Bug Prediction Model Using Code Smells

A dissertation submitted for the Degree of Master of Computer Science

D.L.G.M.Ubayawardana

University of Colombo School of Computing

2018

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name:

Registration Number:

Index Number:

Signature	Date

This is to certify that this thesis is based on the work of
Mr./Ms.
under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name:

Signature	Date

Abstract

The term ‘Code Smells’ was first coined in the book by Folwer [1]. A code smell is a surface indication that usually corresponds to a deeper problem in the system. These poor design choices have the potential to cause an error or failure in a computer program. The objective of this study is to use ‘Code Smells’ as a candidate metric to build a bug prediction model.

Bug prediction models are often very useful. When bugs of a software can be predicted, the quality assurance teams can identify error prone components in advance and effectively allocate more resources to validate those components thoroughly.

Bug prediction is an active research area in the community and various bug prediction models have been proposed using different metrics such as source code, process, network and code smells etc.

In this study we have built a bug prediction model using both source code metrics and code smell based metrics proposed in the literature. We cannot use code smell based metrics only as a single predictor to predict buggy components of a software. There can be files in the source code which do not contain code smells. Therefore we will not be able to predict bug proneness of such components if we use code smell based metrics only. Therefore we initially built a basic model using source code metrics and then enhanced the basic model by using code smell based metrics. We used Naive Bayes, Random Forest and Logistic Regression as our candidate algorithms to build the model. We have trained our model against multiple versions of thirteen different Java based open source projects. The trained model was used to predict bugs in a particular version of a project and a particular project. We have also trained our model among different projects and trained model was used to predict bugs in an entirely different project.

We were able to demonstrate in this study, that code smell based metrics can significantly improve the accuracy of a bug prediction model when integrated with source code metrics. Random Forest algorithm based model showed higher accuracy within a version, within a project and among projects when compared to other algorithms.

Acknowledgements

First I would like to thank University of Colombo School of Computing for the opportunity given me to study for the Masters of Computer Science program.

Foremost I wish to thank Dr. Damith Karunaratna, my research project supervisor, for making the opportunity to explore my knowledge in a latest technique, and for encouraging me technically and spiritually during my studies. His guidance helped me in all the time of research.

In addition, I wish to thank all other staff members of University of Colombo, specially the Computing Department, for their knowledge which directs us to achieve our goals in life.

I want to express my most profound gratitude to my parents and family for their encouragement and companionship.

Finally, I take this opportunity to express my gratitude to everyone who supported me throughout the course of this MCS project.

Contents

List of Figures

List of Tables

List of Abbreviations

ACC Average Cyclomatic Complexity.

ACM Antipattern Complexity Metric.

ACPD Antipattern Cumulative Pairwise Differences.

AMC Average Method Complexity.

ANA Average Number of Antipatterns.

ARL Antipattern Recurrence Length.

Ca Afferent couplings.

CAM Cohesion Among Class Methods.

CBM Coupling Between Methods.

CBO Coupling Between object classes.

CC Cyclomatic Complexity.

Ce Efferent couplings.

CSV Comma Separated Values.

DAM Data Access Metric.

DIT Depth of Inheritance Tree.

IC Inheritance Coupling.

JIT Just In Time.

LCOM Lack of Cohesion of Methods.

LOC Lines of Code.

MCC Maximum Cyclomatic Complextiy.

MFA Measure of Functional Abstraction.

MOA Measure Of Aggregation.

NOC Number of Children.

NPM Number of Public Methods.

OOP Object Oriented Programming.

RFC Response for a Class.

VCS Version Control Systems.

WMC Weighted Method Count.

Chapter 1

Introduction

1.1 Background

Software systems play a vital role in our daily lives. We use software systems to fulfill most of our day-to-day requirements. We use software systems for finance, transportation, health care, education, communication, and even for entertainment. Since human lives heavily depend on software, correct functioning of software systems is crucial, but software can contain bugs.

1.1.1 What is a bug ?

A software bug is an error or failure in a computer program which causes it to produce an incorrect or unexpected result.

1.1.2 Is it possible to write a bug free software ?

Technically 'yes', but practically not so much [2]. The reason is that it is not economically viable to do so unless what we are building is a life or mission critical application. Even though it is not practically possible to build a completely bug free application, software developers always try to release an application with a minimal number of bugs. Therefore testing must be a mandatory part in the software development life cycle and it should happen throughout the project in order to make sure the business flow is correctly preserved.

1.1.3 What are code smells ?

The term 'Code Smells' is first coined in the book by Fowler [1]. A code smell is a surface indication that usually corresponds to a deeper problem in the system. Code smells do not always indicate a problem. They can be considered as an indicator of a problem rather than the problem themselves.

1.1.4 Different types of code smells

SourceMaking.com [3] describes different code smells that can be found in a computer programs. They divide code smells into five categories. *Bloaters*, *Object-Orientation Abusers*, *Change Preventers*, *Dispensables* and *Couplers*.
Bloaters indicate that the classes and methods in our programs have significant proportions of Lines of Code (LOC). Long Method, Large Class and Long Parameter Lists are some of Bloaters.
Object-Orientation Abusers describe incomplete usage of Object Oriented Programming (OOP) principles.Use of lot of switch statements instead of polymorphism, use of temporary fields and creation of two identical functions in two different classes but with different method names are some of them.
There are situations that when we change something in one place of the code, we have to change several other places too. Change Preventers indicate these situations. Divergent Change and Shotgun Surgery are the most popular code smells under this category.
Dispensables are something which is not necessary and without them our code would be much cleaner, understandable and easier to maintain. Comments in a program is needed, but adding explanatory comments in a program cause the program to smell bad. *'The best comment is a good name for a method or class'* [3]. Creating Duplicated Codes all over the program is also a dispensable. Having classes, methods, parameters or variables in a program which are no longer used (AKA Dead Code) also comes under this category.
Couplers indicate tight coupling between the classes. For example a method accesses data of another object than its own data (Feature Envy).
In addition to the above categories, the code smells can also be classified as *smells within the class* (e.g., as long method, long parameter list and duplicated code) and *smells outside the class* (e.g., as data class, data clumps, refused bequest and etc.,) [4]

1.1.5 When and why code smells are introduced ?

Tufano et al. [5] has conducted a study by evaluating 200 open source projects to identify when and why the code starts to smell bad. They have mined over 0.5 Million commits and manually analyzed 9,164 of them classified as smell-introducing. As per the study most of the smell instances are introduced when files are created. However, there are also cases, especially for Blob and Complex Class, where the smells manifest themselves after several changes performed on the file.
The study further describes smells are generally introduced by developers when enhancing existing features or implementing new ones. As expected, smells are generally introduced in the last month before issuing a deadline, while there is a considerable number of instances introduced in the first year from the project start-up. Finally, developers that introduce smells are generally the owners of the file and they are more prone to introducing smells when they have higher workloads.

1.1.6 Do developers really care about code smells ?

Yamashita et al. [6] has conducted a survey on 85 software professionals and they have found out 32 % of the respondents were not aware of the code smells. The survey has indicated those who do not care at all about code smells are the ones who do not know that much about code smells. According to the survey respondents who are somewhat concerned on code smells have shown the lackness of organizational support, adequate tools and deadlines pressure as the common barriers.

1.1.7 What is a bug prediction model ?

A model which is capable of providing the list of bug-prone software artifacts in advance. A bug prediction model requires historical data such as information retrieved from Version Control Systems (VCS), issue tracking systems and so on. The metrics are derived using these data in order to build and train the model. There have been various techniques suggested in the literature to increase the accuracy of bug prediction models.

1.1.8 The advantages of predicting bugs

The quality of a software has a high impact on its reliability. When bugs of a software can be predicted, the quality assurance teams can identify error prone components in advance and effectively allocate more resources to validate those components thoroughly. Since time and manpower are finite resources it is really advantageous to dedicate more time and more resources to inspect more buggy components.

1.2 The problem

The different bug prediction models have been proposed by the research community based on different metrics in order to identify more error prone components in a software system.

1.2.1 Available bug prediction approaches

The widely used metrics in bug prediction are source code and process metrics.Weighted Method Count (WMC), Depth of Inheritance Tree (DIT), Lines of Code (LOC), Lack of Cohesion of Methods (LCOM) and Cyclomatic Complexity (CC) are some of the measurements used by source code metrics based approaches. Process metrics based approaches use number of newly added lines, modified and deleted code lines and recent activity etc as their measurements.

1.2.2 The problem in traditional bug prediction approaches

Traditional bug prediction approaches based on metrics mentioned above have certain issues. For example it may be sometimes true that a codebase with large LOC is more error prone,

but there is no guarantee that a codebase with a relatively less LOC has less number of bugs. Therefore it is worth to investigate on some other additional metrics to predict bugs.

1.2.3 Motivation

Code smells are supposed to point out bad designs that cause to have code with less maintainability . There is a high chance that something might be wrong in the code when we have lot of bad smells in the code. There is empirical evidence that code smells hinder understandability of code [7], increase change and error proneness [8], [9] and lead to less maintainable code [10].

1.3 Aims and objectives

- Study on different code smells in a computer program.
- Study on available literature/ tools to detect code smells in a computer program.
- Develop a bug prediction model based on traditional source code metrics.
- Enhance the bug prediction model with the help of code smell based metrics.
- Evaluate the accuracy of the two approaches within a single version, within a single project and cross projects.

1.4 Scope and limitations

Scope of this project is to build a bug prediction model which uses metrics derived from code smells. We will use these metrics to enhance the traditional bug prediction approaches.

We will not be focusing on how to detect code smells in a given computer program and publicly available datasets of open source projects will be used for this. We will be analyzing and predicting bugs of software systems developed only in Java language.

1.5 Thesis overview

The remainder of this thesis is as follows: Chapter 2 discusses the background and related work. In chapter 3, we present our research methodology. Chapter 4 discusses proposed solution details. In chapter 5, we present and discuss the results from the study. Finally, we conclude in chapter 6 and discuss directions for future works.

Chapter 2

Literature Review

Bug prediction is one of the most active research areas in software engineering and different prediction techniques have been proposed by the research community. This chapter describes major approaches in software defect prediction.

2.1 Brief history on bug prediction

It was assumed in the early days that complexity of a software could cause defects. Akiyama built a simple model using LOC to represent the how complex a software is [11]. Considering LOC as a metric for bug prediction was too simple and therefore MaCabe proposed cyclomatic complexity as a measure for bug prediction in 1976 [12]. Cyclomatic complexity and Halstead complexity [13] were very popular metrics during that period but those models had a major drawback. The model could be evaluated on a new software module and therefore they demonstrated some relationship between metrics and the number of defects.
Shen et al. [14] built a linear regression model in order to examine error proneness of new software modules. However there were some preciseness issues in that model and Munson et al. proposed a classification model with higher accuracy [15].
With the increased popularity of version control systems several process metrics prediction models were proposed during 2000s.
There were certain limitations in bug prediction models developed during 2000s. One limitation was inability to predict defects whenever a source code file is changed. Just In Time (JIT) bug prediction models were introduced to overcome this limitation and it is also an active research area which allows predicting defects whenever we change the source code. Another drawback was predicting defects for new projects and projects having very little historical information. Cross defect prediction models were introduced as a solution to this limitation. Figure 2.1 demonstrates evolution of bud prediction models over the time.

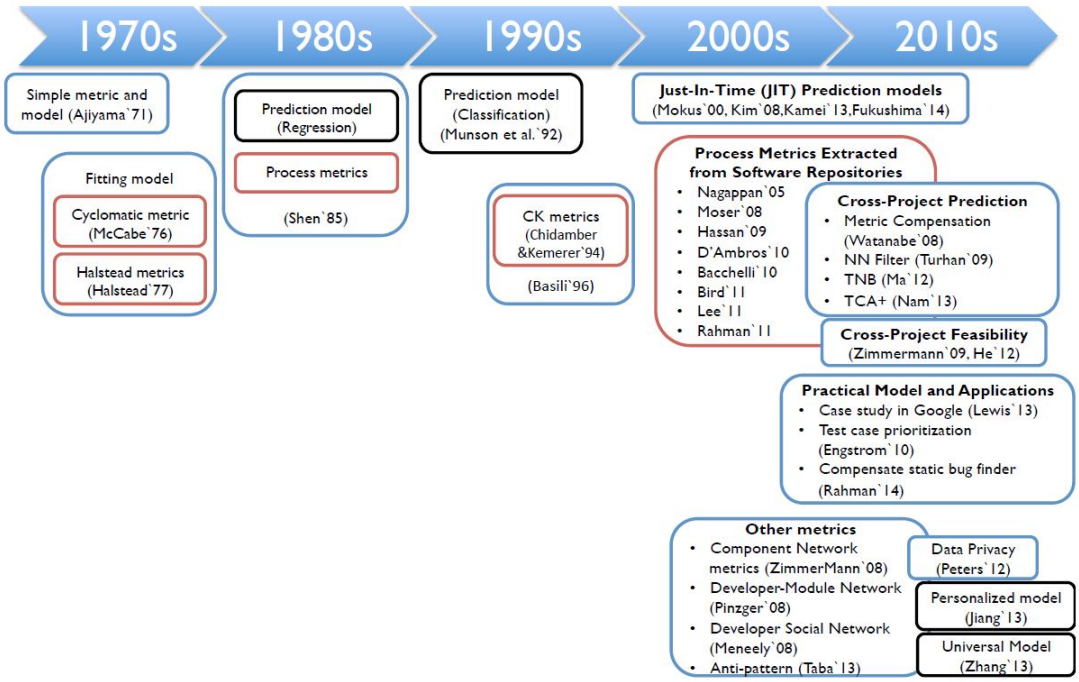

Figure 2.1: History on bug prediction studies

2.2 Bug prediction process

Most of the bug prediction techniques are based on machine learning. The initial step in those approaches are to generate instances from software archives such as VCS and issue tracking systems and etc. Each instance can then be labeled as either clean or buggy. If it is a buggy instance, the number of bugs in the instance can be labeled too. When we have an adequate training data set, we can train our prediction model with the use of these training data. The training model is now able to predict whether a new given instance is buggy or not. Figure 2.2 demonstrates the common process of bug prediction.

2.3 Metrics used in bug prediction

Bug prediction metrics can be mainly categorized into two sections : Code Metrics and Process Metrics. Code metrics are derived directly from the source code whereas process metrics aggregate information from VCS such as Github and issue tracking systems such as Bugzilla.

2.3.1 Code metrics

As mentioned in the introduction chapter LOC, CC, number of classes, number of methods and etc have been used as code metrics to build bug prediction models.

H.Zhang [16] has conducted a research to prove that simple static code attributes such as LOC can be useful predictors of software quality. He has analyzed two public defect datasets: the Eclipse dataset and the NASA dataset (Figure 2.3). The *ranking ability* of LOC proposed by

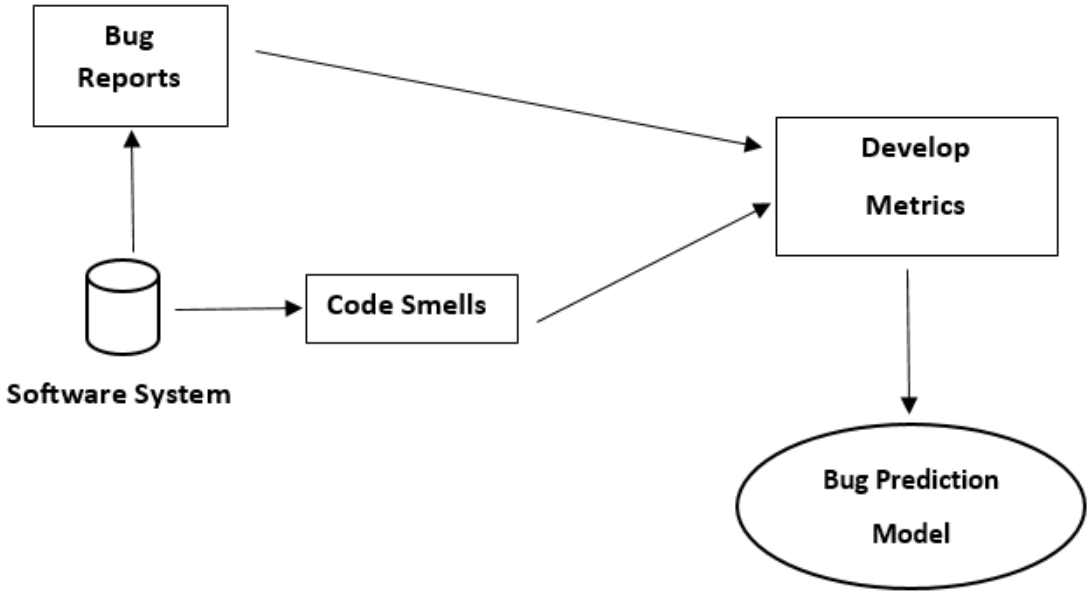

Figure 2.2: The common process of bug prediction

		Package Level				File Level			
		Top 5%	Top 10%	Top 15%	Top 20%	Top 5%	Top 10%	Top 15%	Top 20%
Pre-release defects	2.0	21.74%	33.65%	41.68%	51.64%	24.57%	37.01%	46.99%	53.48%
	2.1	27.60%	40.11%	47.25%	56.17%	28.82%	43.46%	53.97%	61.01%
	3.0	29.88%	43.40%	51.63%	60.34%	32.98%	46.28%	55.05%	62.29%
Post-release defects	2.0	29.55%	41.66%	48.53%	55.73%	34.16%	46.87%	55.73%	61.88%
	2.1	33.53%	43.81%	49.70%	56.04%	28.09%	40.52%	47.72%	54.31%
	3.0	34.16%	46.68%	57.56%	63.49%	29.97%	44.05%	52.41%	60.62%

Figure 2.3: Percentage of defects contained by top k% largest modules in Eclipse [16]

Fenton and Ohlsson [17] has been used in this research and he has found out that this capability can be actually modeled by a Weibull distribution function. He also has discovered that by using defect density values aggregated from a small percentage of the largest modules, he can improve LOC's ability to predict the number of defects. In this research they have demonstrated by using typical classification techniques they are able to predict defective components more accurately based on LOC.

The results of this research is highly dependent on the datasets obtained from Eclipse and NASA defect dataset. If there were some errors in bug data collection and recording, the results may be invalid.

2.3.2 Process metrics

How often code lines are added, changed and deleted as well as recent activities and etc are considered as process metrics. Process metrics require historical information from VCS and issue tracking systems. Nagappan et al. [18] has conducted a research on relative code churn measures to predict system defect density. Code churn measures the changes made to an instance over a period of time.They explain using already conducted literature, absolute measures such as LOC are poor indicators and measures based on change history are better indicators.

The relative code churn measures of the research are as follows :

1. M1: Churned LOC / Total LOC
2. M2: Deleted LOC / Total LOC
3. M3: Files churned / File count
4. M4: Churn count / Files churned
5. M5: Weeks of churn / File count
6. M6: Lines worked on / Weeks of churn
7. M7: Churned LOC / Deleted LOC
8. M8: Lines worked on / Churn count

The study has compared predictive models built using absolute measures against those built using the relative churn. They use the technique of data splitting [19] to measure the ability of the relative code churn measures to predict system defect density. The study has randomly selected two thirds of the binaries (1645) to build the prediction model and use the remaining one third (820) to verify the prediction accuracy.
In order to measure the sensitivity of prediction, the study has run a correlation analysis between the estimated and actual values.A high positive correlation coefficient indicates that when the actual defect density increases estimated defect density also increases accordingly.

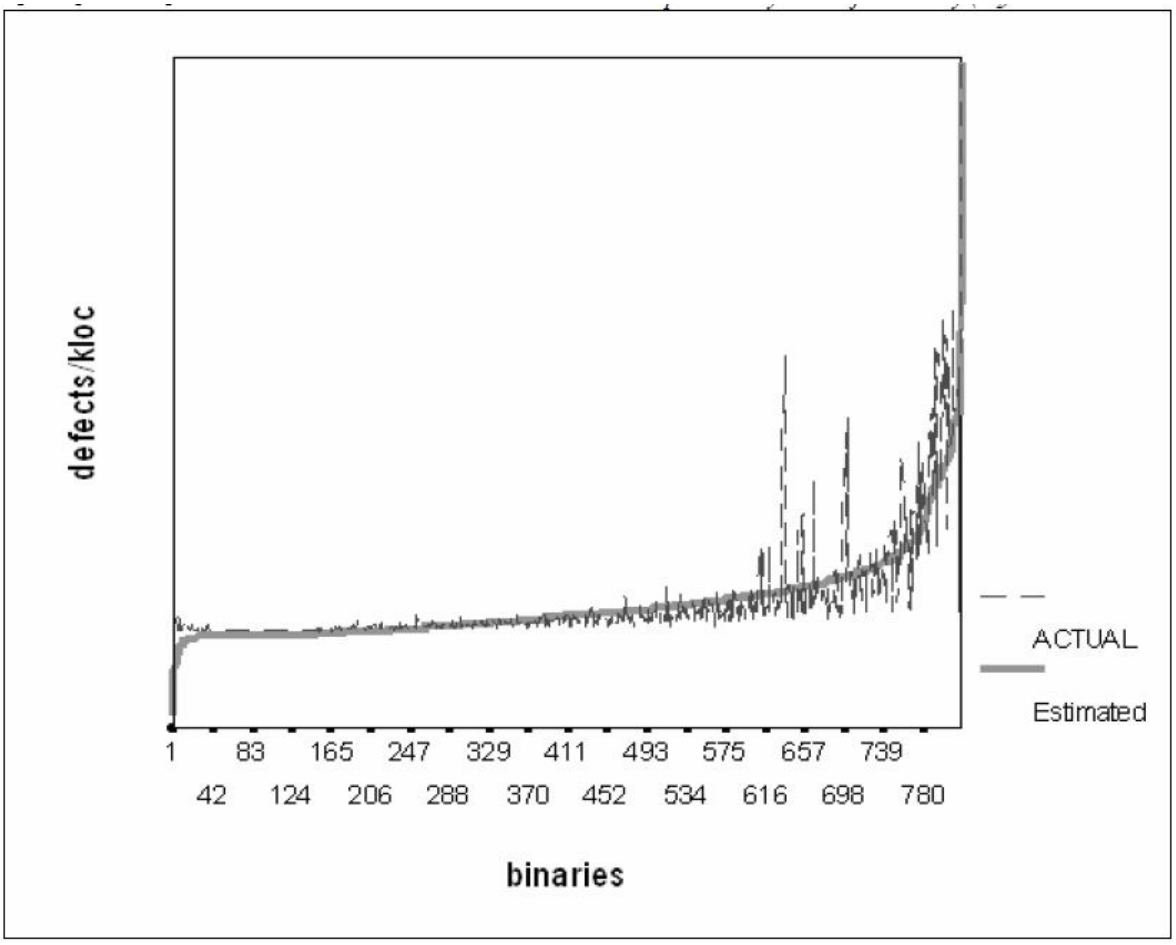

Figure 2.4: Actual vs. estimated system defect density [18]

The results of the study is heavily dependent on data collected from a VCS. VCS only records information when a developer checks out or check in files. A developer might have checked out a file for a long period and may do few changes which will display weeks of churn measures.

2.3.3 Ownership and authorship

According to some studies [20] [21] human factors have a major impact on quality of a software. Bird et al. [22] has conducted a research to investigate the relationship between ownership measures and software failures. They have used Windows Vista and Windows 7 systems for this research. *Ownership* describes responsibility of a developer on a particular software component. According to the study every software has different kinds of contributors. A developer whose ownership is below 5% to software component is known as a *Minor contributor*. A developer whose ownership is at or above 5% is known as a *Major contributor.*

The study has made following recommendations based on the findings :

- Changes made by minor contributors should be reviewed with more scrutiny.

- Potential minor contributors should communicate desired changes to developers experienced with the respective binary.
- Components with low ownership should be given priority by QA resources.

2.3.4 Network metrics

Zimmermann et al. in their study [23] evaluated how well network metrics could be used for bug prediction. The purpose of the research was to identify the interaction between elements and investigate how dependencies correlate with defect prediction. One of the hypothesis that they investigated on the research was that network measures on dependency graphs can predict the number of post release defects. The results of the research have been compared against complexity metrics. The study concludes that the recall of prediction models built with network metrics is 10% higher than for models built using complexity metrics. They are also better predictors of critical bugs.
Tosun et al. [24] has shown that network measures are important indicators of defective modules for large and complex systems, but their impact on small scale projects is not significant.

2.3.5 Code smells based metrics

The objective of this research is to develop a bug prediction model based on code smells. There are some already conducted studies and developed bug prediction models based on code smells. There is empirical evidence in the research community that code smells have negative impact on error proneness and practitioners should pay more attention to systems with a high prevalence of smells during development and maintenance [25].
Taba et al. [26] has conducted a research to investigate the relationship between anti patterns and defect density. They have done the study on multiple versions of two open source systems Eclipse and AgroUML.

The research has proposed 4 anti pattern based metrics

1. Average Number of Antipatterns (ANA)
2. Antipattern Complexity Metric (ACM)
3. Antipattern Recurrence Length (ARL)
4. Antipattern Cumulative Pairwise Differences (ACPD)

The study has shown the files with anti patterns tend to have a higher bug density than others. Proposed metrics provide additional explanatory power over traditional metrics such as LOC, PRE and Churn.Among the measures ARL has shown a significant improvement and it improves bug prediction both within and cross systems. They have investigated different versions of only two projects and the accuracy of the code smell detection tool used has a higher impact since proposed metrics are based on those results.

F Palomba et al. [27] has conducted a study to investigate how severity of code smells impacts the defect density. The intensity is computed using a code smell detector called JCodeOdor and six different types of code smells were taken into account. They are God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling and Message Chains. The intensity index is an estimation of the severity of a code smell, and its value is defined in the range [1,10].
For a given code smell instance, its intensity is computed based on different kinds of information

1. The code smell detection strategy.
2. The metric thresholds used in the detection strategy.
3. The statistical distribution of the metric values computed on a large dataset represented as a quantile function.
4. The actual values of the metrics used in the detection strategies.

The code smell detector classifies the training data set as smelly and non smelly classes in order to build a bug prediction model. Classes which do not have code smells are therefore getting an intensity value of 0.

The results of the study indicate that the use of intensity always positively contributes to find out bug prone code components. The study is limited in the context of within-project bug prediction and does not specifically provide facts on how the model behaves within a single version of a project. The evaluation of the model among cross projects has also not been considered in this study.

2.4 Summary

We cannot use code smell based metrics only as a single predictor to predict buggy components of a software. There can be files in the source code which do not contain code smells. Therefore we will not be able to predict bug proneness of such components if we use code smell based metrics only.
Even though there are number of code smells can be found in a code base, all the code smells do not carry a same weight to make a software instance error prone. Some code smells may have high impact on error proneness whereas impact of some are very minimal. Which code smells have a significant impact on error proneness of a software instance can be studied further.
The accuracy of a bug prediction model also depends on the code smell detection strategy since the metrics derives are based on those results. Therefore selecting a high accurate detection strategy is also important.
Not all the software projects have enough historical data. Even though a project is in initial stage it can contain lot of code smells. It can be further studied how to predict bugs of a newly started projects by using code smells as a metric.

Chapter 3

Methodology

The technical debt metaphor is obtaining a significant attraction in agile community and it describes how complexities created by short term compromises affect the long term health of a project. According to [28], technical debt is a situation where developers accept to sacrifice one dimension of a software product (i.e losing the quality of the software product) when they try to optimize another dimension (i.e implementing set of new features before a deadline). Even though this sacrifice provides short term benefits, the debt needs to be paid back later. When there is too much technical debt it will reduce the pace of the development and cause poor maintainability of the code. 'Code Smells' are one form of technical debt and it may be possible to use these bad smells to identify error-prone components of a software [29] . Therefore it is obvious that code smells are design debt symptoms which are worth investigating.

3.1 Design overview

The initial bug prediction model will be built using source code metrics. We will use different set of source code metrics suggested in the literature to train our basic model. The systems that we analyze in this research are Apache Ant [1] , Apache Camel [2] , Apache Ivy [3] , Apache Log4j [4], Apache Forrest [5], Apache Lucene [6] , Apache POI [7], Apache Synapse [8] , Apache Tomcat [9] , Apache Velocity [10], Apache Xalan [11], Apache Xerces [12] and jEdit [13] .

We will then enhance our basic model by using code smell based metrics. The objective of the

[1] https://ant.apache.org/
[2] http://camel.apache.org/
[3] http://ant.apache.org/ivy/
[4] https://logging.apache.org/log4j/2.x/
[5] https://forrest.apache.org/
[6] https://lucene.apache.org/
[7] https://poi.apache.org/
[8] http://synapse.apache.org/
[9] http://tomcat.apache.org/
[10] http://velocity.apache.org/
[11] https://xalan.apache.org/
[12] http://xerces.apache.org/
[13] http://jedit.org/

research is to combine code smell based metrics suggested by [26] and [27]. We will then evaluate the predictive power of the enhanced model once those metrics are combined. A future work suggested by [27] has also become a part of this research. That is to evaluate the performance of a code smell based prediction model across projects. We will evaluate how effectively metrics suggested by [26] and [27] contribute to the predictive power of our bug prediction model. The study [26] has evaluated its model only on two open source systems Eclipse and AgroUML. But we will apply the metrics on multiple versions of thirteen different open source projects.

3.2 Metrics used

Following are some of the important source code metrics and code smell metrics that we consider in the research.

3.2.1 Source code metrics

1. Weighted Method Count (WMC) : The sum of complexities of all methods defined in a class.
2. Depth of Inheritance Tree (DIT) : Defines the position of the class in the inheritance hierarchy. If the language supports multiple inheritance this value is equal to the maximum length path.
3. Number of Children (NOC) : Measures the number of child classes of a particular class.
4. Coupling Between object classes (CBO) : The number of classes coupled to a class.
5. Response for a Class (RFC) : The number of unique methods and constructors invoked by a class.
6. Lack of Cohesion of Methods (LCOM) : Defines the set of methods in a class that are not related.
7. Number of Public Methods (NPM): Number of public methods defined in the class.
8. Data Access Metric (DAM) : The ratio of the number of private (protected) attributes to the total number of attributes declared in the class.
9. Lines of Code (LOC) : The number of lines in a class.
10. Inheritance Coupling (IC) : The number of parent classes to which a particular class is coupled.

3.2.2 Code smell based metrics

1. Intensity index
   The intensity index is an estimation of the severity of a code smell, and its value is defined in the range [1,10].

2. Average Number of Antipatterns (ANA)
   Measures the distribution of antipatterns in previous buggy versions of a file.

3. Antipattern Recurrence Length (ARL)
   Captures the consecutive occurrence of antipatterns in a file.

4. Antipattern Cumulative Pairwise Differences (ACPD)
   Measures the growth tendency of the antipatterns in a file over time.

5. Antipattern Complexity Metric (ACM)
   Measures complexity of code smells.

3.3 Model architecture

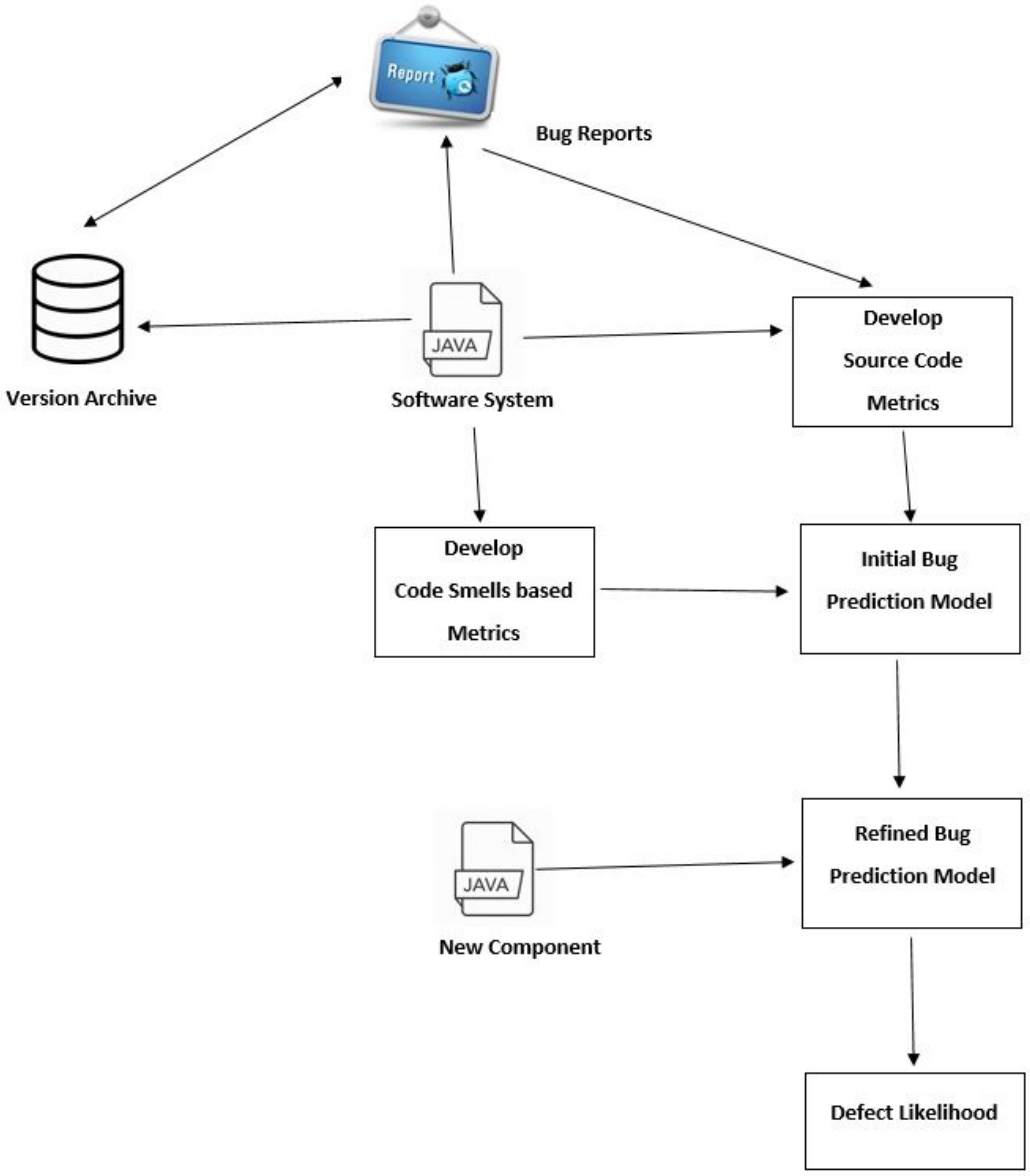

Figure 3.1: Proposed architecture of the bug prediction model

Software systems developed only using Java language will be considered in this research.

A project consists of different versions and a version archive is used to maintain historical information of a project. It stores information related to which files added, deleted, modified to implement a new feature or fix an issue. This is known as a 'Commit'. A commit is always associated with a commit ID, timestamp and the person/developer who made this commit.

Bug reports provide information on erroneous of the existing software components. A bug has a severity. It could be a critical, major, minor or trivial bug. A bug is raised against a version of a software system and assigned to a developer. We consider source code files participating in a

particular file.

The bug prediction model will be built using the above mentioned metrics and its accuracy will be evaluated. The model will also be used to predict the defect likelihood of a new component. Figure 3.1 demonstrates the proposed model architecture.

3.4 Data collection and data processing

We have collected data from publicly available data repositories for this study [30]. Table 3.1 illustrates datasets of different projects and their versions we have gathered for the study.

Project Name	**Version Numbers**
Apache Ant	1.3, 1.4, 1.5, 1.6, 1.7
Apache Camel	1.0, 1.2, 1.4, 1.6
Apache Ivy	2
Apache Log4j	1.0, 1.1, 1.2
Apache Forrest	0.7, 0.8
Apache Lucene	2.0, 2.2, 2.4
Apache POI	1.5, 2.0, 2.5.1, 3.0
Apache Synapse	1.0, 1.1, 1.2
Apache Tomcat	6
Apache Velocity	1.4, 1.5, 1.6.1
Apache Xalan	2.4, 2.5, 2.6, 2.7
Apache Xerces	1.2, 1.3, 1.4.4
jEdit	3.2, 4.0, 4.1, 4.2, 4.3

Table 3.1: Projects and version numbers

The correct preparation of data is essential in order to increase the accuracy of the training model. All the gathered data was in Comma Separated Values (CSV) format and all the files had same set of attributes. The attributes present in a data file is as follows :

Source code metrics present in a data file are : *Project name*, *Version*, *File name*, *Bugs count per file*, *isBuggyFile*, *Weighted Method Count (WMC)*, *Depth of Inheritance Tree (DIT)*, *Number of Children (NOC)*, *Coupling Between object classes (CBO)*, *Response for a Class (RFC)*, *Lack of Cohesion of Methods (LCOM)*, *Afferent couplings (Ca)*, *Efferent couplings (Ce)*, *Number of Public Methods (NPM)*, *Lines of Code (LOC)*, *Data Access Metric (DAM)*, *Measure Of Aggregation (MOA)*, *Measure of Functional Abstraction (MFA)*, *Cohesion Among Class Methods (CAM)*, *Inheritance Coupling (IC)*, *Coupling Between Methods (CBM)*, *Average Method Complexity (AMC)*, *Maximum Cyclomatic Complextiy (MCC)* and *Average Cyclomatic Complexity (ACC)*. *Intensity*, *Average Number of Antipatterns (ANA)*, *Antipattern Complexity Metric (ACM)*, *Antipattern Recurrence Length (ARL)* and *Antipattern Cumulative Pairwise Differences (ACPD)* are the code smell based metrics present in a data file.
Every dataset contains some meta data other than the metric information. Some of the attributes

present in each file has no impact on the bug prediction model. Therefore we have eliminated Project name, File name and Version name from the data sets. If there is at least one bug reported against a file in a particular version that file is considered as a ‘isBuggyFile’ in the data set. The same file can be a non buggy file in a later version. The number of bugs reported against a file has also been recorded. Since predicting number of bugs against a file is not in the scope of this research, we removed ’Bugs count per file’ attribute too.

The data needs to be in numerical format in order to improve the accuracy. Therefore when designing the model we have considered only the numerical attributes. Some of the attributes have higher correlations with another attribute.

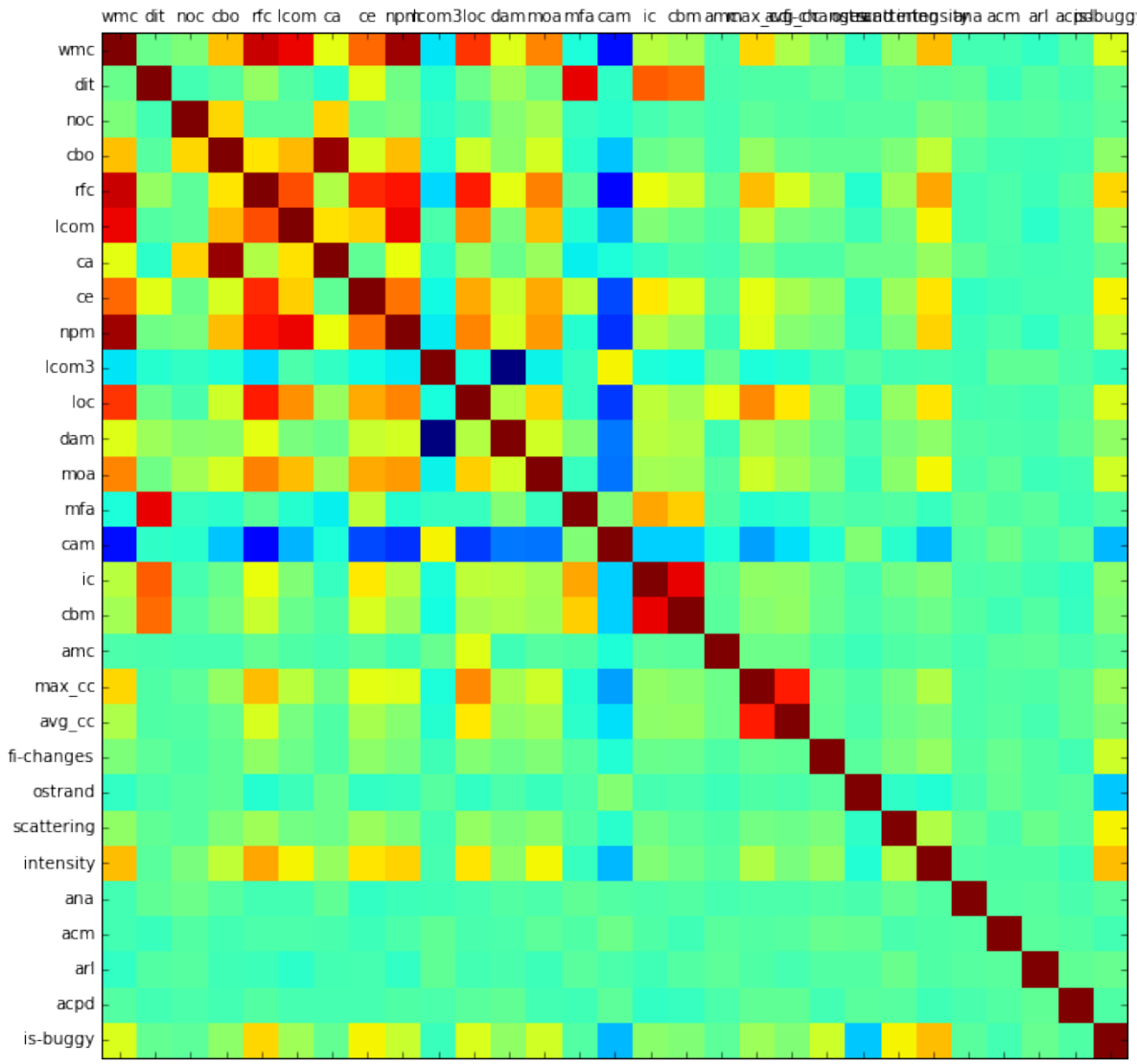

Figure 3.2: Matplotlib representation of higher correlations of some attributes

The Figure 3.2 is the Matplotlib representation of correlation between columns. Blue, Cyan, Yellow, Red to Darked will represent less to more correlations respectively. We do expect a darked line running from top left to bottom right. This diagonal indicates overlapping of the same column in both x and y axis, but red or darked lines in other areas represent that different columns in our dataset have higher correlations. We have observed the impact of each attribute

on our training model and only the most effective attributes were selected for prediction.

3.5 The role of the algorithm

We will use machine learning classifiers to build the model. We will supply training data to the algorithm. As demonstrates in the Figure 3.3 the data is analyzed using the logic of the algorithm. This analysis evaluates the data with respective to a mathematical model and logic associated with the algorithm. The algorithm uses the results of the analysis to adjust internal parameters to produce the model that has been trained to best fit the features in the training data to produce the associated class results. This best is defined by evaluating a function specific to a particular algorithm. When the model is trained it is later called via predict function by passing the real data. Using only the features in the real data, the train model is now capable of classifying the real data.

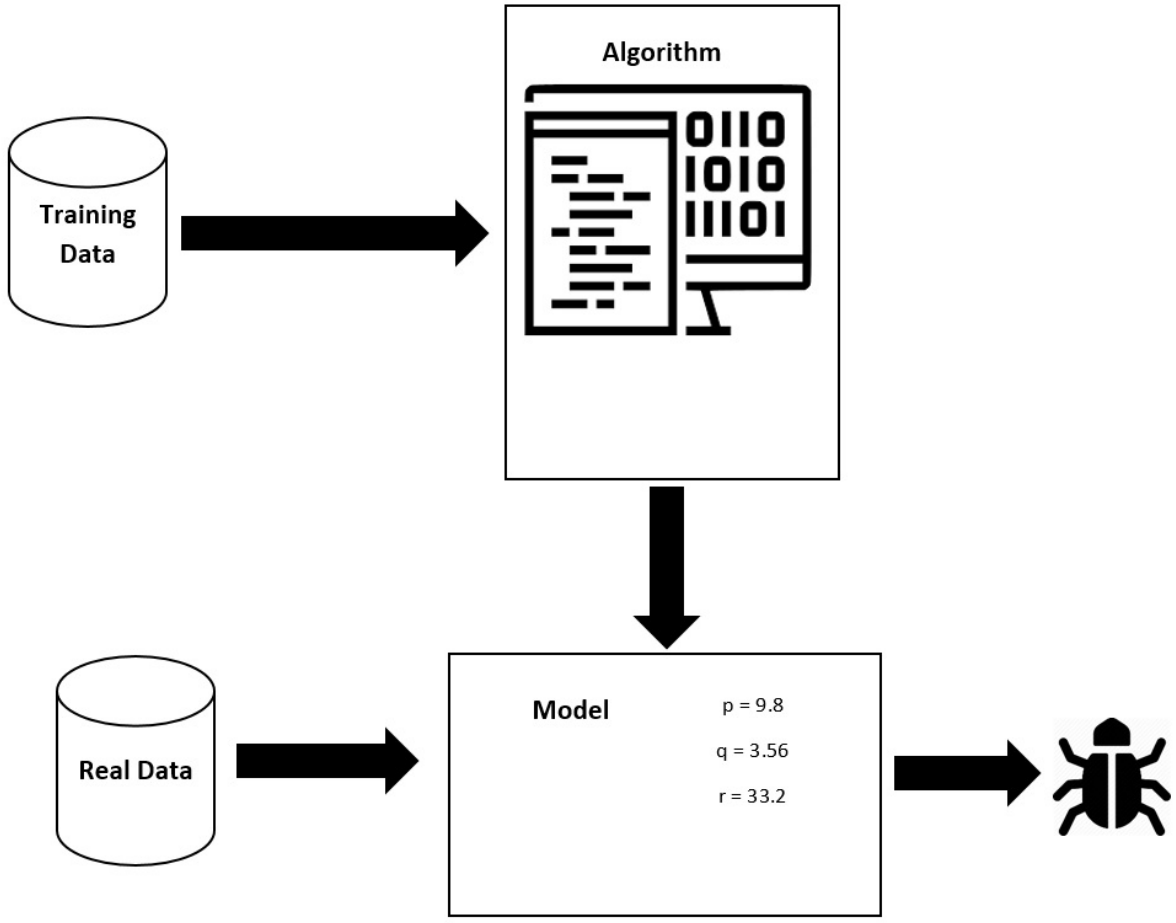

Figure 3.3: The role of algorithm

3.6 Algorithm selection factors

There are different machine learning techniques available. This is a predictive mode. Therefore our **Learning Type** would be *Supervised Learning*. We will consider only the algorithms which

support Supervised Learning in this study. The prediction results can be classified into two categories : *Regression* and *Classification*. Regression predictive modeling maps inputs variables to a continuous output variables whereas classification maps input variable to a discrete output value. The **Result Type** of the bug prediction model needs to be a binary outcome. i.e whether we can classify a source code file as buggy or not. Since this is a classification problem we will consider only the algorithms which supports binary classification.

3.7 Candidate algorithms

3.7.1 Naive Bayes algorithm

Naive Bayes algorithm is based on Bayes Theorem. Naive Bayes classifier assumes that all the features are unrelated and independent from each other. Therefore the availability or absence of a feature does not influence the availability or absence of any other feature.It calculates the probability based on likelihood and probability of previous data. The class with highest probability is considered as the most likely class.

3.7.2 Logistic Regression

Logistic regression is based on logistic function (AKA sigmoid function). The algorithm measures the relationship of each feature and weights them based on their impact and result. The resultant value is in between 1 and 0.

3.7.3 Random Forest

This algorithm is based on Decision Trees. It creates the forest with a number of trees. The classifier produces a higher accurate result when there are higher number of trees in the forest.

3.7.4 Rationale for the use of the algorithms

We selected *Naive Bayes*, *Logistic Regression* and *Random Forest* as our candidate algorithms to build the model. The main reasons for selecting these three algorithms are we have labeled data, they are in numerical format and they are categorical data, we do experiments with the increasing number of examples and we do the training incrementally.

Chapter 4

Proposed Solution Details

4.1 Programming environment

We have used Python 3 as the programming language. Python has a rich set of libraries which is very useful for Machine Learning development. We have used scikit-learn, NumPy, Pandas and Matplotlib libraries to build the model. We have used the Jupyter Notebook as our programming environment. The Jupyter Notebook is an open-source web application that allows to create and share documents that contain live code, equations, visualizations and narrative text.

4.2 Preparing data

The objective of this research is to evaluate the model within a particular version of a project, within a particular project and across multiple projects. We split every dataset into two parts. We have used 70% of each dataset to train the model and 30% of each dataset to test the model. We have never used test instances to train the model.

For example if we consider within version scenario, there were 745 instances in Apache Ant 1.7 version. We first randomly removed 5 instances of this dataset to be used as real data. The real data will be used to evaluate the accuracy of the prediction. There were 740 instances left in the dataset. We used 518 instances (70%) to train our model and 222 instances (30%) to test the model. Similarly for within project scenario, we studied 5 different versions of Apache Ant. They were 1.3, 1.4, 1.5, 1.6 and 1.7. We selected version 1.7 as the real dataset and other versions were used to train and test the model. There were 947 instances in the other 4 versions of Apache Ant. There were 662 training instances (70%) and 285 test instances (30%). For cross project prediction we used Apache Ivy version 2 as the real dataset. This can be assumed as a newly started project with less historical information. There were 16576 instances in all the versions in all the projects. We used 11603 instances to train the model (70%) and 4973 instances (30%) to test the model.

The boolean attribute is-buggy will indicate whether a file in a particular version is buggy (represented as 1) or not (represented as 0). The number of true instances contain in the dataset will also have an impact on the accuracy of the model. There must be a balanced number of true and

false instances in our datasets. Preparing a real dataset with balanced number of buggy and not buggy instances is really challenging. The number of buggy instances in all versions of Apache Ant project were only 19.43%. In all versions of all projects, the number of instances reported as buggy were 33.96%.

4.3 Building the model

4.3.1 Modules

1. File Loader Module
   All the datasets used in the study were in CSV format and they all had same number of attributes. We used read_csv() method of Pandas library to read the files.

2. Feature Column Selector Module
   There are some redundant attributes in the dataset which do not have impact on the bug prediction. We should train the model only using the optimal attributes. This module will filter only the feature columns from the dataset.

3. Data Split Module
   As discussed in Preparing Data module 70% of the dataset will be used to train the model and 30% will be used for testing purposes. The below Python code snippet shows data splitting functionality.

```
def split_data(data_frame, feature_col_names):
    x = data_frame[feature_col_names].values  # predictor feature columns
    y = data_frame[predicted_class_names].values  # predicted class (1 = True, 0 = False)
    split_test_size = 0.30  # test size = 0.3 is 30%
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size= split_test_size, random_state= 42)
    return x_train, x_test, y_train, y_test
```

Figure 4.1: Python code for data split module

4. Imputer
   Sometimes certain values can be null. Even though they are not null, they may be represented in the form empty strings or empty spaces. These null values need to be removed or replaced. We have used imputer function provided by scikit-learn for this.

5. Training with the data
   We used scikit-learn implementations to develop Machine Learning algorithms. Using a Machine Learning algorithm with scikit-learn involves three steps.

   (a) Import the particular algorithm implementation.

   (b) Initiate the model object.

   (c) Invoking fit() method with training data.

We have used GaussianNB, RandomForestClassifier and LogisticRegression provided by scikit-learn to train the model.

Chapter 5

Evaluation and Results

This chapter describes how the bug prediction model was evaluated against different evaluation metrics.

5.1 Evaluation measures

There are four different prediction outcomes provided by a classification model.

- True positive (TP): Buggy instances predicted as buggy.
- False positives (FP): Clean instances predicted as buggy.
- True negative (TN): Clean instances predicted as clean.
- False negative (FN): Buggy instances predicted as clean.

There are different evaluation measures proposed in the literature based on the above prediction outcomes.

5.1.1 Accuracy

The fraction of all correctly classified instances with respect to all the instances.

$$\frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

5.1.2 Precision

The fraction of the positive predictions that are actually positive.

$$\frac{TP}{TP + FP} \tag{5.2}$$

5.1.3 Recall

The fraction of correctly predicted buggy instances among all buggy instances.

$$\frac{TP}{TP + FN} \tag{5.3}$$

5.1.4 F-measure

F-measure is a harmonic mean of precision and recall.

$$\frac{2x(PrecisionxRecall)}{Precision + Recall} \tag{5.4}$$

5.1.5 Receiver Operating Characteristic (ROC) curve

ROC curve plots TP rate Vs FP rate.

5.1.6 Precision Recall (PR) curve

PR curve plots Precision Vs Recall.

5.2 Evaluation results

5.2.1 Within version

Following are the results of Apache Ant 1.7 version.

5.2.1.1 Basic model

We selected relatively less correlated features as our source code metrics. The selected attributes were

1. Weighted Method Count (WMC)
2. Depth of Inheritance Tree (DIT)
3. Number of Children (NOC)
4. Afferent couplings (Ca)
5. Lines of Code (LOC)
6. Data Access Metric (DAM)
7. Measure Of Aggregation (MOA)
8. Cohesion Among Class Methods (CAM)

9. Inheritance Coupling (IC)
10. Coupling Between Methods (CBM)
11. Average Method Complexity (AMC)
12. Maximum Cyclomatic Complextiy (MCC)

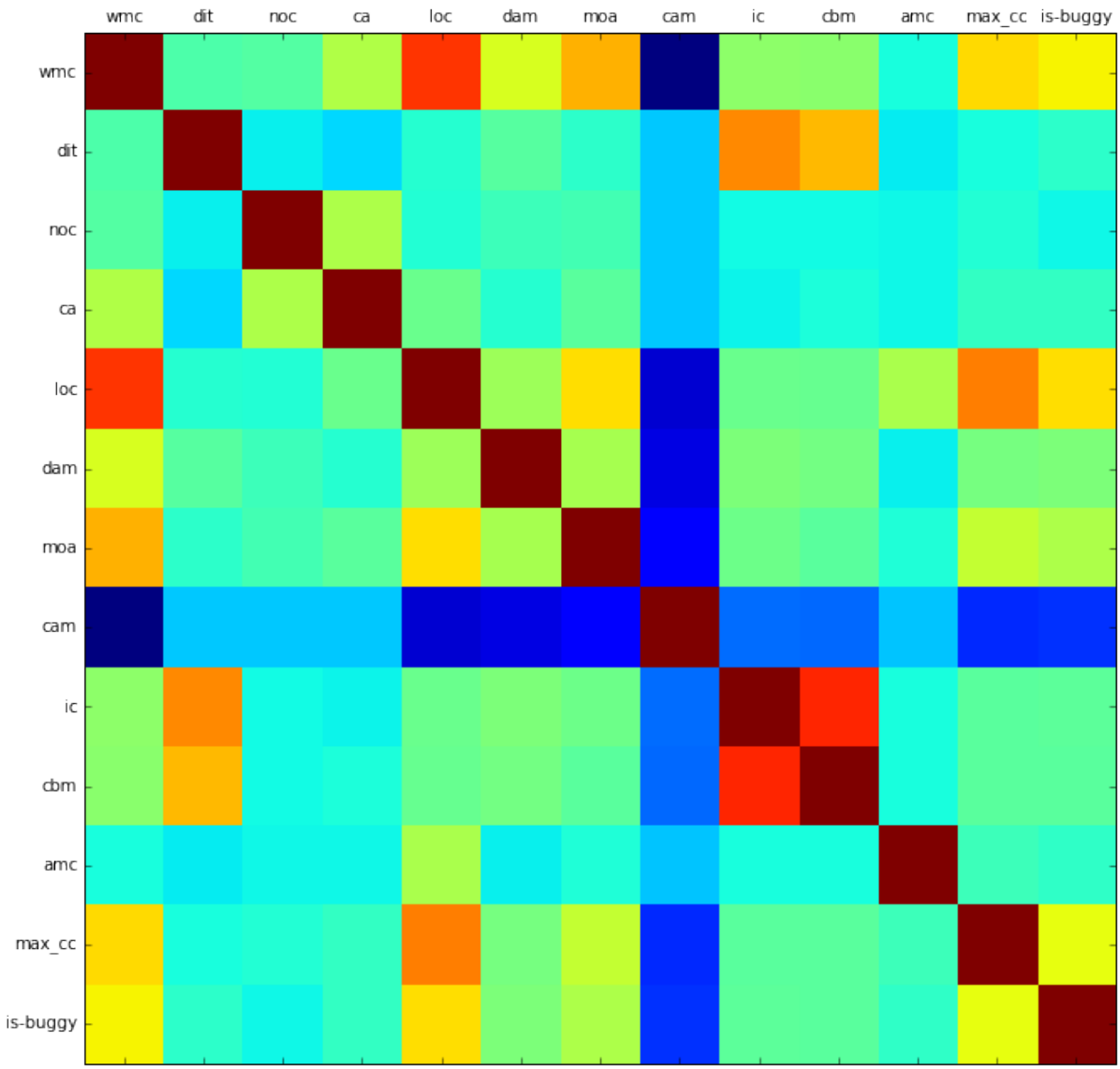

Figure 5.1: Matplotlib representation of correlations of selected source code attributes

Measure	Naive Bayes	Random Forest	Logistic Regression
Accuracy of train data	81.27 %	98.84%	83.20 %
Accuracy of test data	80.63 %	82.43%	82.88 %
TP	10.36%	10.36%	9.01%
TN	70.27%	72.07%	73.87%
FP	6.76%	4.95%	3.15%
FN	12.61%	12.61%	13.96%
Precision	0.61	**0.68**	**0.74**
Recall	0.45	**0.45**	**0.39**
F1 score	0.52	**0.54**	**0.51**
Number of True cases in train data	**22.16%**		
Number of False cases in train data	**77.84%**		
Real Data Set	[1, 0, 0, 1, 0]		
NB Prediction	[1, 1, 1, 1 ,1]		
RF Prediction	[1, 0 ,1, 0 ,0]		
LR Prediction	[1, 0, 0, 0 ,0]		

Table 5.1: Basic model within version Apache Ant 1.7

As per Table 5.1 Random Forest and Logistic Regression algorithms provide good results compared to Naive Bayes, but none of the algorithms are showing a good precision and recall. Logistic Regression algorithm based model has predicted 4/5 randomly selected instances of the real data accurately, but none of the algorithms show good prediction results within version. Number of buggy instances presented in the dataset is 22.16% and we have a huge number of clean instances in the dataset. This could be a reason why our basic model is not performing as expected. Therefore we used the same dataset on our enhanced model to check whether there is an improvement in prediction accuracy.

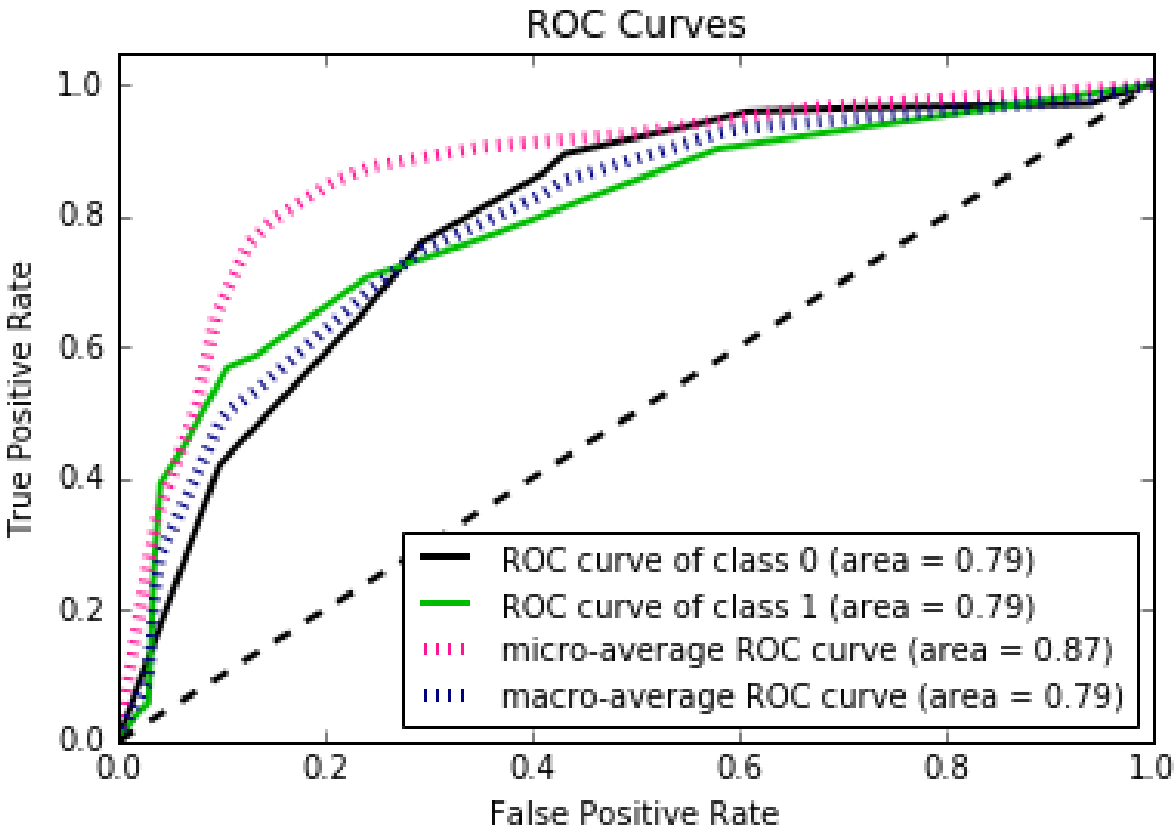

Figure 5.2: Apache Ant 1.7 ROC curve for basic model - Random Forest

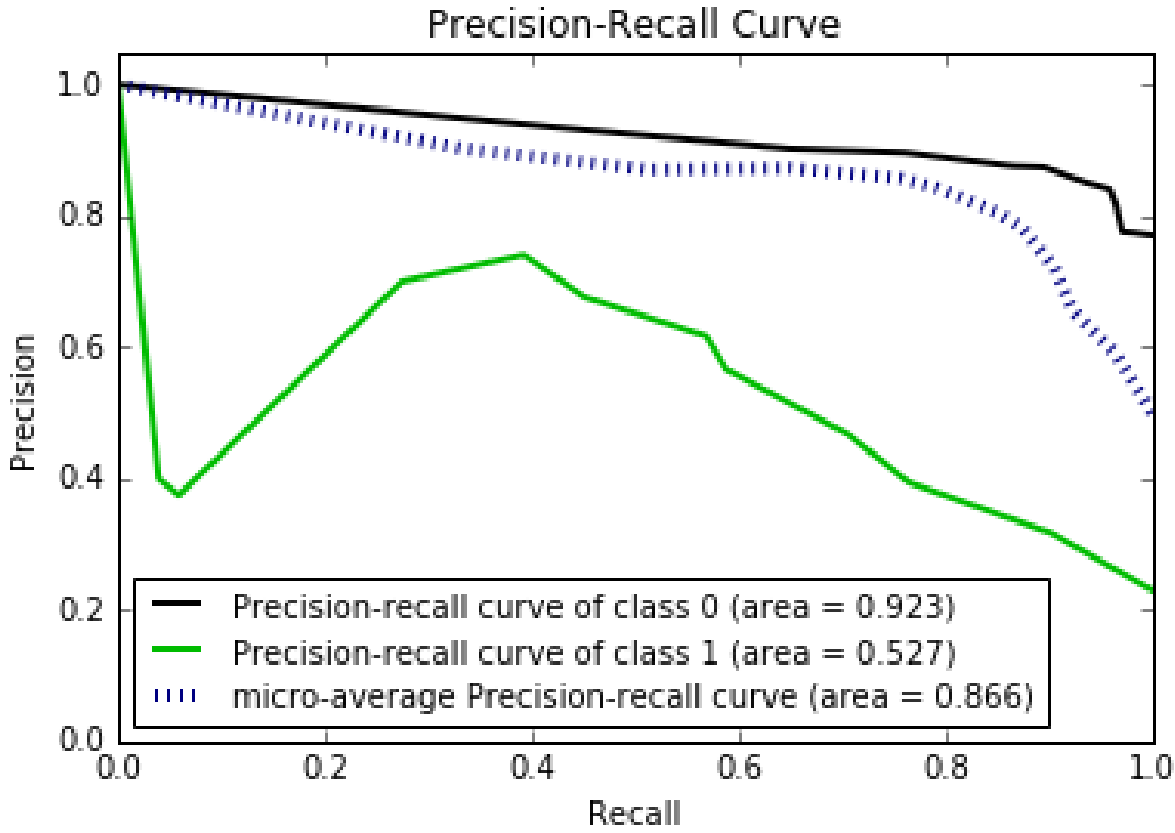

Figure 5.3: Apache Ant 1.7 PR curve for basic model - Random Forest

5.2.1.2 Enhanced model with code smell based metrics

Following are the results of enhanced model once code smell based metrics

1. Intensity
2. Average Number of Antipatterns (ANA)
3. Antipattern Complexity Metric (ACM)
4. Antipattern Recurrence Length (ARL)
5. Antipattern Cumulative Pairwise Differences (ACPD)

are integrated into Apache 1.7 Version.

Measure	**Naive Bayes**	**Random Forest**	**Logistic Regression**
Accuracy of train data	85.91 %	100.00 %	95.17 %
Accuracy of test data	86.49 %	100.00 %	98.20 %
TP	13.96%	22.97%	22.52%
TN	72.52%	77.03%	75.68%
FP	4.50%	0.00%	1.35%
FN	9.01%	0.00%	0.45%
Precision	0.76	**1.00**	0.94
Recall	0.61	**1.00**	0.98
F1 score	0.67	**1.00**	0.96
Number of True cases in train data	**22.16%**		
Number of False cases in train data	**77.84%**		
Real Data Set	[1, 0, 0, 1, 0]		
NB Prediction	[1, 1, 1, 1 ,1]		
RF Prediction	**[1, 0, 0, 1, 0]**		
LR Prediction	[1, 0, 0, 0 ,0]		

Table 5.2: Enhanced model within version Apache Ant 1.7

As Table 5.2 has demonstrated we were able to achieve a very good results by integrating code smell based metrics to our basic model. Random Forest Algorithm provided very accurate results with both Precision and Recall to be 1.00. Logistic Regression based model also provided significantly improved results compared to source code based metrics. RF algorithm based model also accurately predicted randomly selected read data instances.

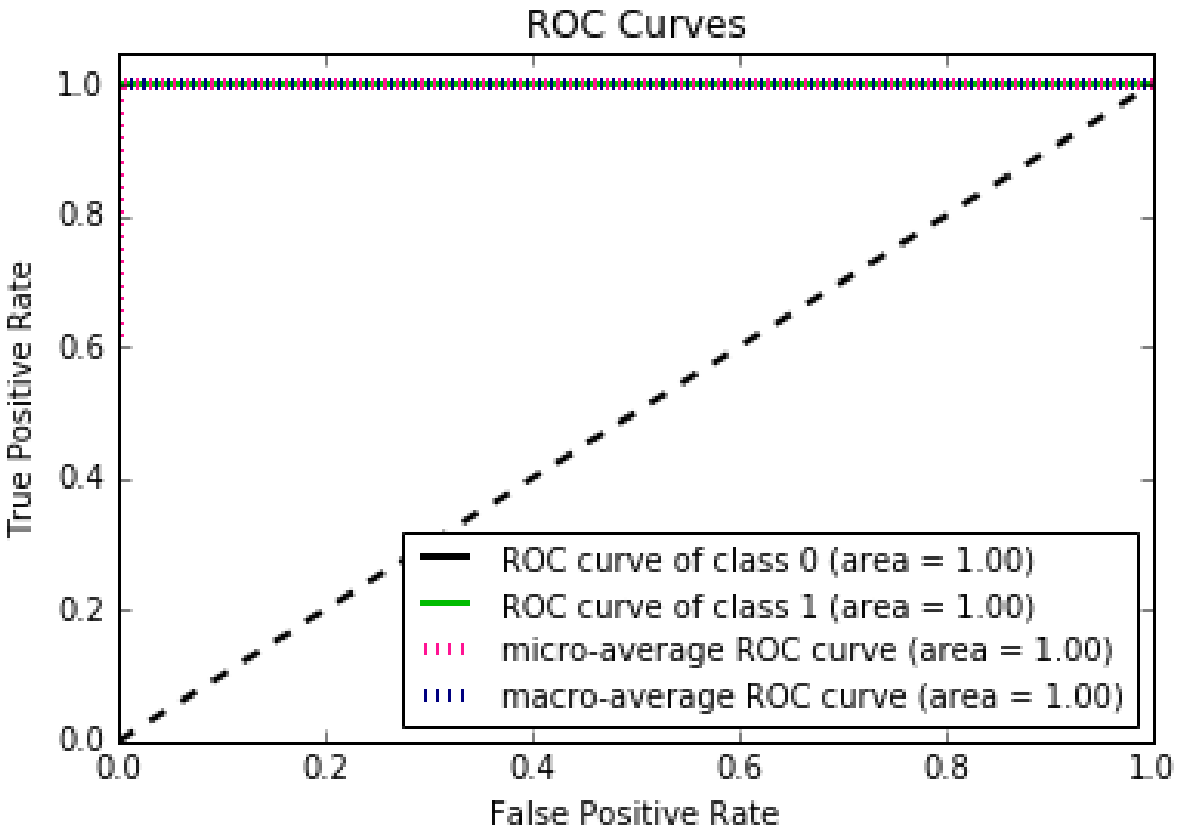

Figure 5.4: Apache Ant 1.7 ROC curve for enhanced model - Random Forest

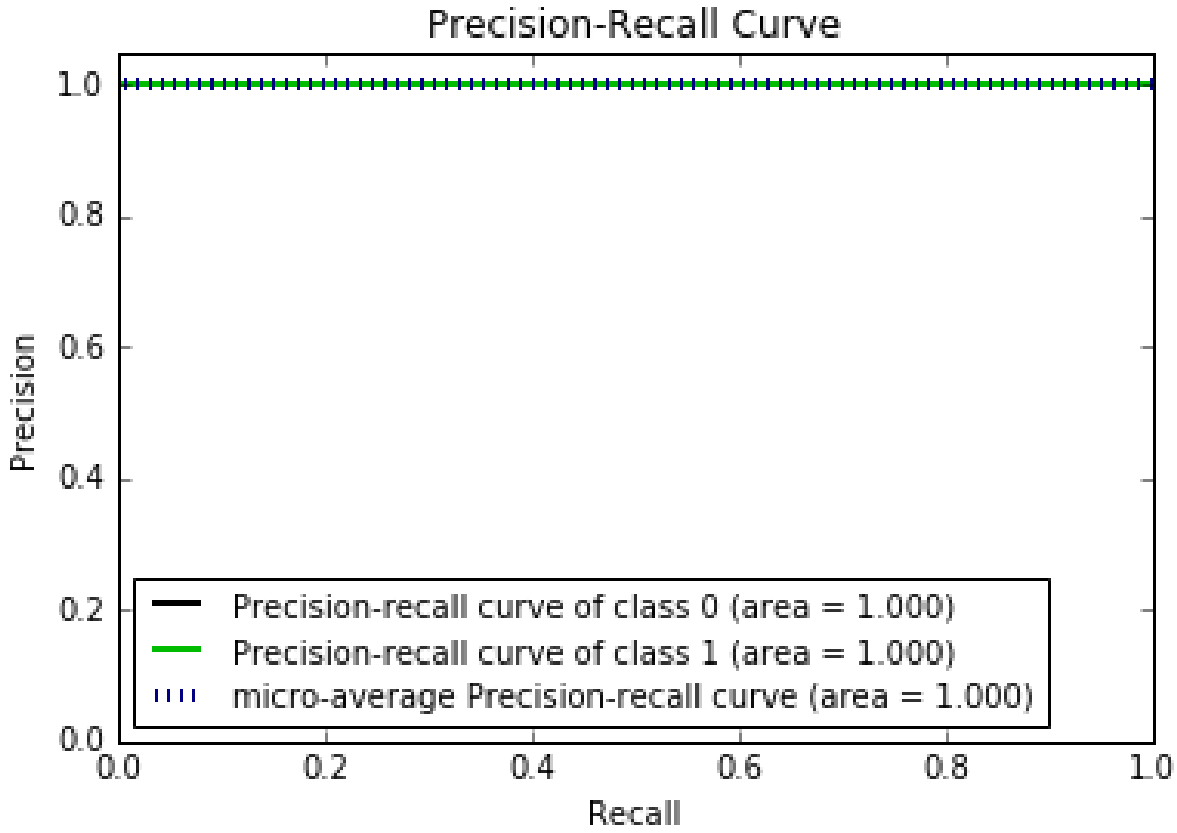

Figure 5.5: Apache Ant 1.7 PR curve for enhanced model

5.2.2 Within project

Following are the results for Apache Ant project. We have analyzed five versions.

5.2.2.1 Basic model

Measure	Naive Bayes	Random Forest	Logistic Regression
Accuracy of train data	77.49 %	96.53 %	81.42 %
Accuracy of test data	74.39 %	77.54 %	81.75 %
TP	8.77%	7.02%	2.81%
TN	65.61%	70.53%	78.95%
FP	15.79%	10.88%	2.46%
FN	9.82%	11.58%	15.79%
Precision	0.36	0.39	0.53
Recall	0.47	0.38	0.15
F1 score	0.41	0.38	0.24
Number of True cases in train data	**19.43%**		
Number of False cases in train data	**80.57%**		
Real Data Set (V 1.7)	[1,1,1,1,1]		
NB Prediction	[1, 0, 1, 1, 1]		
RF Prediction	[0 ,0 ,1, 1, 0]		
LR Prediction	[0, 0, 0, 0, 0]		

Table 5.3: Basic model within Apache Ant project

As we can see in the Table 5.3 none of the algorithms has performed well within the project. We have analyzed the same dataset against our enhanced bug prediction model.

5.2.2.2 Enhanced model with code smell based metrics

Measure	Naive Bayes	Random Forest	Logistic Regression
Accuracy of train data	83.38 %	100.00 %	91.09 %
Accuracy of test data	81.75 %	87.72 %	87.72 %
TP	10.88%	7.37%	10.18%
TN	70.88%	80.35%	77.54%
FP	10.53%	1.05%	3.86%
FN	7.72%	11.23%	8.42%
Precision	0.51	0.88	0.72
Recall	0.58	0.40	0.55
F1 score	0.54	0.55	0.62
Number of True cases in train data	**19.43%**		
Number of False cases in train data	**80.57%**		
Real Data Set (V 1.7)	[1,1,1,1,1]		
NB Prediction	[1, 0 ,1 ,1 ,1]		
RF Prediction	[0 1 1 1 0]		
LR Prediction	[1, 1, 1 ,1 ,1]		

Table 5.4: Enhanced model within Apache Ant project

The evaluation results show some progress with respect to precision and recall in the enhanced model, but our model is not performing as expected in Apache Ant project even when code smell based metrics are integrated.
The number of True instances in the training set is relatively less. It is 19.43% of the total dataset. This could also be a reason. Therefore we have tested our model in a more balanced data set.

We analyzed 4 versions Apache Xalan project. The total number of all instances were 2411 and there were 37.66% of buggy instances in the data set.

Measure	Naive Bayes	Random Forest	Logistic Regression
Accuracy of train data	71.25 %	**100.00 %**	96.38 %
Accuracy of test data	71.55 %	**100.00 %**	95.30 %
TP	15.33%	38.95%	38.40%
TN	56.22%	61.05%	56.91%
FP	4.83%	0.00%	4.14%
FN	23.62%	0.00%	0.55%
Precision	0.76	**1.00**	0.90
Recall	0.39	**1.00**	0.99
F1 score	0.52	**1.00**	0.94
Number of True cases in train data	**37.66%**		
Number of False cases in train data	**62.34%**		
Real Data Set (V 1.7)	[1,1,1,1,1]		
NB Prediction	[1,1,1,1,1]		
RF Prediction	[1,1,1,1,1]		
LR Prediction	[1,1,1,1,1]		

Table 5.5: Enhanced model within Apache Xalan

As we can see in the Table 5.5 when we have a good amount of buggy instances in the training set the enhanced model has produced significantly good results and particularly Random Forest algorithm based model has provided the most accurate results.

5.2.3 Cross projects prediction

We used all the versions of all the projects to train our both basic model and enhanced model. The trained models were tested against one version of Apache Ivy project (Version 2).

Measure	**Naive Bayes**	**Random Forest**	**Logistic Regression**
Accuracy of train data	66.43 %	90.88 %	68.40 %
Accuracy of test data	66.66 %	68.05 %	68.79 %
TP	4.12%	13.47%	6.94%
TN	62.54%	54.57%	61.85%
FP	4.14%	12.11%	4.83%
FN	29.20%	19.85%	26.38%
Precision	0.50	0.53	0.59
Recall	0.12	0.40	0.21
F1 score	0.20	0.46	0.31
Number of True cases in train data	**33.96%**		
Number of False cases in train data	**66.04%**		
Real Data Set (V 1.7)	[0,0,0,0,1]		
NB Prediction	[0, 0, 1, 0 ,0]		
RF Prediction	[0, 1, 1, 0, 1]		
LR Prediction	[1,1,1,1,1]		

Table 5.6: Basic model prediction on cross projects

Measure	**Naive Bayes**	**Random Forest**	**Logistic Regression**
Accuracy of train data	71.58 %	99.73 %	90.46 %
Accuracy of test data	71.91 %	91.72 %	90.39 %
TP	10.84%	31.61%	33.18%
TN	61.07%	60.10%	57.21%
FP	5.61%	6.58%	9.47%
FN	22.48%	1.71%	0.14%
Precision	0.66	**0.83**	**0.78**
Recall	0.33	**0.95**	**1.00**
F1 score	0.44	**0.88**	**0.87**
Number of True cases in train data	**33.96%**		
Number of False cases in train data	**66.04%**		
Real Data Set (V 1.7)	[0,0,0,0,1]		
NB Prediction	[0, 0 ,1, 0 ,1]		
RF Prediction	[0 ,0, 0 ,0 ,1]		
LR Prediction	[0, 0, 0, 0, 1]		

Table 5.7: Enhanced model prediction on cross projects

There were 16576 instances in the training data set and 33.96% were buggy instances. Similar to within version and within project prediction, the accuracy of the basic model across projects is very low even when there are more than 30% of buggy instances in the training set. We can clearly observe the prediction accuracy is much higher across projects in our enhanced model. Random Forest based model has shown the most accurate results and the accuracy of Logistic Regression based model also has improved significantly when compared with the basic model.

Chapter 6

Conclusion and Future Work

This chapter summarizes the work that has been done in the study, limitations and the future work.

6.1 Conclusion

In this study we provided empirical evidence that code smells based metrics can be really helpful in bug prediction. We developed a bug prediction model by using different source code metrics and code smell based metrics proposed in the literature. We used Naive Bayes, Random Forest and Logistic Regression as our algorithms to build the model. The developed prediction model was trained against multiple versions of thirteen different open source projects. We analyzed the behavior of our bug prediction model within a particular version, within a particular project and across different projects.

We would like to highlight analysis of our study as follows :

- Only the use of source code metrics is not sufficient to predict bugs of a project.
- Higher accuracy could be obtained when code smell based metrics are integrated with source code metrics.
- Among the used algorithms Random Forest has shown a higher accuracy compared to other algorithms. Having rich amount of numerical/ categorical data and training with the increasing number of examples could be main reasons why Random Forest provided better results. The reasons why Naive Bayes did not perform well in the study are our features are not fully independent of each other.
- Cross project bug prediction can be accurately done with the help of code smell based metrics.
- We were able to obtain higher accurate results when no of buggy instances in the training set were more than 30%.

6.2 Future work

The basic model is still based on source code based metrics and we would like to incorporate process and network based metrics to the model and check the accuracy of the model. The current model has been evaluated against only Java based projects. We would like to evaluate the accuracy of this model against Javascript based projects as a future work. This model cannot be used to provide warnings to developers whenever they change the files. As a future work, we would like to investigate how this model could be used for just-in-time (JIT) prediction.

Bibliography

[1] M. Fowler, Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[2] R. Varshneya. (Oct. 2015). There's no such thing as a bug-free app, [Online]. Available: https://www.entrepreneur.com/article/251742.

[3] SourceMaking.com. (n.d). Code smells, [Online]. Available: https://sourcemaking.com/refactoring/smells.

[4] CodingHorror.com. (2006). Code smells, [Online]. Available: https://blog.codinghorror.com/code-smells/.

[5] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," IEEE Transactions on Software Engineering, vol. PP, no. 99, pp. 1–1, 2017, ISSN: 0098-5589. DOI: 10.1109/TSE.2017.2653105.

[6] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in 2013 20th Working Conference on Reverse Engineering (WCRE), Oct. 2013, pp. 242–251. DOI: 10.1109/WCRE.2013.6671299.

[7] M. Abbes, F. Khomh, Y. G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in 2011 15th European Conference on Software Maintenance and Reengineering, Mar. 2011, pp. 181–190. DOI: 10.1109/CSMR.2011.24.

[8] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," Empirical Softw. Engg., vol. 17, no. 3, pp. 243–275, Jun. 2012, ISSN: 1382-3256. DOI: 10.1007/s10664-011-9171-y. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9171-y.

[9] F. Khomh, M. D. Penta, and Y. G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in 2009 16th Working Conference on Reverse Engineering, Oct. 2009, pp. 75–84. DOI: 10.1109/WCRE.2009.28.

[10] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in 2013 35th International Conference on Software Engineering (ICSE), May 2013, pp. 682–691. DOI: 10.1109/ICSE.2013.6606614.

[11] F. Akiyama, “An example of software system debugging.,” in IFIP Congress (1), Jan. 3, 2002, pp. 353–359. [Online]. Available: http://dblp.uni-trier.de/db/conf/ifip/ifip71-1.htmlAkiyama71.

[12] T. J. McCabe, “A complexity measure,” IEEE Trans. Softw. Eng., vol. 2, no. 4, pp. 308–320, Jul. 1976, ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[13] M. H. Halstead, Elements of Software Science (Operating and Programming Systems Series). New York, NY, USA: Elsevier Science Inc., 1977, ISBN: 0444002057.

[14] V. Y. Shen, T.-j. Yu, S. M. Thebaut, and L. R. Paulsen, “Identifying error-prone software ;an empirical study,” IEEE Transactions on Software Engineering, vol. SE-11, no. 4, pp. 317–324, Apr. 1985, ISSN: 0098-5589. DOI: 10.1109/TSE.1985.232222.

[15] J. C. Munson and T. M. Khoshgoftaar, “The detection of fault-prone programs,” IEEE Transactions on Software Engineering, vol. 18, no. 5, pp. 423–433, May 1992, ISSN: 0098-5589. DOI: 10.1109/32.135775.

[16] H. Zhang, “An investigation of the relationships between lines of code and defects,” in 2009 IEEE International Conference on Software Maintenance, Sep. 2009, pp. 274–283. DOI: 10.1109/ICSM.2009.5306304.

[17] N. E. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” IEEE Transactions on Software Engineering, vol. 26, no. 8, pp. 797–814, Aug. 2000, ISSN: 0098-5589. DOI: 10.1109/32.879815.

[18] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., May 2005, pp. 284–292. DOI: 10.1109/ICSE.2005.1553571.

[19] J. C. Munson and T. M. Khoshgoftaar, “The detection of fault-prone programs,” IEEE Transactions on Software Engineering, vol. 18, no. 5, pp. 423–433, May 1992, ISSN: 0098-5589. DOI: 10.1109/32.135775.

[20] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, “Putting it all together: Using socio-technical networks to predict failures,” in Proceedings of the 2009 20th International Symposium on Software Reliability Engineering, ser. ISSRE ’09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 109–119, ISBN: 978-0-7695-3878-5. DOI: 10.1109/ISSRE.2009.17. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2009.17.

[21] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, “Identification of coordination requirements: Implications for the design of collaboration and awareness tools,” in Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work, ser. CSCW ’06, Banff, Alberta, Canada: ACM, 2006, pp. 353–362, ISBN: 1-59593-249-6. DOI: 10.1145/1180875.1180929. [Online]. Available: http://doi.acm.org/10.1145/1180875.1180929.

[22] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ser. ESEC/FSE '11, Szeged, Hungary: ACM, 2011, pp. 4–14, ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025119. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025119.

[23] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in 2008 ACM/IEEE 30th International Conference on Software Engineering, May 2008, pp. 531–540. DOI: 10.1145/1368088.1368161.

[24] A. Tosun, B. Turhan, and A. Bener, Validation of network measures as indicators of defective modules in software systems, Jan. 2009.

[25] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," Empirical Software Engineering, vol. 17, no. 3, pp. 243–275, Jun. 2012, ISSN: 1573-7616. DOI: 10.1007/s10664-011-9171-y. [Online]. Available: https://doi.org/10.1007/s10664-011-9171-y.

[26] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in 2013 IEEE International Conference on Software Maintenance, Sep. 2013, pp. 270–279. DOI: 10.1109/ICSM.2013.38.

[27] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), Oct. 2016, pp. 244–255. DOI: 10.1109/ICSME.2016.27.

[28] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, ser. FoSER '10, Santa Fe, New Mexico, USA: ACM, 2010, pp. 47–52, ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882373. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882373.

[29] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," J. Syst. Softw., vol. 80, no. 7, pp. 1120–1128, Jul. 2007, ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.10.018. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2006.10.018.

[30] The promise repository of empirical software engineering data, 2015.

Appendices

Appendix A

Code Smells

A.1 Bloaters

Bloaters are introduced due to long term poor code choices. They start to emerge as the program evolves. Bloaters hinder maintainability of the code.

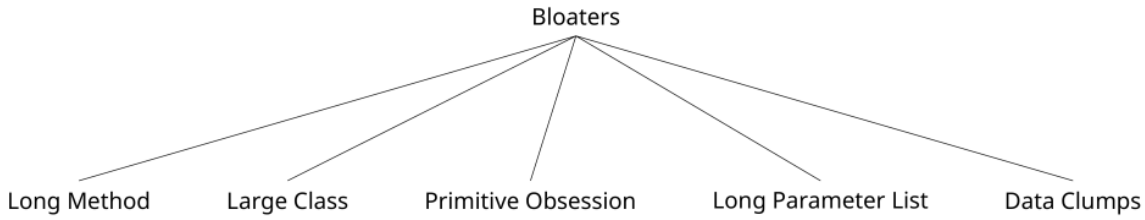

Figure A.1: Different types of Bloaters

- Long Methods : A method which has too many lines.
- Large Class : A class with too much of lines of codes. It contains lot of variables and methods.
- Primitive Obsession : Primitives are used in the code more often rather than grouping them into meaningful classes.
- Long Parameter List : A method which accepts lot of arguments. An idea method will have maximum of 3-4 parameters.
- Data Clumps : There can be code segments which has similar set of variables which need to be grouped into its own classes.

A.2 Object-Orientation Abusers

Improper implementation of object oriented principles are the reason for Object-Orientation Abusers

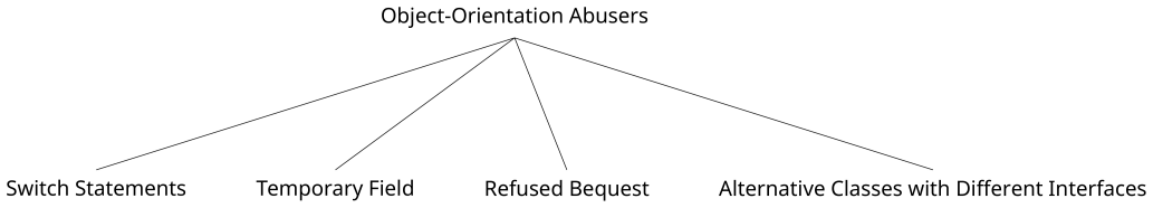

Figure A.2: Different types of Object-Orientation Abusers

- Switch Statements : Having complicated switch statements or if-else statements.
- Temporary Field : The values of these fields are set only under certain circumstances.
- Refused Bequest : A subclass inherits members from its super class, but what is actually needed is a very little of it.
- Alternative Classes with Different Interfaces : There are two different classes which functions identically, but they have different names.

A.3 Change Preventers

These smells indicate that we will have to change multiple places due to a change done in one place.

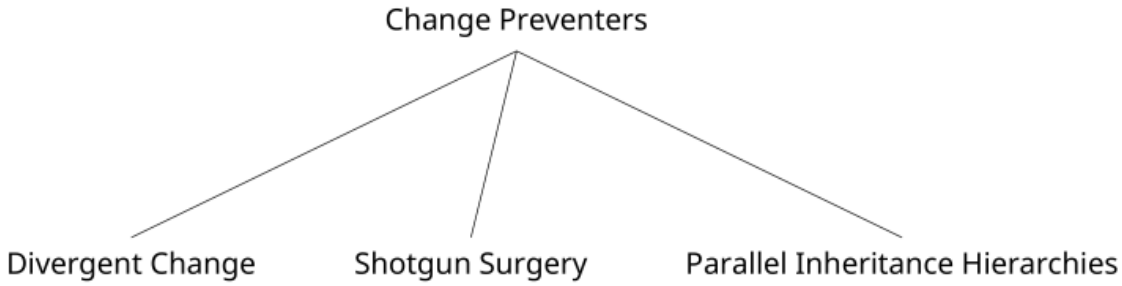

Figure A.3: Different types of Object-Orientation Abusers

- Divergent Change : Having to change many unrelated methods due to a change done in one class.
- Shotgun Surgery : Making a modification leads to make small modifications to many different classes.
- Parallel Inheritance Hierarchies : In order to create a subclass of a class we will have to create a subclass of another.

A.4 Dispensables

Dispensables are useless to have in the code and without them the code will be much more cleaner and maintainable.

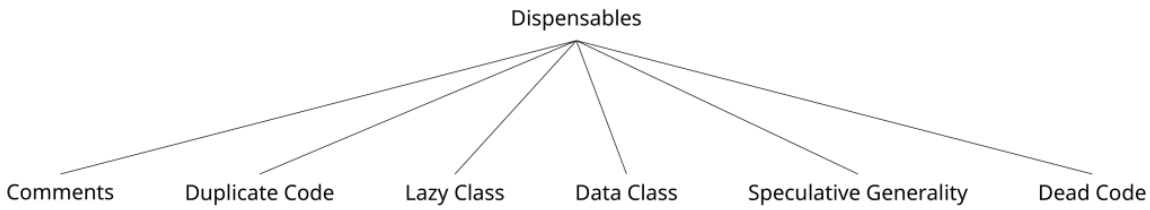

Figure A.4: Different types of Dispensables

- Comments : Filling a method or class with too much of unnecessary comments.
- Duplicate Code : Two code segments which are identical
- Lazy Class : A class does not seem to be doing enough to consider it as a standalone class.
- Data Class : Classes with only fields, getters and setters, but no methods.
- Speculative Generality : There is an unused class, method, field or parameter.
- Dead Code : A variable, method or class which is no longer used or referred in the code.

A.5 Couplers

As the name indicates tightly bound classes are the reason for these code smells.

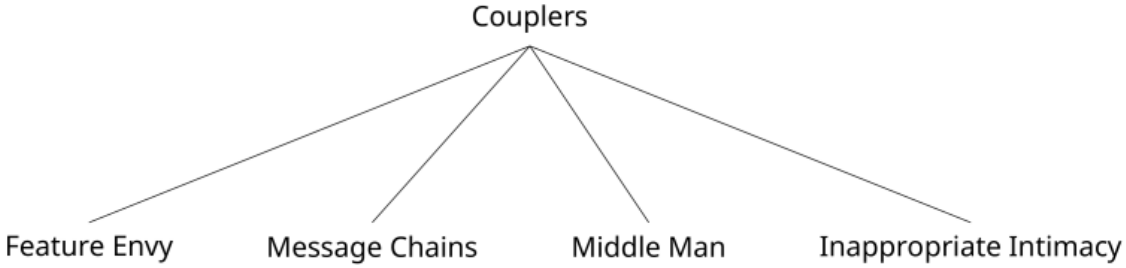

Figure A.5: Different types of Couplers

- Feature Envy : A method refers more features from a class other than the one it attached to.
- Message Chains : One class refers another class which in turns refers to another class thus creating a message chain.

- Middle Man : The sole purpose of this class is to delegate tasks to other class and does not perform its own task.
- Inappropriate Intimacy : One class uses internal fields and methods of another class.

Appendix B

Environment Setup

The bug prediction model in this study has been developed using Python 3. We have used Jupyter Notebook as our programming environment. There are numerous ways to setup the environment.

B.1 Installing Jupyter using Anaconda and conda

Anaconda installs Python, the Jupyter Notebook, and other commonly used packages for scientific computing and data science easily. Therefore installing Anaconda is the most convenient method.

1. Download the latest Anaconda distribution depending on your operating system from https://www.anaconda.com/download/.
2. Install the executable file by following the instructions.
3. Run Jupyter notebook with the command *jupyter notebook*.

B.2 Installing Jupyter with pip

We can also install Jupyter with the python package manager *pip*.

1. Download and install the latest version (3.X) of Python.
2. Check the pip version through *pip -V*.
3. Run Jupyter notebook with the command *jupyter notebook*.
4. Then install Jupyter Notebook with *pip3 install jupyter*.
5. Run Jupyter notebook with the command *jupyter notebook*.

Appendix C

Evaluation Results

C.1 Basic model within version

C.1.1 Apache Tomcat version 6

Measure	Naive Bayes	Random Forest	Logistic Regression
Accuracy of train data	85.83 %	97.98 %	93.25 %
Accuracy of test data	87.84 %	94.12 %	94.12 %
TP	0.78%	0.39%	0.39%
TN	87.06%	93.73%	93.73%
FP	7.45%	0.78%	0.78%
FN	4.71%	5.10%	5.10%
Precision	0.10	0.33	0.33
Recall	0.14	0.07	0.07
F1 score	0.11	0.12	0.12
Number of True cases in train data	**6.49%**		
Number of False cases in train data	**93.51%**		
Real Data Set	[0, 0, 0, 0, 0]		
NB Prediction	[1, 0, 0, 0, 0]		
RF Prediction	[0, 0, 0, 0, 0]		
LR Prediction	[0, 0, 0, 0, 0]		

Table C.1: Basic model within version Apache Tomcat 6

C.1.2 Apache Xalan version 2.7

Measure	**Naive Bayes**	**Random Forest**	**Logistic Regression**
Accuracy of train data	95.07 %	100.00 %	98.57 %
Accuracy of test data	98.89 %	99.26 %	98.52 %
TP	98.89%	98.89%	98.52%
TN	0.00%	0.37%	0.00%
FP	1.11%	0.74%	1.11%
FN	0.00%	0.00%	0.37%
Precision	0.99	**0.99**	0.99
Recall	1.00	**1.00**	1.00
F1 score	0.99	**1.00**	0.99
Number of True cases in train data	**98.78%**		
Number of False cases in train data	**1.22%**		
Real Data Set	[1,1,1,1,1]		
NB Prediction	[1,1,1,1,1]		
RF Prediction	[1,1,1,1,1]		
LR Prediction	[0, 0, 0, 0, 0]		

Table C.2: Basic model within version Apache Xalan 2.7

C.2 Enhanced model with code smell based metrics within version

C.2.1 Apache Xalan version 2.7

Measure	Naive Bayes	Random Forest	Logistic Regression
Accuracy of train data	95.07 %	100.00 %	98.89 %
Accuracy of test data	98.89 %	99.26 %	98.89 %
TP	98.89%	98.89%	98.89%
TN	0.00%	0.37%	0.00%
FP	1.11%	0.74%	1.11%
FN	0.00%	0.00%	0.00%
Precision	0.99	**0.99**	0.99
Recall	1.00	**1.00**	1.00
F1 score	0.99	**1.00**	0.99
Number of True cases in train data	**98.78%**		
Number of False cases in train data	**1.22%**		
Real Data Set	[1,1,1,1,1]		
NB Prediction	[1, 1,1,1,1]		
RF Prediction	[1,1,1,1,1]		
LR Prediction	[1,1,1,1,1]		

Table C.3: Enhanced model within version Apache Xalan 2.7

C.3 Basic model within project

C.3.1 Apache Camel

Measure	**Naive Bayes**	**Random Forest**	**Logistic Regression**
Accuracy of train data	76.04 %	96.07 %	79.73 %
Accuracy of test data	77.66 %	78.57 %	79.12 %
TP	6.96%	5.68%	1.83%
TN	70.70%	72.89%	77.29%
FP	8.79%	6.59%	2.20%
FN	13.55%	14.84%	18.68%
Precision	0.44	0.46	0.45
Recall	0.34	0.28	0.09
F1 score	0.38	0.35	0.15
Number of True cases in train data	**20.56%**		
Number of False cases in train data	**79.44%**		
Real Data Set	[0,0,0,1,1]		
NB Prediction	[0,0,1,1,0]		
RF Prediction	[1,0,0,1,0]		
LR Prediction	[0,0,0,0,0]		

Table C.4: Basic model within project Apache Camel

C.3.2 Apache Log4j

Measure	Naive Bayes	Random Forest	Logistic Regression
Accuracy of train data	82.35 %	96.47 %	85.29 %
Accuracy of test data	83.78 %	82.43 %	85.14 %
TP	13.51%	13.51%	13.51%
TN	70.27%	68.92%	71.62%
FP	6.76%	8.11%	5.41%
FN	9.46%	9.46%	9.46%
Precision	0.67	0.62	0.71
Recall	0.59	0.59	0.59
F1 score	0.62	0.61	0.65
Number of True cases in train data	**29.10%**		
Number of False cases in train data	**70.90%**		
Real Data Set	[1,1,1,1,1]		
NB Prediction	[1,1,1,1,0]		
RF Prediction	[1,1,1,1,0]		
LR Prediction	[1,1,1,1,0]		

Table C.5: Basic model within project Apache Log4j

C.4 Enhanced model with code smell based metrics within project

C.4.1 Apache Camel

Measure	Naive Bayes	Random Forest	Logistic Regression
Accuracy of train data	79.73 %	100.00 %	83.82 %
Accuracy of test data	80.77 %	83.70 %	85.35 %
TP	8.79%	5.31%	6.96%
TN	71.98%	78.39%	78.39%
FP	7.51%	1.10%	1.10%
FN	11.72%	15.20%	13.55%
Precision	0.54	0.83	0.86
Recall	0.43	0.26	0.34
F1 score	0.48	0.39	0.49
Number of True cases in train data	**20.56%**		
Number of False cases in train data	**79.44%**		
Real Data Set	[0,0,0,1,1]		
NB Prediction	[0,0,1,1,0]		
RF Prediction	[0,0,0,0,0]		
LR Prediction	[0,0,0,1,0]		

Table C.6: Enhanced model within project Apache Camel

C.4.2 Apache Log4j

Measure	**Naive Bayes**	**Random Forest**	**Logistic Regression**
Accuracy of train data	84.12 %	100.00 %	93.53 %
Accuracy of test data	82.43 %	81.08 %	79.73 %
TP	10.81%	4.05%	8.11%
TN	71.62%	77.03%	71.62%
FP	5.41%	0.00%	5.41%
FN	12.16%	18.92%	14.86%
Precision	0.67	1.00	0.60
Recall	0.47	0.18	0.35
F1 score	0.55	0.30	0.44
Number of True cases in train data	**29.10%**		
Number of False cases in train data	**70.90%**		
Real Data Set	[1,1,1,1,1]		
NB Prediction	[1,1,1,1,1]		
RF Prediction	[1,1,1,1,1]		
LR Prediction	[1,1,1,1,1]		

Table C.7: Enhanced model within project Apache Log4j