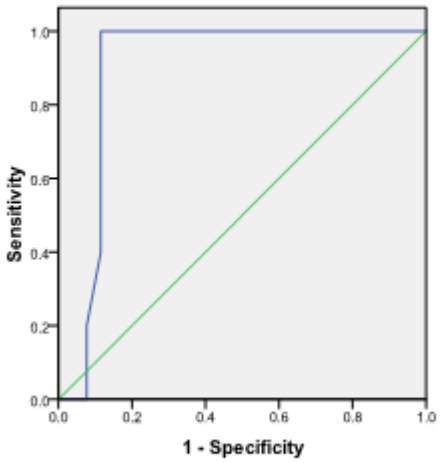


Fig 5.79: ROC Curve for LWJGL

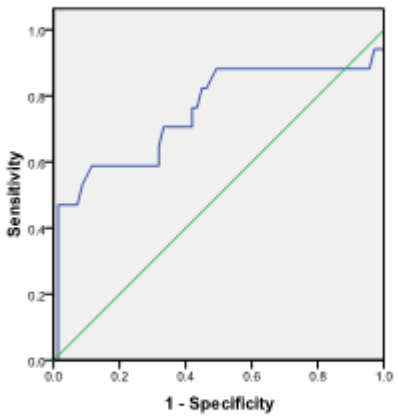


Fig 5.80: ROC Curve for ModBus

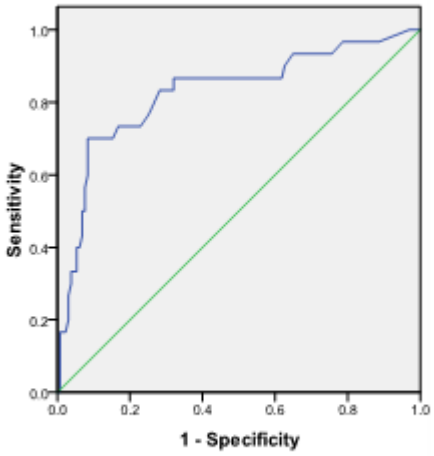


Fig 5.81: ROC Curve for openGTS

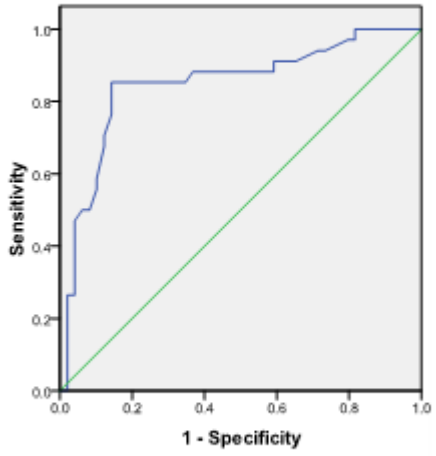


Fig 5.82: ROC Curve for openRocket

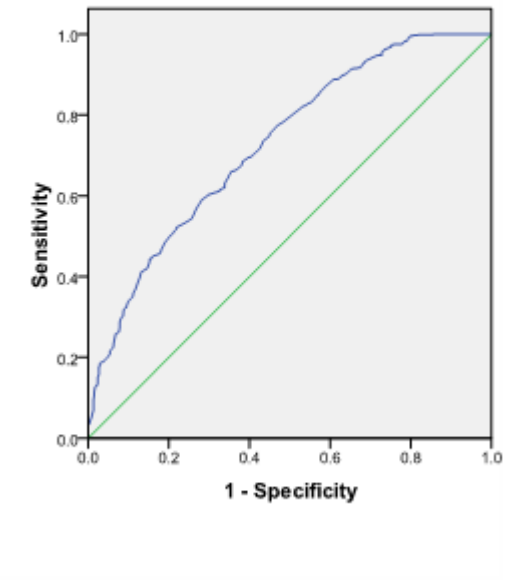


Fig 5.83: ROC Curve for Spring

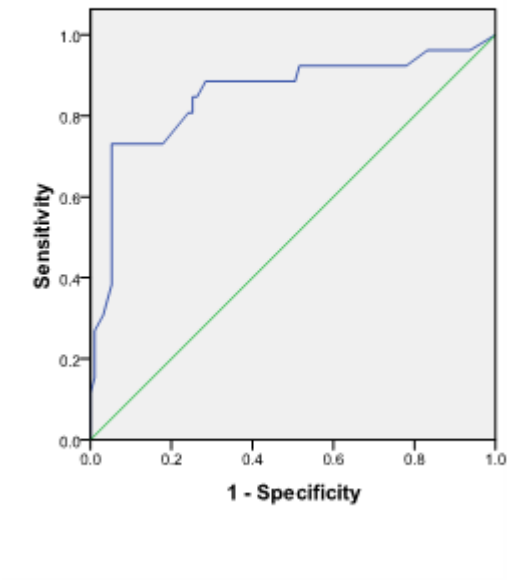


Fig 5.84: ROC Curve for SubSonic

5.7 Model Evaluation

We have not selected any arbitrary cutoff point and to obtain a balance between the number of classes predicted as change prone and not, the cutoff point of the predicton model is determined by ROC analysis. Area under the ROC Curve (AUC) is a combined measure of sensitivity and specificity. Hence, we have used the AUC metric for computing the accuracy of the predicted models. The models are applied on the same dataset from which they are derived using 10-cross validation of all the models.

The AUC of all the models predicted using Multilayer Perceptron technique is greater than the AUC of all the other models predicted using the other machine learning techniques (Naïve Bayes, Random Forest, Bagging, LogitBoost, Decision Table). The details of AUC can be checked from Table and the authenticity of the same can be verified.

Both the sensitivity and specificity should be high to predict good and bad websites. The models predicted with the Naïve Bayes, Random Forest and Multilayer Perceptron techniques have higher accuracy in terms of sensitivity and specificity.

Overall, in terms of sensitivity, specificity and AUC, the best model suitable for predicting weather a class in change prone or not is determined to be Multilayer Perceptron.

5.8 Discussion

For a technique to be effective in making predictions a probability of correct classification should be at least 70%. We use two different measures to evaluate the correctness of a model, i.e., sensitivity and specificity. Sensitivity is the number of correctly classified true instances while specificity is the number of correctly classified false instances. For a model to be effective, both these values should be high. This would mean that the model makes correct classifications for both true and false values. In other words, the model performs well for both true and false values.

From the results it is clear that the Multilayer Perceptron model performs best in comparison to all other models. In our dataset of 14 software systems it exhibited a sensitivity of .70 or more in most cases and specificity of .67 or more. This means that over the dataset of 4120 classes, the multilayer perceptron was able to correctly classify 1272 change prone classes out of 1817 and 1544 classes as not change prone out of 2303. A very encouraging number.

Chapter 6

CONCLUSION & FUTURE WORK

The aim of this study was to determine the prediction power of code smells for class level change proneness. We also constructed prediction models based on machine learning methods for the same.

We started by collecting data in form of classes in open-source software systems. A total of 4120 classes were selected for this study after pre-processing. The data hence obtained is then parsed through Understand in order to estimate metric values for all the 4120 classes. The metric values were then used to find which code smells persisted in which class. This step was followed by estimation of exact change a class went through. This was done by making use of an open-source tool called CLOC. Finally we applied machine learning methods to assess the effect of code smells on class level change proneness.

In the process we developed a tool to pre-process software systems and make them ready for estimation of exact change.

We conclude this study by stating that,

- The use of code smells to predict change proneness for a class is a step in the right direction. The percentage of correct classification for this method is pretty good.
- The Multilayer Perceptron technique provides the best results and prediction power in comparison or other machine learning models.

We also came across some problems that interested people can take up in the future,

1. The threshold determining technique we used is trial and error. This technique is good enough for conducting studies but to make this method more repeatable and usable, we need to establish some kind of mathematical relations to estimate thresholds on the fly.
2. We used only 13 code smells in this study. One of the limitations of our previous study was a small dataset which has been taken care of in this attempt, but the number of code smells proposed till date and the number of smells examined by us has a huge difference.
3. The dataset we considered is based on open-source software where the systems are built by communities. Such prediction methods should be tested over closed source systems developed by professional developers in a real life software engineering situation.

Chapter 7

PUBLICATIONS

7.1 Communicated Papers

The work done by us in this area has been communicated to an International Journal for review.

The details of the paper are provided below,

Journal Name: Software Quality Professional

Web URL: http://asq.org/pub/sqp/

Paper Title: Empirical Validation of Code Smells for Predicting Software Change Proneness

Authors: Ruchika Malhotra, Nakul Pritam

REFERENCES

[1] Foutse Khomh, Massimiliiano Di Penta, Yann-Gaël Guéhéneuc, "An Exploratory Study of the Impact of Code Smells on Software Change Proneness", Proceedings of the 16th Working Conference on Reverse Engineering, IEEE Computer Society, 2009.

[2] Tuming Zhou, Hareton Leung, "Examining the Potentially Confounding Effect of Class Size on the Associations between Object Oriented Metrics and Change-Proneness", IEEE Transactions on Software Engineering, Vol. 35, No. 5, September 2009.

[3] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design, IEEE Transactions on Software Engineering", Vol. 20, No. 6, pp 476-493, June 1994.

[4] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F Le Meur, "DECOR: A method for the specification and detection of code and design smells", IEEE Transactions on Software Engineering, vol. 36, no. 1, January 2010.

[5] M. Fowler, Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley, June 1999.

[6] Li. W. and Henry, S., "Object Oriented Metrics that Predict Maintainability", J. Systems and Software, Vol. 23, pp. 111-122, 1993.

[7] Abreu, F. B., Miguel Coulao, and Rita Esteves, “Towards the design quality evaluation of object-oriented software systems”. In Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, October 1995.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1st edition, 1994.

[9] B. F. Webster. Pitfalls of Object Oriented Development. M & T Books, 1st edition, February 1995.

[10] M. Fowler. Refactoring – Improving the Design of Existing Code. Addison-Wesley, 1st edition, June 1999.

[11] M. Mantyla. Bad Smells in Software - a Taxonomy and an Empirical Study. PhD thesis, Helsinki University of Technology, 2003.

[12] W. C.Wake. Refactoring Workbook. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] W. J. Brown, R. C.Malveau,W. H. Brown, H.W.McCormick III, and T. J. Mowbray. Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1st edition, March 1998.

[14] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In Proceedings of the

14th Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 47–56. ACM Press, 1999.

[15] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the 20th International Conference on Software Maintenance, pages 350–359. IEEE Computer Society Press, 2004.

[16] M. J. Munro. Product metrics for automatic identification of “bad smell” design problems in java source-code. In F. Lanubile and C. Seaman, editors, Proceedings of the 11th International Software Metrics Symposium. IEEE Computer Society Press, September 2005.

[17] E. H. Alikacem and H. Sahraoui. Generic metric extraction framework. In Proceedings of the 16th International Workshop on Software Measurement and Metrik Kongress (IWSM/ MetriKon), pages 383–390, 2006.

[18] K. Dhambri, H. Sahraoui, and P. Poulin. Visual detection of design anomalies. In Proceedings of the 12th European Conference on Software Maintenance and Reengineering, Tampere, Finland, pages 279–283. IEEE CS, April 2008.

[19] F. Simon, F. Steinbr¨uckner, and C. Lewerentz. Metrics based refactoring. In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR’01), page 30, Washington, DC, USA, 2001. IEEE Computer Society.

[20] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization based analysis of quality for large-scale software systems. In proceedings of the 20th international conference on Automated Software Engineering. ACM Press, Nov 2005.

[21] Mika V. Mantyla, Casper Lassenius. Subjective evaluation of software evolvability using code smells: A empirical study. Empirical Software Engineering (2006) 11: 395-431.

[22] Daryl Posnett, Christian Bird, Prem Devanbu. An empirical study on the influence of pattern roles on change proneness. Empirical Software Engineering (2011) 16: 396-423.

[23] Taghi M. Khoshgoftaar, Naeem Seliya. Comparative Assesment of Software Quality Classification Techniques: An Empirical Case Study. Empirical Software Engineering (2004) 9: 229-257.

[24] Du Zhang, Jefferey J.P Tsai. Machine Learning and Software Engineering. Software Quality Journal 11, 87-119, 2003.

[25] Wolfgang Holz, Rahul Premraj, Thomas Zimmermann, Andreas Zeller. Predicting Software Metrics at Design Time. PROFES 2008, LNCS 5089, pp. 34-44, 2008.

[26] Hongmin Lu, Yuming Zhou, Baowen Xu, Hareton Leung, Lin Chen. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. Empirical Software Engineering (2012) 17: 200-242.

[27] Foutse Khomh, Massimiliano Di Penta, Yann-Gael Guehaneuc, Guiliano Antoniol. An exploratory study of the impact of anti-patterns on class change- and fault-proneness. Empirical Software Engineering (2012) 17: 243-275.

[28] http://www.spinellis.gr/sw/ckjm/doc/index.html The Official Documentation of CKJM

[29] Understand Your Code – Official Website http://www.scitools.com

[30] Understand Your Code – List of Complexity Metrics Available http://www.scitools.com/documents/metricsList.php?metricGroup=complex

[31] Understand Your Code – List of Count Metrics Available http://www.scitools.com/documents/metricsList.php?metricGroup=count

[32] Understand Your Code – List of Object Oriented Metrics Available http://www.scitools.com/documents/metricsList.php?metricGroup=oo

[33] M. Stone, "Cross-validatory choice and assessment of statistical predictions," In Journal of the Royal Statistical Society, Series B (Methodological), 36, 111–147, 1974.

[34] Y. Singh, R. Malhotra, and P. Gupta, "Empirical Validation of Web Metrics for Improving the Quality of Web Page," In *International Journal of Advanced Computer Science and Applications (IJASCA),* Vol. 2, No. 5, 2011.

[35] Weka 3: Data Mining Software in Java. Available from http://www.cs.waikato.ac.nz/ml/weka/. Accessed 5 December 2011.

[36] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression: a statistical view of boosting," In *The Annals of Statistics*, Vol. 28, No. 2, 337-407, 2000.

[37] I. H. Witten, E. Frank, and M.A. Hall, "Data Mining: Practical Machine Learning Tools and Techniques," *Morgan Kaufmann, San Francisco,* 3 edition, 2011.

[38] K. P. Murphy, "Naive Bayes Classifiers," *Technical Report,* October 2006.

Appendix A

NO. OF CLASSES IN SYSTEMS

S.No.	Name of System	No. of Common Classes
1	AOI	249
2	CheckStyle	693
3	FreePlane	572
4	jKiwi	45
5	Joda	135
6	jStock	207
7	jText	314
8	LWJGL	31
9	ModBus	86
10	openGTS	161
11	openRocket	83
12	Quartz	93
13	Spring	1333
14	SubSonic	121

Total Classes = 4120*2 = 8240

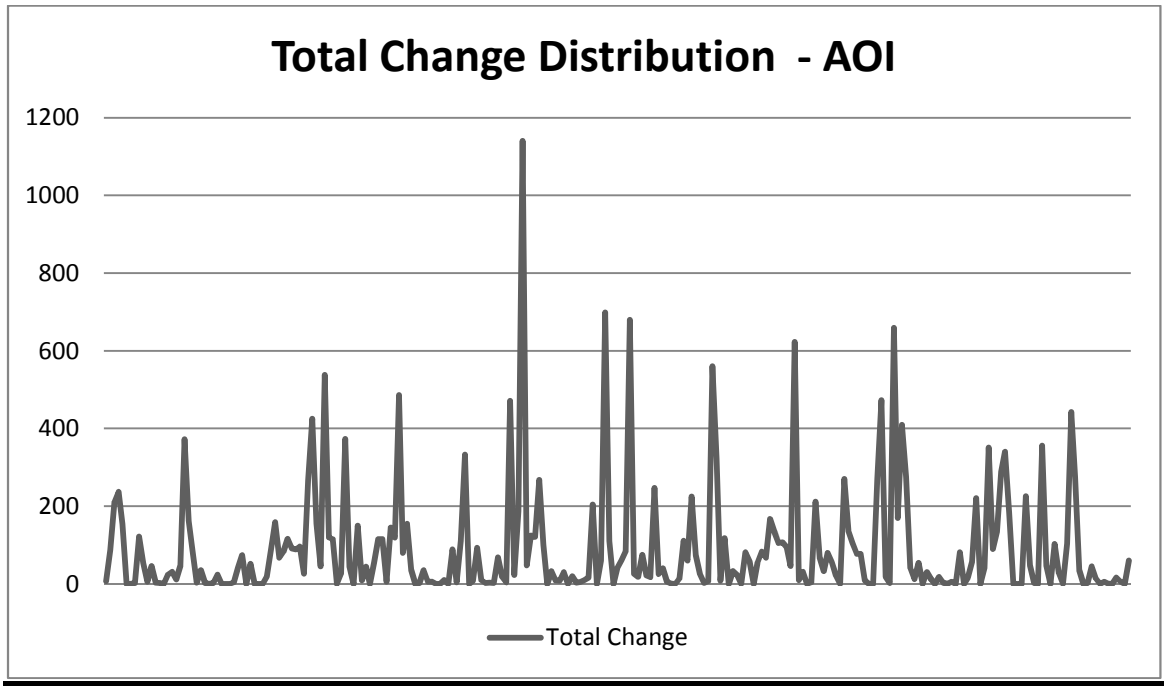
Appendix B

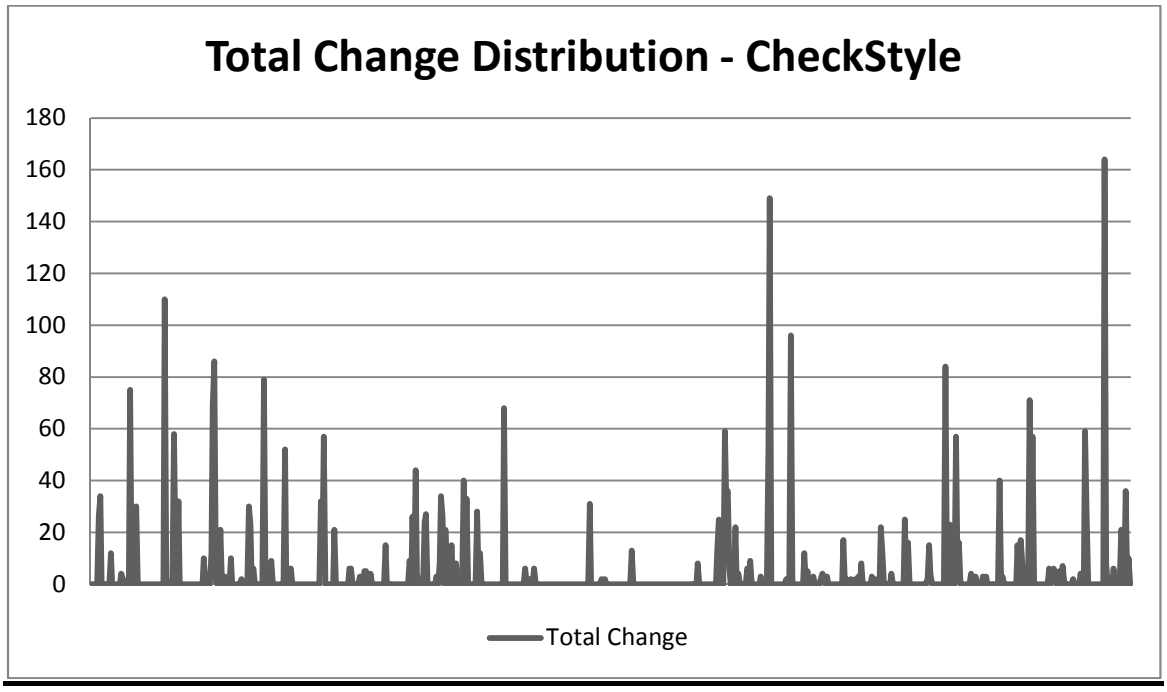
THRESHOLDS FOR METRICS

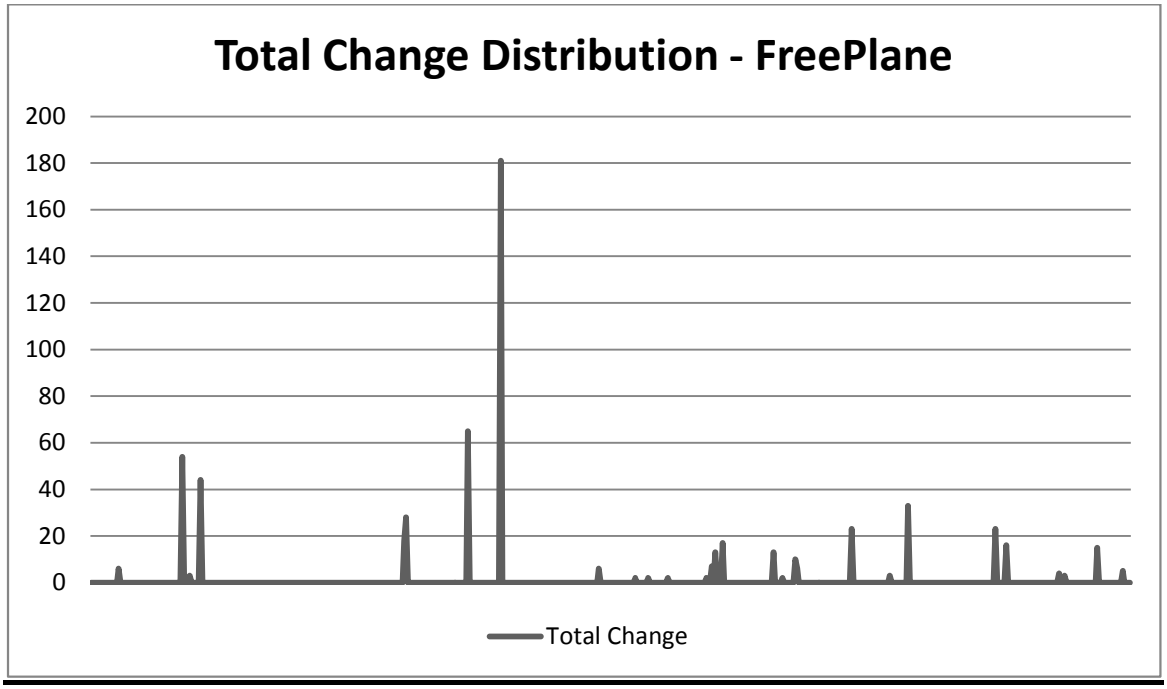
Parameter	Threshold
WMC	4
LCOM	10
Private Variables	3
Public Variables	5
NOM	5
Long Parameter List	4

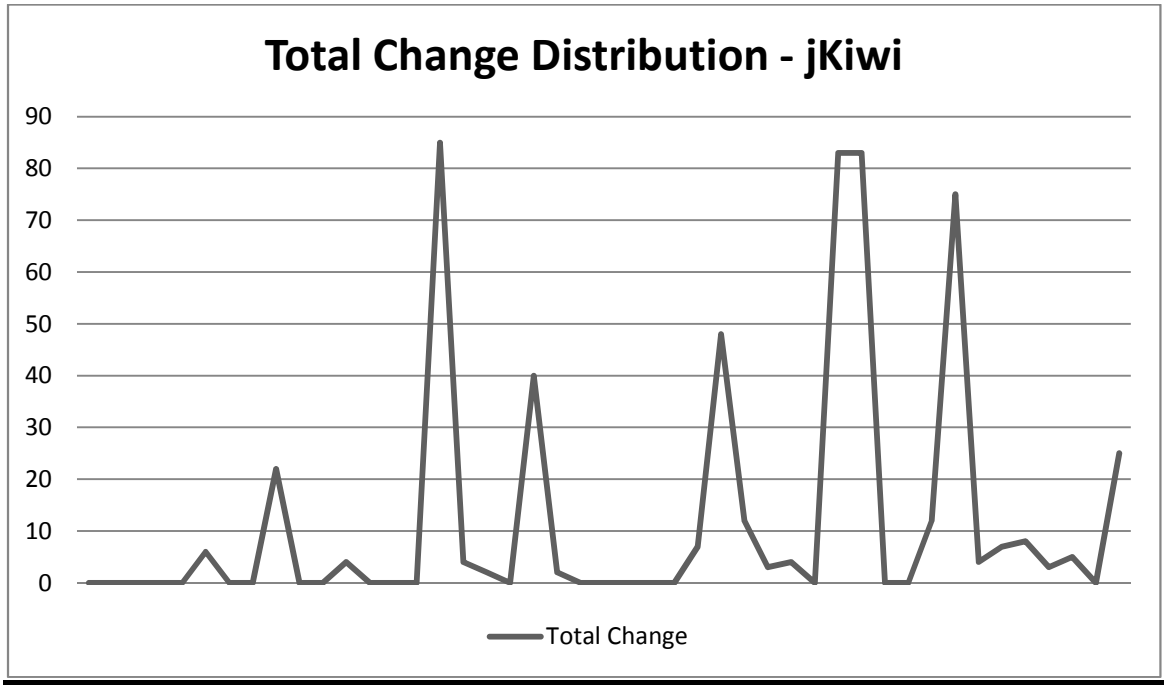
Appendix C

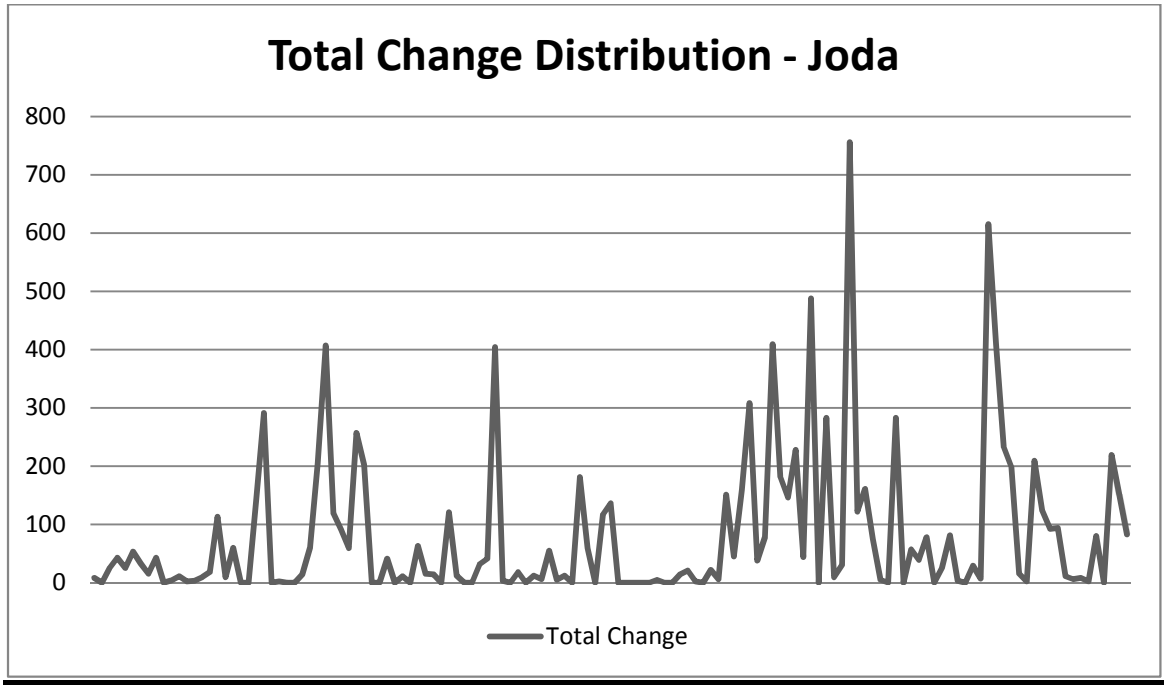
CHANGE DISTRIBUTION

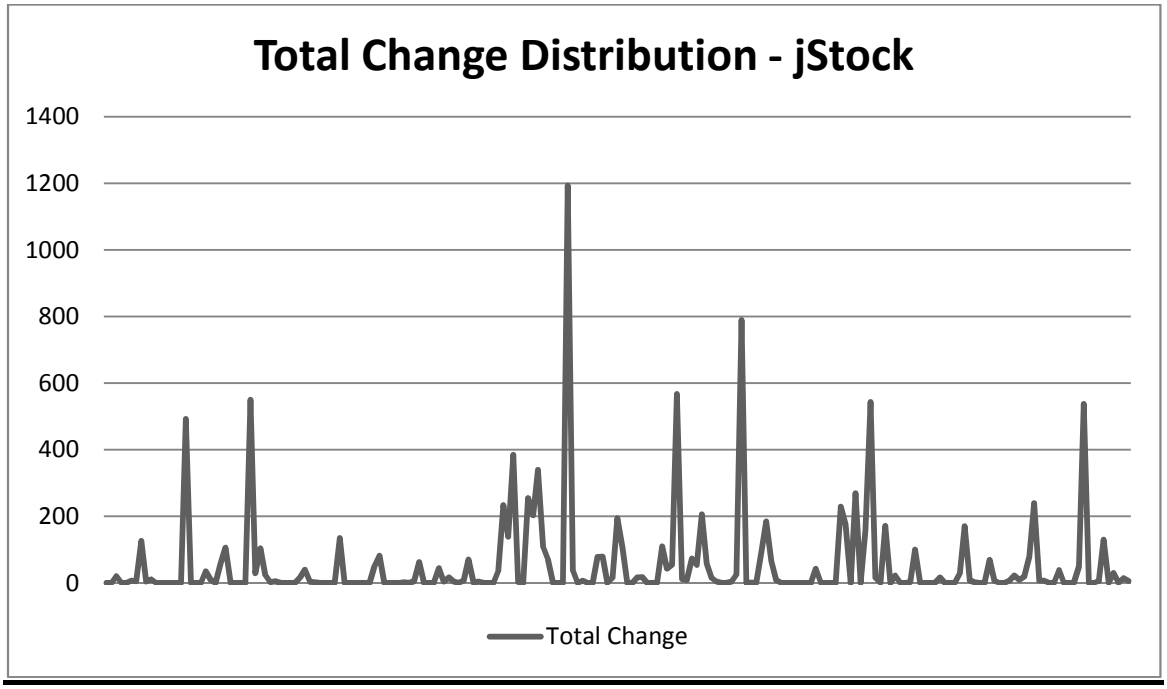


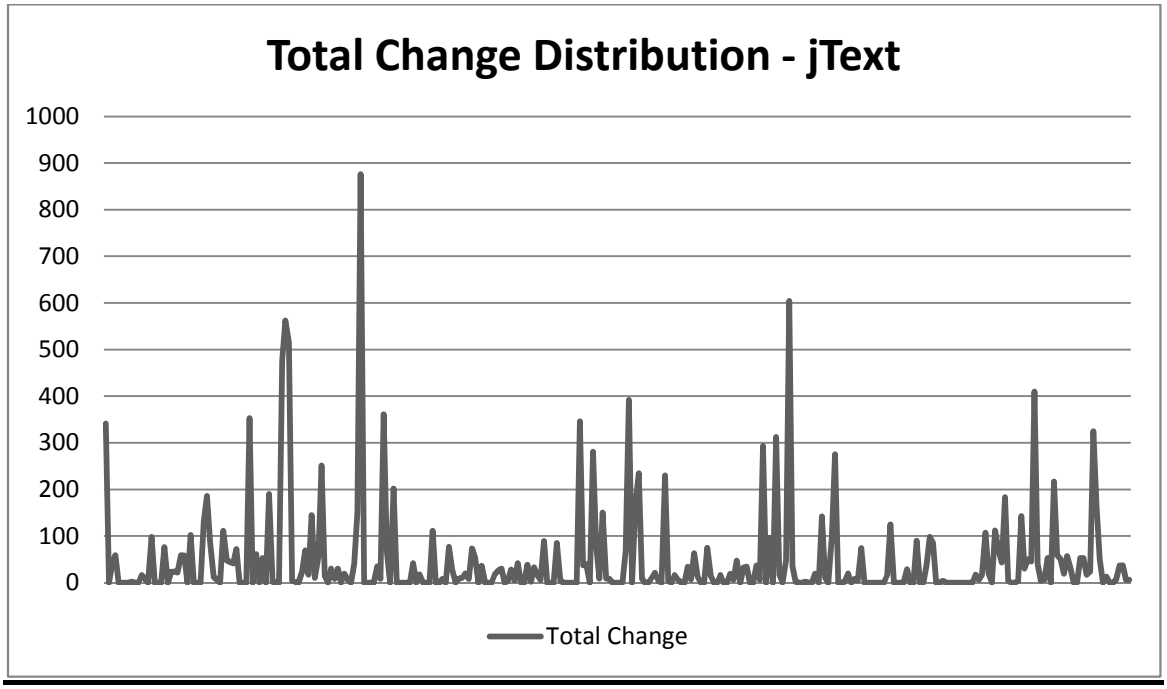


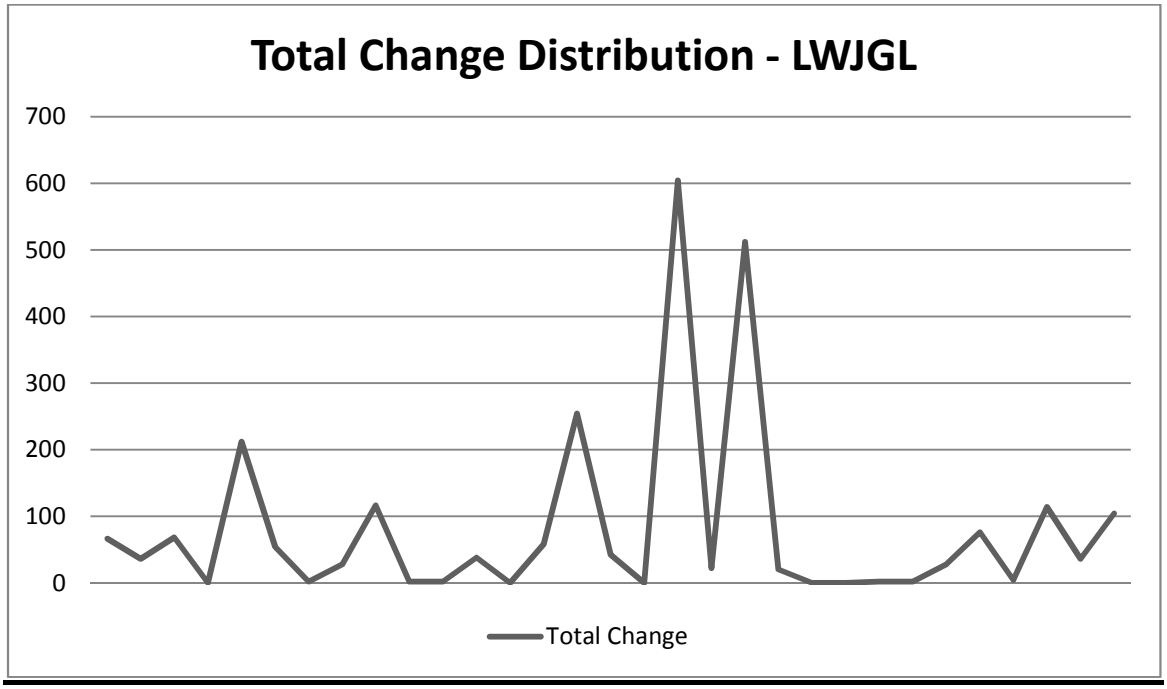

Total Change Distribution - FreePlane
200
180
160
140
120
100
80
60
40
20
0
Total Change

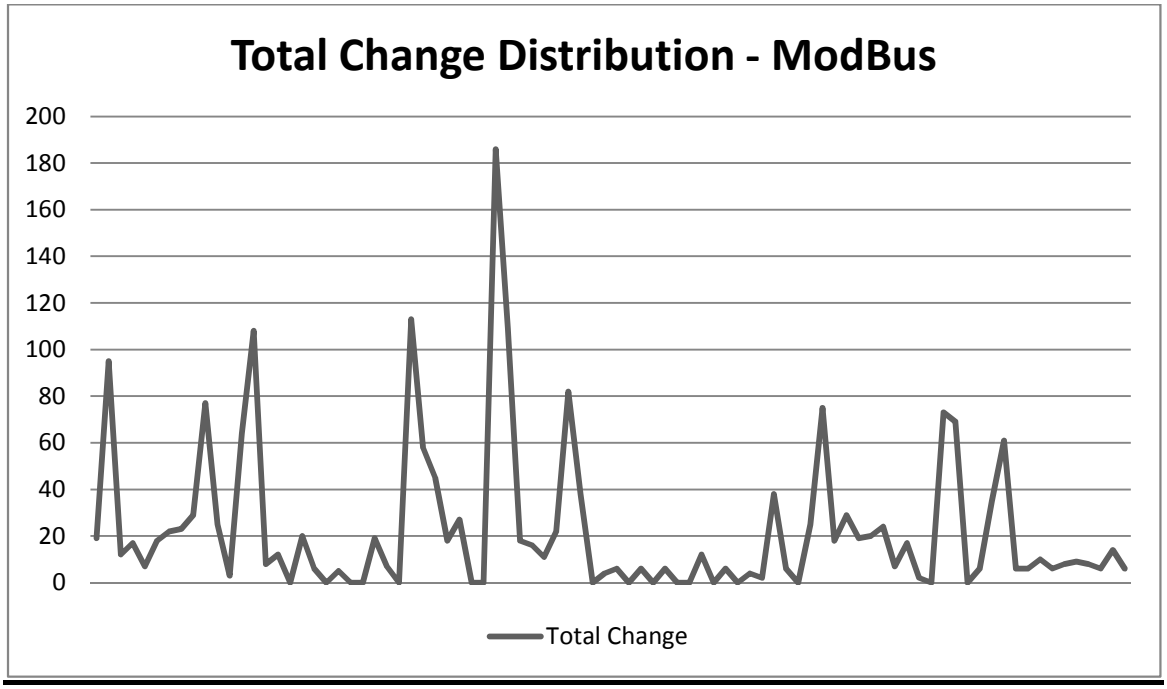

Total Change Distribution - jKiwi
90
80
70
60
50
40
30
20
10
0
Total Change

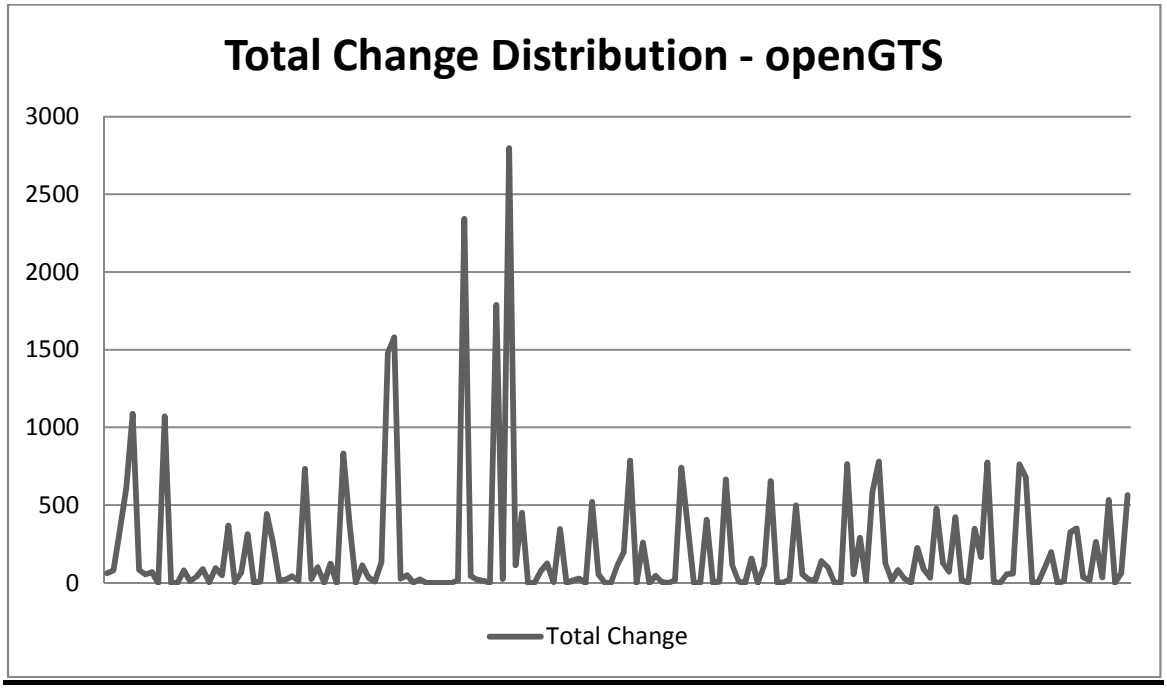

Total Change Distribution - Joda
800
700
600
500
400
300
200
100
0
Total Change

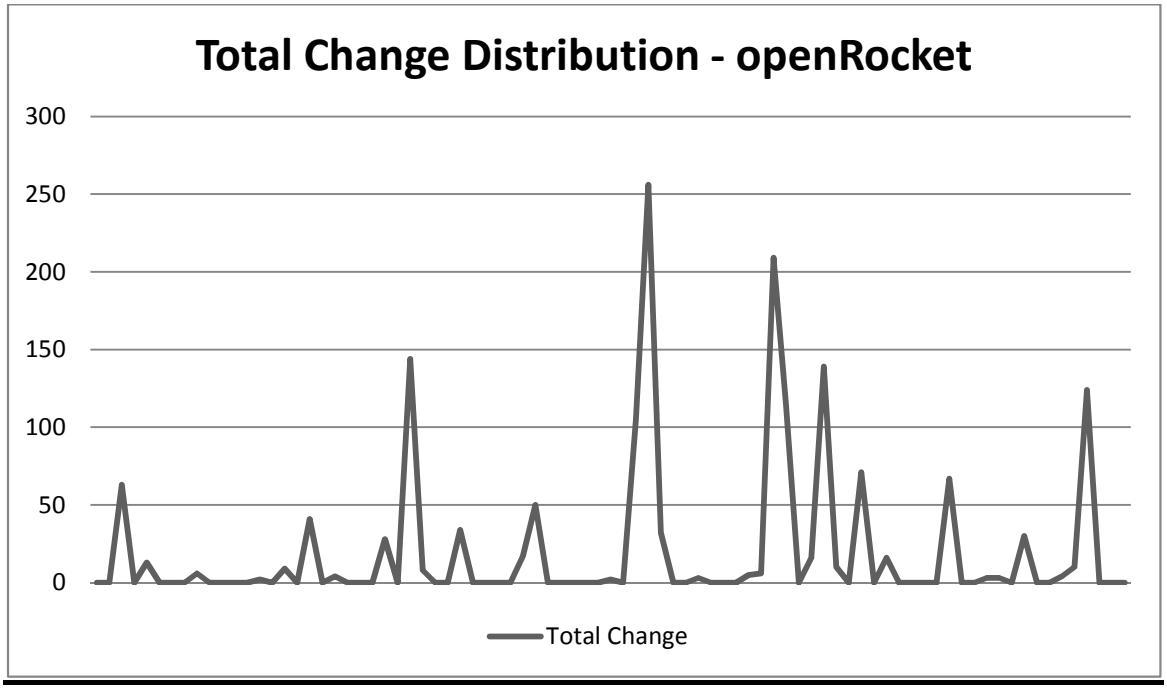

Total Change Distribution - jStock
1400
1200
1000
800
600
400
200
0
Total Change

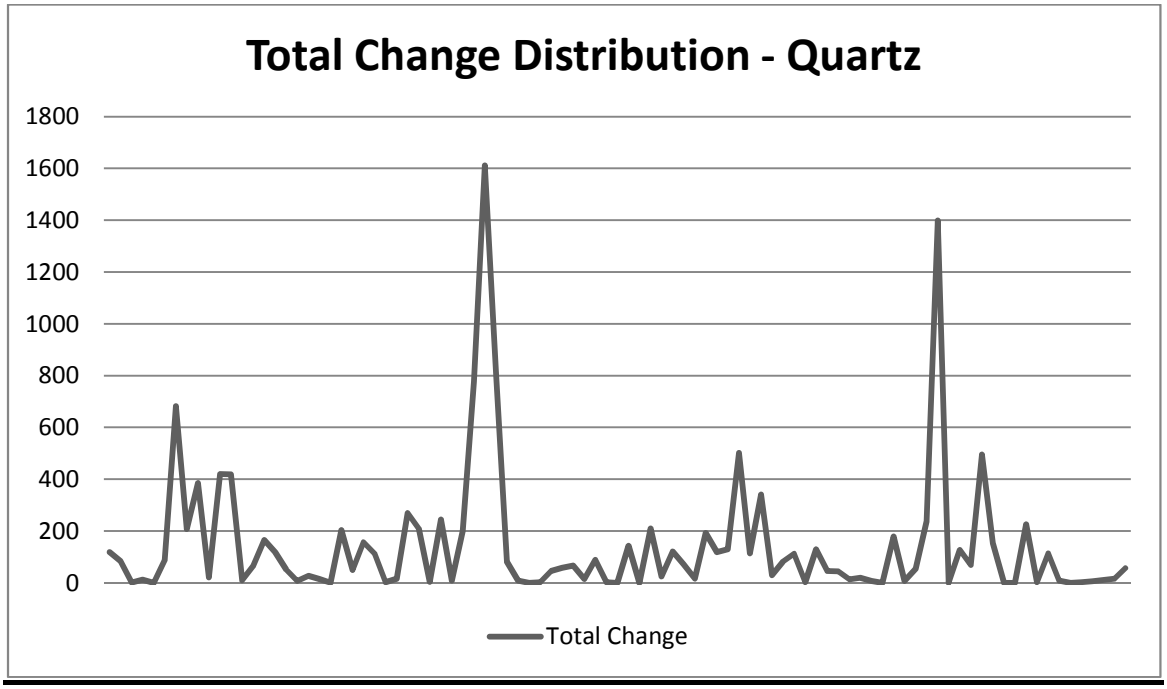

Total Change Distribution - jText
1000
900
800
700
600
500
400
300
200
100
0
Total Change

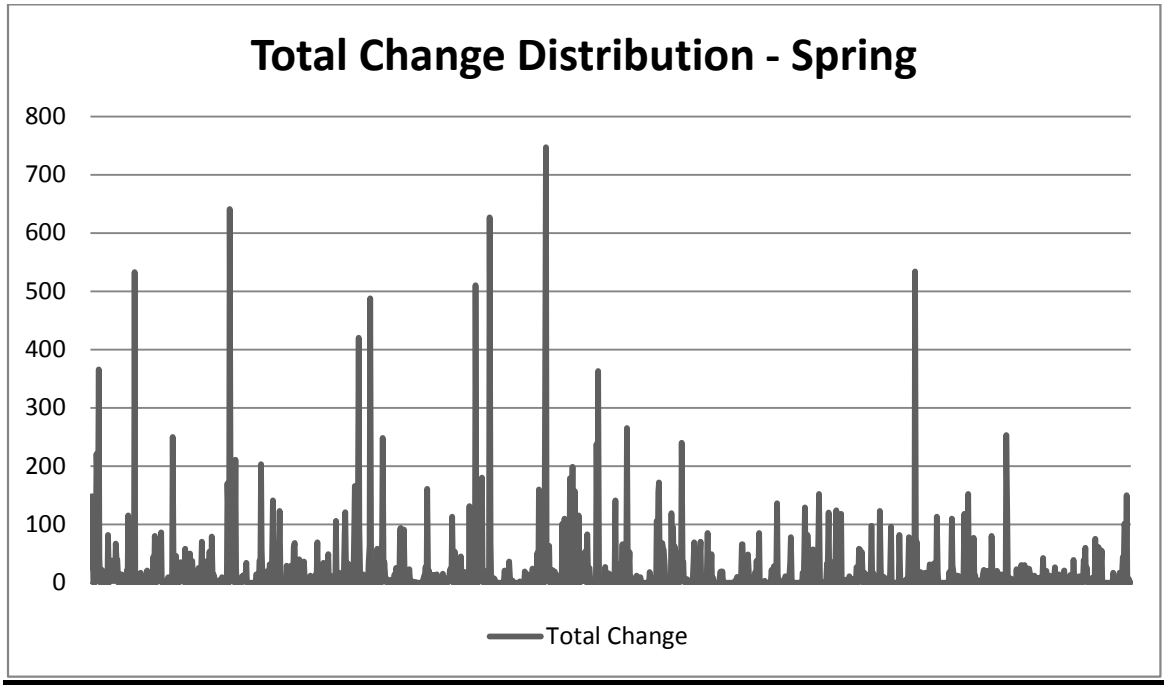

Total Change Distribution - LWJGL
700
600
500
400
300
200
100
0
Total Change

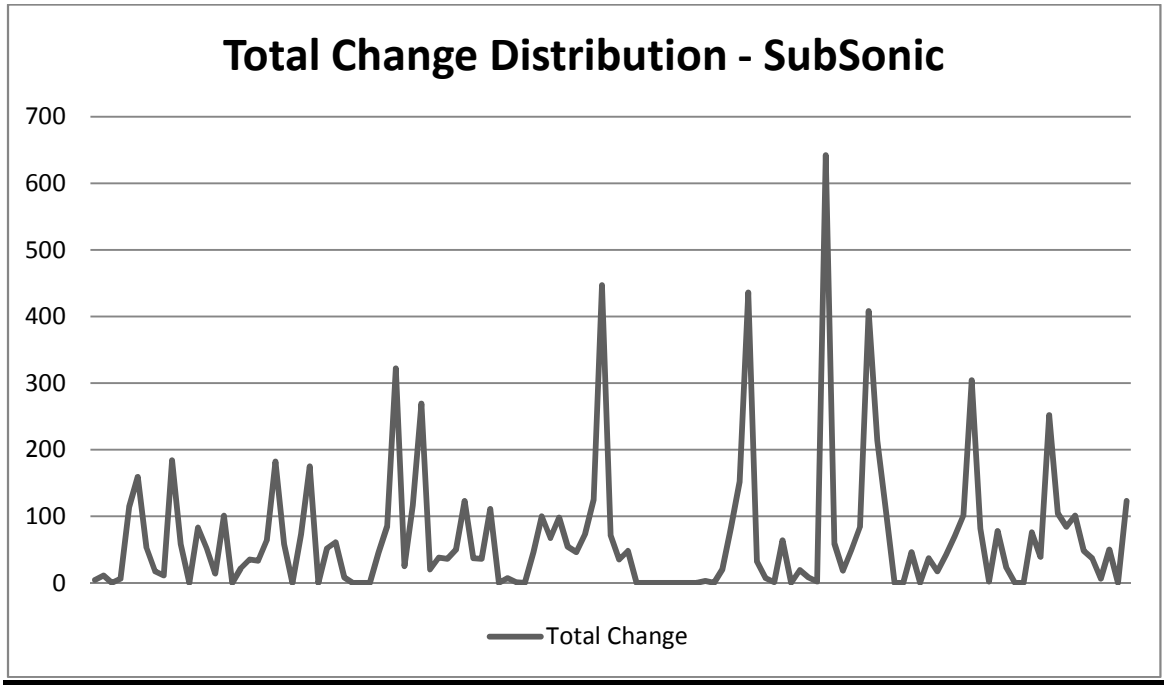

Total Change Distribution - ModBus
200
180
160
140
120
100
80
60
40
20
0
Total Change


Total Change Distribution - openGTS
3000
2500
2000
1500
1000
500
0
Total Change


Total Change Distribution - openRocket
300
250
200
150
100
50
0
Total Change


Total Change Distribution - Quartz
1800
1600
1400
1200
1000
800
600
400
200
0
Total Change


Total Change Distribution - Spring
800
700
600
500
400
300
200
100
0
Total Change


Total Change Distribution - SubSonic
700
600
500
400
300
200
100
0
Total Change

Appendix C

Communicated Research Paper

Assessment of Code Smells for Predicting Class Change Proneness

Ruchika Malhotra, Nakul Pritam

Department of Software Engineering, Delhi Technological University, Delhi

ruchikamalhotra2004@yahoo.com, nakul.pritam@gmail.com

***Abstract*- Poor design choices called anti-patterns manifest themselves in the source code as code smells. Code smell is a synonym for bad implementation and is assumed to make maintenance tasks difficult to perform. In this study we attempt to empirically validate whether it is possible to determine the degree of change-proneness for a class on the basis of certain code smells in an object-oriented system. In this study, we develop a tool to detect the presence of 13 different code smells in a Java class using thresholds. The data used for assessment is source code of Quartz, an open source job scheduler, from two versions 1.5.2 and 1.6.6. A total of 79 classes are examined in this study. The results suggest a clear relationship between code smells and change proneness of a class.**

***Keyword*- Code Smells, Change Proneness, Software Maintenance**

1. INTRODUCTION

As the complexity of software being developed is increasing so is the cost of maintaining it. With so many factors like time to market, budget and shortage of skilled labour limiting the development process it has become very difficult to built quality software. However, maintenance is something that every developer has to worry about because it is not possible to satisfy future requirements or to test the system inside out. Since it is not possible to predict the changes that a particular software system may invite, the maintenance costs and manpower for a software system remain a mystery till the time they realize themselves.

Recently, a quality factor called change proneness has emerged and is used to quantify the amount of change a particular software system has undergone over two successive releases. The quantification is done at class level so the exact change a class went through over successive releases can be computed.

Researchers over the globe have tried to club change proneness to various other attributes like design patterns, code smells and metrics. But the results however are still in the experiment phase. The most unpredictable component about change is that it is very much tied to software design and since software design is more of an art, it is not possible to confine it in theories. The theories we have till date usually aim at suggesting best practices rather than specifying exactly how a design must be made.

There has been tremendous success in estimation of some parameters of software as early as in the design stage using patterns and metrics. These measures allow us to quantify certain aspects of the software and predict things that otherwise cannot be predicted till the implementation phase. This includes the work done by [3], [6] and [7] in presenting metric suites, each having its own domain of application and speciality.

The motivation for this work comes from [1], where the authors have established a link between change proneness of a class and some code smells it carries. They tried out multiple versions of Eclipse and Azures and found out a relationship between code smells and change proneness.

In this study we empirically validate the ability of code smells to predict the degree of change proneness a class exhibits. We use a data set containing 79 classes obtained from an open-source task scheduler called Quartz as the subject of our study. We find the error in classifications that occur in predictions of change proneness made using code smells. Two versions of the system are used for this study, 1.5.2 and 1.6.6. We use the study conducted in [1] as the foundation and assess the presence of 13 code smells in each class in order to predict the degree of change proneness.

The paper is organized as follows: Section 2 summarizes the work done in the field of metrics and quality. Section 3 explains the research background. Section 4 presents the methodology followed in this study. The analysis of results is discussed in Section 5. Finally, in Section 6 we summarize the conclusion and the future work that can be done in this area.

2. Related Work

A lot of existing work has focussed on detection of smells. Moha et al. proposed DECOR[4] for specification and detection of smells, [1] has established a relationship between code-smells in a class and the probability of it undergoing change in subsequent releases (change-proneness of a class).

Theoretically, code smells [5] are the manifestation of bad implementation choices on the source code. Practically, they occur between design and implementation phases but their effects are visible most strongly in the source code.

The structure of a class can be analysed by studying the metric values it produces. The earliest and most fundamentally strong metric suite was proposed by [3] with a total of 6 metrics. This was followed by [6] where a set of metrics that predict maintainability were proposed. In [7] the author proposed a set of design quality metrics which can be used directly at system level. Hence, a large number of metrics has been proposed till date and each set has its own importance. The class-level metrics are used to indicate the internal properties of a class (LOC, NOM, etc) and the association between classes (CBO, NOC, etc). We use C & K metrics [3] because of their simplicity and the broad range of coverage they have on the entire software system with inter-class and intra-class measures.

Code smells [5] are bad implementation choices. Mostly, the roots of a code smell lie in the design phase but only in the implementation do they manifest themselves completely. Good implementation choices are called design patterns [8] while bad choices are called anti-patterns. The first description of anti-patterns was given by [9]. In [10], Fowler defined 22 code smells and suggests the areas where refactoring should be applied. And [11], [12] and [13] all define different classifications of smells and anti-patterns.

Moha et al. have presented a method called DECOR in [4] which specifies and detects code smells. Many other techniques exist for this purpose, ranging from manual approaches [14], to heuristic based [15] and [16] and many others [17], [18], [19] and [20].

Change proneness of a class is the odds of it undergoing change in the subsequent version. Changes in a class can occur due to multiple reasons like change in requirements, adaptive maintenance, corrective maintenance, detected or undetected faults, performance enhancement, etc. Usually change in a class in measured manually by comparing two versions of the software but [1] has conducted an exploratory study and linked class change proneness to certain code smells. The results of [1] provide a good foundation to explore further in the direction.

3. Research Background

3.1 Code Smells Selected For Study

Table 1 summarizes the code smells selected to conduct this study. We selected this set of smells because in [1] all of the smells mentioned below affected the change proneness of at least one of the software systems considered by good measure.

Smell	Criteria for Presence
ClassOneMethod	A class with one method only
ChildClass	A class which declares large number of attributes & methods
FieldPrivate	A class which declares large number of private fields

FieldPublic	A class which declares public attributes
HasChildren	A class which has a large number of children
LargeClass	A class with large measure of LOC
LongMethod	A class which declares method(s) with large measure of LOC
LowCohesion	A class which lacks cohesion
LongParameterListClass	A class which declares method(s) which take large number of attributes
ComplexClassOnly	A class which declares method(s) highly complex methods
MethodNoParameter	A class which declares method(s) which take no parameter
MultipleInterface	A class which implements large number of interfaces
NotComplex	A class which is not doing much.

TABLE 5: CODE SMELLS SELECTED FOR STUDY

We have developed a CodeSniffer in JAVA to examine JAVA Classes for presence of the above code smells. The tool analyses two sets of data for that class, these are discussed below.

3.2 Object-Oriented Metrics Selected for Study

Table 2 summarizes the object oriented metrics used in this study. We use C & K [3] metrics for estimation of some of the code smells used in this study. The reason for using C & K metrics is availability of open-source tools to calculate them for certain software systems.

We use CJKM [21] in the backend of our tool to estimate metric values for Java classes and use the following metrics for our study,

Metric	Description
WMC	Estimate of total class complexity
DIT	Estimates the maximum depth of inheritance
NOC	Number of immediate subclasses
CBO	Number of classes the given class is coupled to
RFC	Number of methods a particular class can call
LCOM	Estimates the degree of cohesion for a class

TABLE 6: OBJECT ORIENTED METRICS SELECTED FOR STUDY

3.3 Source Parameters Selected for Study

Apart from using C & K [3] metrics, we use some parameters derived directly from Java source files to calculate some of the code smells and at the same time improve our estimation of smells derived from C & K metrics. Table 3 summarizes the 8 parameters used in this study,

Parameters	Description
Number of Private Fields	The count of total private fields declared in a class
Number of Public Fields	The count of total public fields declared in a class
LOC for Class	Lines of Code for a class
Number of Interfaces Implemented	This count estimates the number of interfaces implemented by a class
Number of Long Methods	The count of the number of long methods present in a class.
Number of Methods Implemented	The number of methods declared in a class.
Number of Methods without Parameters	The number of methods that take no parameters.
Number of Methods with Long Parameter List	The total number of methods in a class which take more than 4 parameters.

TABLE 7: SOURCE PARAMETERS SELECTED FOR STUDY

3.4 Empirical Data Collection

Quartz is an open-source job scheduler that can be used to develop schedules for simple to complex industrial problems. The tasks in Quartz are defined as Java components and can be executed virtually. The reason for choosing Quartz is its size. We downloaded two subsequent stable releases (1.5.2 and 1.6.6) of Quartz which contain more than 120 source files each. This allows us to ensure that the data collected for this study is as accurate as possible.

Since Quartz is average sized software we can manually pre-processed the source files. Pre-processing included removing all Java files in either version that are not present in the other version. Quartz 1.5.2 contained 132 Java files while Quartz 1.6.6 contained 183 Java files. Out of these, 129 Java files were present in both versions. Since a lot of files are also dedicated to UI implementations are redundant we remove them too. Finally we are left with exactly 79 Java files which we are going to process further. After selecting the 79 classes, we calculate the amount of change in each class.

The data collection for this study is twofold. The first step is to estimate the exact change the classes in 1.5.2 undergo when upgraded to 1.6.6. This is done by using an open source tool called CLOC. The amount of change is accurately calculated using the following rules,

1. If a line is added or deleted it is counted as one change.
2. If a line is modified, it is counted as two changes.

Once we have the LOC change for each class we normalize the results by dividing them by the LOC count for the class in the prior version. By normalizing the results we ensure that we get a factor similar to percentage change. Finally, we select a threshold and classify the change in classes as either HIGH or LOW. Figure 1 shows the percent distribution of HIGH and LOW change prone classes.

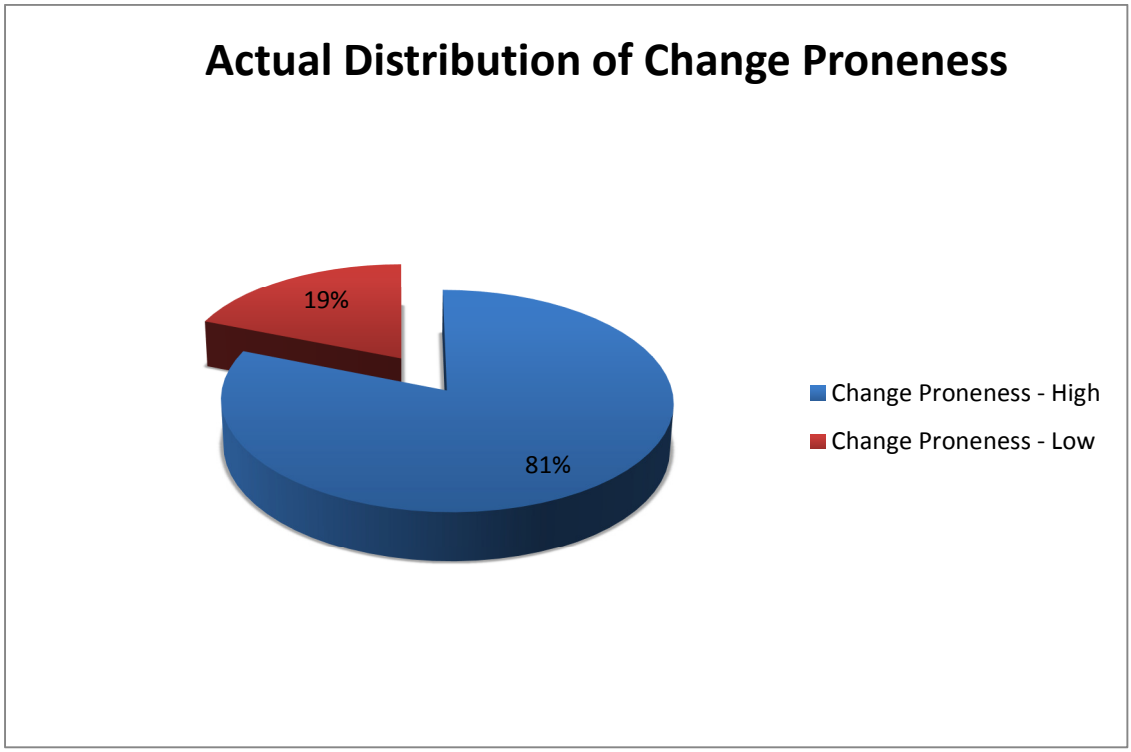


FIGURE 1: PERCENT DISTRIBUTION OF HIGHLY CHANGE PRONE CLASSES AND LOW CHANGE PRONE CLASSES.

The second set of data we collect is of the predicted change by using code smells. To get this data we first pass the Java files of the prior version (1.5.2) through the tool we developed and note down the smells present in each class. A very crucial step in this process is setting of thresholds for various metrics in order to enable the algorithm to detect the presence of a particular smell in the class. In this study we set the thresholds manually. Table 4 show the different thresholds used in this study for various metrics and their values. Selection of thresholds can impact the study deeply so selection has been done without any extravagant assumptions.

Parameter	Threshold
WMC	4
LCOM	10
Private Variables	3

Public Variables	5
NOM	5
Long Parameter List	4
LOC	50

TABLE 4: VARIOUS THRESHOLDS USED IN PREDICTION PROCESS.

4. RESEARCH METHODOLOGY

We employ object oriented metrics and source parameters to examine software systems for code smells and to predict the category of a class as highly change prone or low change prone. Figure 2 shows the flowchart of methodology used in this study.

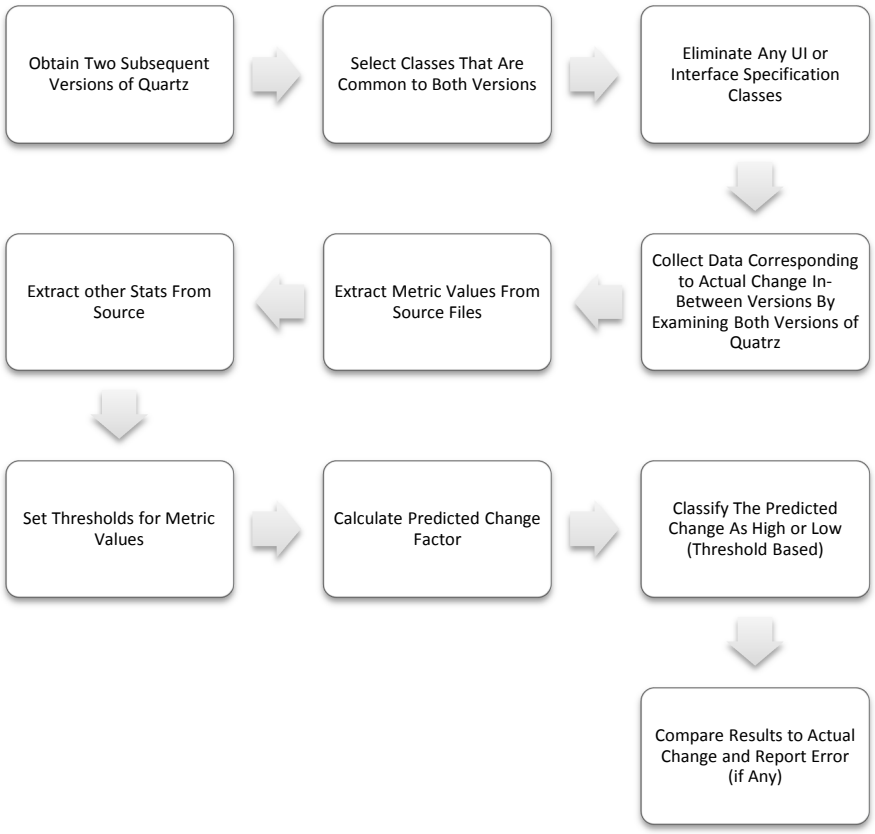


FIGURE 2: FLOWCHART OF METHODOLOGY

4.1 Description of Tool

We have developed a tool to analyse Java source code for presence of 13 code smells. The tool is designed in a way that the users have the freedom to set their own thresholds for the metrics and statistics used for estimation of smells. The tool takes as input the raw .java files as well as .class files. Object oriented metric values are obtained by using CKJM [21] in the background while other estimates are made directly from the source code. Below shown is a snapshot of the interface of the tool,

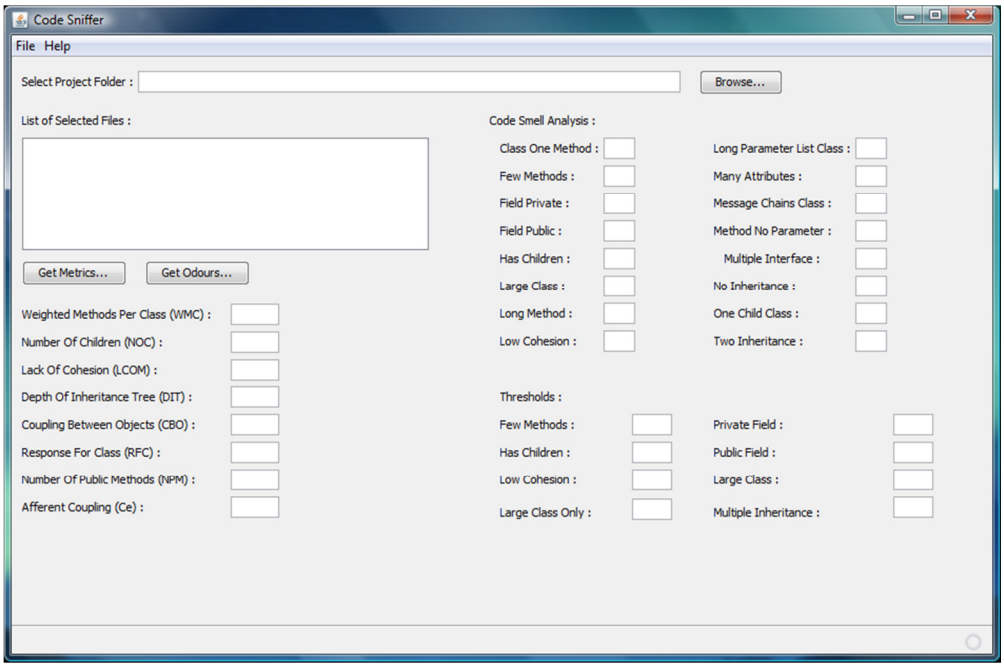


FIGURE 1: SNAPSHOT OF CODESNIFFER

The tool is organized into four modules, these are,

1) Module 1 - CKM_Loader

The CKM_Loader module interacts with the .CLASS files to software to calculate the C & K metrics array. According to [3], the C & K metrics are one of the most accurate identifiers of certain code smells like ComplexClass and NoInheritance, etc.

The CKM_Loader uses a very popular and efficient tool called CKJM running at the backend to estimate the metrics. The result of processing the .CLASS files with CKM_Loader generates two arrays.

a. A 2D matrix containing the actual metrics.
b. A class name vector.

Working of The Module – The module uses the standard CKJM command to run CKJM on the system console and retrieves the output into a buffer. This output is then processed in the following manner to construct the two arrays.

2) Module 2 - DataProcessor

The DataProcessor module pre-processes the source files to make it fit for the SourceStatCalculator module to work upon.

This module conducts the following pre-processing on the source,

a. Remove all standard C Style comments.
b. Tokenize the sources and organize separate classes as separate arrays.
c. Organize different symbols like), (, int, char, float, for, while, if, else and ; as separate tokens as those are very important in measuring statistics.

3) Module 3 - SourceStatCalculator

This module uses the pre-processed source code tokens to calculate statistics that are used in parallel to C & K metrics for estimating the presence of code smells in the class.

There are a total of eight statistic values that are needed for further estimation. These are,

a. Number of Private Fields
b. Number of Public Fields
c. LOC for Class
d. Number of Interfaces Implemented
e. Number of Long Methods
f. Number of Methods Implemented
g. Number of Methods Without Parameters
h. Number of Methods With Long Parameter List

4) Module 4 - SmellCalculator

The smell calculator takes uses the source statistics and the C & K metrics to estimate weather any of the 17 code smells is present in the class under consideration.

The module takes as input,

a. The C & K metrics array.
b. The SourceStat array
c. The source file name vector.
d. The class name vector.

The output of this module is a 17 element vector containing the number of classes carrying the corresponding code smell.

4.2 Comparison of Data

The two sets of data we obtained above are compared to each other class by class and a count of correct and incorrect classifications is kept. We use SPSS to conduct this analysis and determine the probability of correct classification and error in classification.

5. ANALYSIS RESULTS

In this section, we analyse the relationship between code smells and change proneness of a class.

5.1 Sensitivity and Specificity

Sensitivity and specificity are used to predict the correctness of a model. The percentage of classes with HIGH change proneness correctly classified as HIGH is called sensitivity (True Positive Rate i.e. TPR) of the model. The percentage of classes with LOW change proneness correctly classified LOW is called specificity (False Positive Rate i.e. FPR) of the model. Ideally, both the sensitivity and specificity should be high.

5.2 Receiver Operating Characteristic (ROC) analyses

The performance of the outputs of the predicted models is evaluated using ROC analysis. It is an effective method of evaluating the quality or performance of predicted models.

The ROC curve is a plot of sensitivity (y-axis) versus its 1-specificity (x-axis). It is called Relative Operating Characteristic curve, because it is a comparison of two operating characteristics (TPR & FPR). The ROC curve allows us to select cut-off points between 0 and 1, and to calculate sensitivity and specificity at each cut-off point. The optimal cut-off point maximizes both sensitivity and specificity.

The model has sensitivity of 75%, specificity of 73.3%. The results of the study are shown in Table 6 and Table 7. The probability of correct classification is well above 70 % which means the code smell based technique is good at classifying the supplied classes are change prone or not. The following observation is made from the analysis shown in table:

1. Out of 65 classes with HIGH change proneness, 49 are correctly classified, and 16 HIGHLY change prone classes are incorrectly classified.
2. Similarly, out of 14 classes with low change proneness, 10 are classified correctly, while 5 are classified incorrectly.

Parameter	Actual Values	Estimated Values
Total Classes (Total No. of Selected Classes Common to Both Versions)	132(79)	183 (79)
Classes With High Change Proneness	65	53
Classes With Low Change Proneness	14	26

TABLE 6: RESULTS OF ESTIMATION CONTRASTING ACTUAL AND ESTIMATED VALUES OF CHANGE PRONE CLASSES

We noted that the prediction method has a tendency of misclassifying the LOW possibility classes as HIGH. The accuracy for HIGH is much more compared to the accuracy of LOW. This can be explained from the fact that most of the classes in the system exhibited an actual change proneness factor in the higher side and relatively fewer classes were on the LOWER side. The algorithm was able to classify almost all the classes with HIGH probability while the LOW change prone classes were frequently misclassified.

Parameter	Value
No. of Classes With High Change Proneness Classified Correctly	49
No. of Classes With High Change Proneness Classified Incorrectly	16
No. of Classes With Low Change Proneness Classified Correctly	10
No. of Classes With Low Change Proneness Classified Incorrectly	5

TABLE 7: TABLE SPECIFYING DATA FOR SENSITIVITY AND SPECIFICITY OF PREDICTIONS

The analysis shows that this method for predicting change proneness is good enough to be explored further. Below shown is the ROC curve for the analysis,

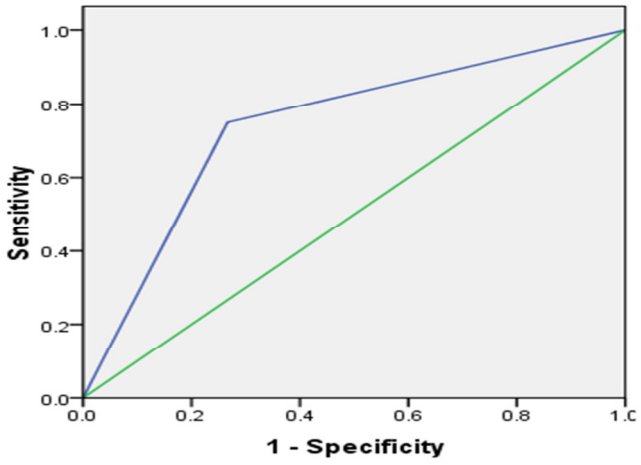


FIGURE 2: ROC CURVE

6. CONCLUSION & FUTURE WORK

We conclude this study by stating that the use of code smells to predict change proneness for a class is a step in the right direction. The percentage of correct classification for this method is pretty good considering it is still in infancy.

We found out some important results and came across problems that interested people can take up in the future,

1. The threshold determining technique we used is trial and error. This technique is good enough for conducting studies but to make this method more repeatable and usable, we need to establish some kind of mathematical relations to estimate thresholds on the fly.
2. We used only 13 code smells in this study over a system with 79 classes to process; this method should be applied over a much larger system, perhaps a system from a completely different domain like embedded or real time to check its applicability across domains.

3. The area of experiment till date has been in open-source where the systems are built by communities. Such methods should be tested over closed source systems by developers.

REFERENCES

[1] Foutse Khomh, Massimiliiano Di Penta, Yann-Gaël Guéhéneuc, "An Exploratory Study of the Impact of Code Smells on Software Change Proneness", Proceedings of the 16th Working Conference on Reverse Engineering, IEEE Computer Society, 2009.
[2] Tuming Zhou, Hareton Leung, "Examining the Potentially Confounding Effect of Class Size on the Associations between Object Oriented Metrics and Change-Proneness", IEEE Transactions on Software Engineering, Vol. 35, No. 5, September 2009.
[3] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design, IEEE Transactions on Software Engineering", Vol. 20, No. 6, pp 476-493, June 1994.
[4] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F Le Meur, "DECOR: A method for the specification and detection of code and design smells", IEEE Transactions on Software Engineering, vol. 36, no. 1, January 2010.
[5] M. Fowler, Refactoring – Improving the Design of Existing Code, 1st ed. Addison-Wesley, June 1999.
[6] Li. W. and Henry, S., "Object Oriented Metrics that Predict Maintainability", J. Systems and Software, Vol. 23, pp. 111-122, 1993.
[7] Abreu, F. B., Miguel Coulao, and Rita Esteves, "Towards the design quality evaluation of object-oriented software systems". In Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, October 1995.
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1st edition, 1994.
[9] B. F. Webster. Pitfalls of Object Oriented Development. M & T Books, 1st edition, February 1995.
[10] M. Fowler. Refactoring – Improving the Design of Existing Code. Addison-Wesley, 1st edition, June 1999.
[11] M. Mantyla. Bad Smells in Software - a Taxonomy and an Empirical Study. PhD thesis, Helsinki University of Technology, 2003.
[12] W. C.Wake. Refactoring Workbook. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
[13] W. J. Brown, R. C.Malveau,W. H. Brown, H.W.McCormick III, and T. J. Mowbray. Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, 1st edition, March 1998.
[14] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 47–56. ACM Press, 1999.
[15] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the 20th International Conference on Software Maintenance, pages 350–359. IEEE Computer Society Press, 2004.
[16] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In F. Lanubile and C. Seaman, editors, Proceedings of the 11th International Software Metrics Symposium. IEEE Computer Society Press, September 2005.
[17] E. H. Alikacem and H. Sahraoui. Generic metric extraction framework. In Proceedings of the 16th International Workshop on Software Measurement and Metrik Kongress (IWSM/
MetriKon), pages 383–390, 2006.
[18] K. Dhambri, H. Sahraoui, and P. Poulin. Visual detection of design anomalies. In Proceedings of the 12th European Conference on Software Maintenance and Reengineering, Tampere, Finland, pages 279–283. IEEE CS, April 2008.
[19] F. Simon, F. Steinbr¨uckner, and C. Lewerentz. Metrics based refactoring. In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01), page 30, Washington, DC, USA, 2001. IEEE Computer Society.
[20] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualizationbased analysis of quality for large-scale software systems. In proceedings of the 20th international conference on Automated Software Engineering. ACM Press, Nov 2005.
[21] http://www.spinellis.gr/sw/ckjm/doc/index.html The Official Documentation of CKJM

ABOUT THE AUTHORS

Ruchika Malhotra. She is an assistant professor at the Department of Software Engineering, Delhi Technological University (formerly known as Delhi College of Engineering), Delhi, India, She was an assistant professor at the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Prior to joining the school, she worked as full-time research scholar and received a doctoral research fellowship from the University School of Information Technology, Guru Gobind Singh Indraprastha Delhi, India. She received her master's and doctorate degree in software engineering from the University School of Information Technology, Guru Gobind Singh Indraprastha University, Delhi, India. Her research interests are in software testing, improving software quality, statistical and adaptive prediction models, software metrics, neural nets modeling, and the definition and validation of software metrics. She has published more for than 50 research papers in international journals and conferences. Malhotra can be contacted by e-mail at ruchikamalhotra2004@yahoo.com.

Nakul Pritam. He received his B.Tech degree in Computer Science & Engineering from Uttar Pradesh Technical University, Lucknow India in 2010. He is now pursuing his M.Tech at Delhi Technological University (formerly Delhi College of Engineering), Delhi, India. His research interests are Software Engineering, Software Metrics and Software Quality Analysis using Metrics. Nakul can be contacted by e-mail at nakul.pritam@gmail.com.