



S	
E1	
E2	
For Office Use Only	

Masters Project Final Report
(MCS)
2019

Project Title	Predicting Security Vulnerable Developers on Behavioral Characteristics
Student Name	R. M. S. V. Rathnayaka
Registration No. & Index No.	2014/MCS/065
Supervisor's Name	Dr. J. S. Goonathilake

For Office Use ONLY



Predicting Security Vulnerable Developers on Behavioural Characteristics

**A dissertation submitted for the Degree of Master of
Computer Science**

R. M. S. V. Rathnayaka

University of Colombo School of Computing

2019



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: R. M. S. V. Rathnayaka

Registration Number: 2014/MCS/065

Index Number: 14440654

Signature:

Date:

This is to certify that this thesis is based on the work of

Mr. R. M. S. V. Rathnayaka

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Dr. J. S. Goonathilake

Signature:

Date:

Acknowledgement

Working on a research study would be a hard time for a student with a work-life balance. It will get harder if there's not enough support from the peoples around. But, I was lucky enough to have a bunch of supporters around me.

Among them; first I would like to convey my heartfelt gratitude to my supervisor Dr. Jeevani Goonathilake for her guidance, motivation provided me on my research studies and all the support given during the difficult times. Without her all support and feedback; I couldn't imagine completing my research works successfully.

I'd like to remind my dearest parents who always wished me every success in life with their unconditional love. I'm always happy to be their pride, with each and every success that I made in my life. Thus, I let this achievement also to become a reason to make them proud.

At last but not least, I'd like to convey my sincere gratitude to all my friends who have supported me to realize my dream milestone in academia.

Table of Contents

List of Abbreviations	7
List of Figures.....	8
List of Tables	9
List of Experiments.....	9
Chapter 01: Introduction	10
1.1 Problem Identification	10
1.2 Motivation.....	12
1.3 Goals and Objectives.....	14
1.4 Scope	15
1.5 Limitations and Constraints	15
Chapter 02: Related Work and Fundamentals	17
2.1 Related Works.....	17
2.2 Related Work Summary	26
2.3 Security Vulnerabilities	27
2.4 Developer-Centric Security Vulnerabilities	28
2.5 Commit Log.....	29
2.6 Feature Metrics	30
2.7 Feature Subset Selection (FSS).....	30
2.8 Class Balancing.....	32
2.9 Prediction Models	33
2.10 Performance Evaluation.....	35
2.11 Error Analysis	39
Chapter 03: Problem Analysis and Methodology.....	41
3.1 Candidate Feature Vector	41
3.2 Repository Selection for Dataset.....	48

3.3 Data Extraction	51
3.4 Data Set.....	57
3.5 Data Set Preprocessing	59
3.6 Feature Selection	61
3.7 Prediction Model Selection.....	62
3.8 Training	67
Chapter 04: Developer Vulnerability Prediction and Evaluation	69
4.1 Benchmark	69
4.2 All Feature Vector Performance	70
4.3 Performance through Feature Selection	70
4.4 Performance through Class Balancing	74
4.5 Ensemble Techniques	76
4.6 Experiment Summary	77
Chapter 05: Discussion	80
5.1 Summary.....	80
5.2 Findings	81
5.3 Future Work.....	91
References	93
Appendix	96

List of Abbreviations

TP - True Positive

TN - True Negative

FP - False Positive

FN - False Negative

FI - File Inspection Ratio

PF - Probability of False Alarm

OWASP - Open Web Application Security Project

CWE - Common Weakness Enumeration

FSS - Feature Subset Selection

SVM - Support Vector Machine

HTML - Hyper Text Markup Language

XML - eXtended Markup Language

JSON - Javascript Object Notation

List of Figures

Figure 1.1: Transformed Feature vs Developer Feature

Figure 2.1: ROC curve behavior against confusion matrix

Figure 2.2: Detailed Structure of a Commit Log

Figure 2.3: ROC curve behavior against confusion matrix

Figure 2.4: Different ROC curves and respective classification distributions

Figure 2.5: Bias-Variance Tradeoff

Figure 3.1: Top Active Languages in GitHub

Figure 3.2: Output Data Set Format of Jmine Analyzer

Figure 3.3: Developer Network of Metrics GitHub Project

Figure 3.4: Sample set of security vulnerability rules

Figure 3.5: Overall SonarQube Assessment of Netty GitHub Project

Figure 3.6: Identified Vulnerabilities of Netty GitHub Project

Figure 3.7: Exported Sample JSON Data Set of Vulnerable Developers

Figure 3.8: Feature Selection Approach Diagram

Figure 3.9: Class distribution histogram of collected dataset

Figure 3.10: Feature Selection Approach Diagram

Figure 3.11: Prediction Model Selection Approach Diagram

Figure 3.12: Random Forests Algorithm

Figure 3.13: SVM for 2-dimensional Feature Space

Figure 3.14: K - Fold Cross Validation

Figure 4.1: RUS Class Balanced Dataset Statistics in WEKA

Figure 4.2: SMOTE Class Balanced Dataset Statistics in WEKA

Figure 4.3: ROC curve of Zero - R Classifier

Figure 4.4: ROC curve of Random Forest Classifier

Figure 4.5: ROC curve of Naive Bayes Classifier

Figure 5.1: Number of Rows Distribution

Figure 5.2: Average Commit Interval Distribution

Figure 5.3: Frequent Commit Hour Distribution Histogram

Figure 5.4: Lines Inserted Distribution

Figure 5.5: Lines Deleted Distribution

Figure 5.6: Developer Age Distribution

Figure 5.7: Average Cyclomatic Complexity Distribution

Figure 5.8: Comment Percentage Distribution

Figure 5.9: Developer Closeness Distribution

List of Tables

Table 2.1: Related works on defect/vulnerability predictions

Table 2.2: Feature vectors of related studies

Table 3.1: General information about selected Github repositories

Table 3.2: General statistics of complete collected dataset

Table 4.1: Benchmarking classifier performance measures

Table 4.2: Classifier performance measures on raw dataset

Table 4.3: Feature elimination sequence and accuracy

Table 4.4: Classifier performance measures on feature selected dataset

Table 4.5: Classifier performance measures on class balanced dataset

Table 4.6: Classifier performance measures on ensemble techniques

Table 5.1: Feature Vector Correlation Summary

List of Experiments

Experiment 4.1: Classifier Subset Evaluation with Bagging

Experiment 4.2: Random Forest classifier evaluation on feature selected dataset

Experiment 5.1: Generated Decision Tree of Decision Tree Classifier

Chapter 01: Introduction

This chapter exchanges the views on initial backgrounds of this study by identifying the problem that motivated for this study and the ultimate goals and outcomes in finding the solution for the problem identified. Meanwhile, it identifies the possible limitations and constraints that would be affected by this study and defines the scope of this study.

1.1 Problem Identification

Computer software basically satisfies two types of requirements in an application domain. Those are functional and nonfunctional requirements. Both are equally important to be satisfied regardless of their operational industry domain. Functional requirements basically describe how and what the software supposed to accomplish and non-functional requirements which also known as quality requirements describe the constraints of the software design and implementation.

In the modern information era, the software development industry is gaining a drastic improvement since the applications of computer software are wide spreading across many industry domains. This potentiality has moved to the usage of computer software into large scale, mission and life-critical applications in high stake industries such as defense, aerospace, medicine, nuclear power, etc. which involves a higher tradeoff between the cost and the quality of output that the software can produce. Thus, it can be identified that quality is an important factor in many mission and life critical industries to withstand in software solutions.

Although there are many Computer Aided Software Engineering (CASE) tools that are in use, humans still dominate as the core contributors of the software development process. But it's a common and natural argument that humans tend to make mistakes. Thus, it's an inevitably applicable scenario that software is vulnerable to functional and non-functional defects.

In software development projects, defects can be introduced in any phase of the software development lifecycle. But, most of the defects are tend to occur by the time of its development phase [4] and is difficult and introduces huge cost to overcome from them in later stages of software development. Therefore, early prediction of software defects and quality issues has become a vital topic in the modern world software project management domain.

Software quality has been described by many characteristic aspects. There are a number of software quality metrics used to evaluate the qualities of the software such as Scalability, Security, Reliability, Usability, etc. [3] Among all these quality metrics, software security has been described as one of the major quality attribute concerned and it is a high critical quality factor [37] in above mentioned high stake industry domains [2]. Since, a security vulnerability can be a cause of huge disaster which can lose billion dollars of assets [1] and lives [22], [37].

Generally, functional issues of software referred to a defect in software which can be tested and validated by executing the test scenarios of the business logic. But, infringement of non-functional requirements such as software security requirements will lead to security vulnerabilities which are difficult to identify [10], [37] since it may not get exposed by the execution of predetermined functional test scenarios. Thus, it will be even more difficult when predicting the deviations of such nonfunctional requirements.

In general terms, security vulnerability has a broader definition which specifically not limited to software related security bugs (or defects). There can be vulnerabilities that are exploitable due to hardware, environmental and social reasons [37]. But in this study, the term “security vulnerability” is specifically referred to software level security defects that can be exploitable through malicious execution of the software and tools.

Currently, several methods have been suggested in the literature to predict future defects, security vulnerabilities of software components based on software-centric metrics that are extracted from the source code. But the tendency of defective code contribution could happen due to many techno-behavioral factors of each individual developer. These software-centric metrics alone cannot represent the exact human factors of each individual developer which can affect the overall quality of the software.

Further, predicting the potential vulnerabilities of contributing developer profiles is a way ahead approach rather than predicting the security vulnerability of ongoing software projects. Because

if there are set of developers working on a mission-critical system who follow certain behavioral practices that could be contingent to contribute security vulnerable codes. Finally, it can be collectively affected by overall software quality attributes such as software security. Thus, understanding significant techno-behavioral characteristic features of the developer is required to have an insightful and improved prediction on the potentiality of vulnerable code contributions done by each developer.

The remaining chapters of this dissertation will be organized as follows. Chapter 2 outlines the background of this work and the related works have been done in the literature. Chapter 3 analyze the problem and its characteristics. Further, outlines the possible research methodologies to solve identified research problems. In Chapter 4 we discuss the current progress of the research and upcoming tasks scheduled to be executed in the research design.

1.2 Motivation

A software developer is one of the most important human resources in a software development project who creates and maintains software. Wurster, G. et al [32] argues that although software developers are most often considered as allies in software development, it won't be a reliable assumption at all. Each software developer has a unique skill level, experience, capacity, technology interests, domain interests and many other characteristics which can affect the overall quality of the software positively or negatively. Based on these developer's characteristics, their intentional or unintentional contributions such as the implementation of source code and configurations may lead to security vulnerabilities in software. Hereafter this developer characteristic is referred to Developer-Centric Vulnerability (DV) for convenience in this study.

In large scale software development projects, a number of developers could be capable of working parallel while having many interdependencies between each other. Thus, it's important, but a complex task to ensure all developers are skilled and experienced enough to contribute and collaborate for the project avoiding any security vulnerabilities in software throughout the development lifecycle.

So in a social aspect, it will be a challenging task to evaluate and differentiate a developer's tendency to contribute vulnerable software codes. For a highly secured or mission-critical

software project, it will be a complicated task to handle if there are non-functional loopholes such as security vulnerabilities identified in the middle of the software development process. If the developer code contributions haven't been up to the security standards, practices and to avoid security defects; it'll be a difficult situation to manage due to the human resource, institution policies, project timeline and many other technical and non-technical constraints and ultimately it will cost to the software project.

When there is a need for selecting developer teams for a highly secured mission critical system development task, it would have been better if predicted developer-centric vulnerability taken as a parameter into the developer selection criteria of software project since developers with less or no vulnerability prediction will minimize risk of security vulnerabilities of the software at the later stages of SDLC.

To analyze the quality aspects of a software project including security vulnerabilities, there are many tools and techniques. But, is there any straightforward way or measures that have been identified to determine the vulnerability of a particular developer who is contributing to a software project?

Alternatively, the most possible approach to get some idea about individual developer-centric vulnerability is to evaluate the quality features of software through static code analysis and find out the responsible developers who have contributed security wise weak code fragments. But, that process is complex and time-consuming since there are many tools and steps to be followed. Additionally, in order to do that; static code analyzers need to access the code base to initiate the vulnerability analyzing process where sometimes the access might be restricted to the interested party. Thus, it would be more beneficial if there's a way to get a general idea about developer-centric security vulnerability by their behavioral activities done in the software development process while without accessing the code level information about the contributed code base.

According to the literature, several attempts have been made to predict software vulnerabilities from the perspective of software source code. But there are fewer attempts have been made to predict software vulnerabilities considering the behavioral features of its contributing developers which can improve the predicting performance.

Thus, it will be a huge advantage when there is a methodology on predicting possible vulnerabilities that could occur in a software project at each contributing developer's level by analyzing each developer's techno-behavioral features.

Finally, the classifier models that are trained with a better prediction accuracy can be applied to predict the developer-centric vulnerability of any other developer if the identified developer behaviors are given. The most important advantage with that approach is that the developer-centric vulnerability can be evaluated even source code level access to the previous historical software repositories are restricted or not available.

1.3 Goals and Objectives

This research is focusing on identifying a suitable proactive approach for predicting the developers who are contingent to expose security vulnerable source codes by analyzing their behavioral (committing pattern, development collaboration, individual code complexity, etc.) characteristics. The Following components have been addressed within this research.

1. Identifying the most suitable developer characteristic features of the developer which has a higher correlation with security vulnerabilities.
2. Ascertaining related, as well as existing mechanisms in the literature which has been used for software-centric, developer-centric vulnerability prediction.
3. Adopt and modify selected mechanisms for better vulnerability prediction of developers.
4. Evaluate the accuracy and performance of each mechanism against an identified set of developer profiles (through static code analysis and previous research outcomes) who had contributed for security vulnerable codes in a software project.
5. Produce the most accurate methodology in predicting the developers who tend to expose more security vulnerable contributions.

1.4 Scope

1. As described in section 1.1, the software security vulnerabilities that are focused here is limited to the vulnerabilities that get exposed through security defects in the source code of the software while the vulnerabilities that are exploitable by other means are excluded.
2. Software security vulnerabilities can appear in many forms and documented by many forms about them. But, this study only OWASP security measures [6] and CWE exposed security measures are considered in analyzing developers.
3. Github [5] social coding platform will be used as the data source to extract the dataset for identifying developer characteristics on security error-proneness.
4. Although GitHub contains a massive number of repositories, the most significant repositories that have been used and suggested in the literature will be used in the analysis.
5. In the feature selection, all the developer features will be selected through the outcomes of the literature review.
6. Although suggested by the literature review; some of the developer features may not be able to consider due to the unavailability of repository mining tools to extract those feature measures.
7. There is a number of approaches that have been defined in the literature for prediction problems, the most significant mechanisms (maximum 5) will be used for the evaluation.

1.5 Limitations and Constraints

1. The software repositories that are used for this work are sourced from GitHub social coding platform where all the repositories are publicly available to explore. But there are limited opportunities available for validate the prediction models against private repositories.
2. Repository mining for developer behavior features is has its limitations due to less availability of mining tools and weak performance of available tools. Thus, some of the identified developer behaviors may have to be omitted in this study.

3. This work intends to compare the prediction performances of several prediction models that has been proposed in the literature. The number of prediction models to be evaluated in this study has limited to minimum three since there are time constraints to complete this study.

Chapter 02: Related Work and Fundamentals

In this chapter, the literature of the related previous works is compared and analyzed in order to get more understanding of current outcomes, future research works and the improvements to be done on the related domain. This study associated with the research field of data mining and machine learning. Thus, this chapter intends to explore the theoretical fundamentals of the techniques which have been applied in this research work.

2.1 Related Works

As per the reviewed literature, many author's works on this domain of vulnerability prediction has been focused on predicting the vulnerabilities of software or software components based on the features directly extracted from source code which also known as product (software) metrics. Further, most of the other related works have been focused on software defect prediction. But software vulnerability is a very specialized defect type which is more difficult to identify. Many related works here having the focus of identifying the predictor model from the perspective of software while this study mainly focusing on identifying the predictor model for vulnerabilities from the developer's perspective.

Ostrand, T.J. et al [12] looking forward to continue and improve their preliminary works on defect prediction models by incorporating developer-specific information. They have defined a bug ratio for each individual developer and compared each individual bug ratio to the average bug ratio and assessed the consistency of the bug ratios across software releases. They have achieved minimal improvement on bug prediction performances which aren't significant with respect to the actual bugs that individual developers have committed later. They claim that, since their study was limited to one single software repository and bug ratio is the one and only feature considered to predict defects; their outcomes on developer-centric defect prediction may not hold widely.

Going further Jiang, T et al [11] have proposed personalized defect prediction models by utilizing three types of feature categories which are characteristic vectors, bag-of-words, and metadata. In the software repository version controlling domain, a developer commit is referred to the recent source code changes that a developer pushed to the main software repository in order to make them persist in the software repository. Single developer commit may contain several code changes in one or more source files with one or more lines of code changes. The characteristic vector represents the syntactic structure of a developer commit; which is a mathematical representation of a developer commit (a code fragment) by counting the number of syntax type nodes (eg: if, for, while) in the Abstract Syntax Tree (AST). Bag-of-words is a vector that represents the occurrence count of keywords of a developer commit. Additionally, they utilize the metadata features in each developer commits such as commit hour, commit day, cumulative change count, cumulative buggy change count, source code file/pathnames, and file age in days. However, Jiang, T et al [11] concludes that this research work scope didn't intend to identify the best feature combination for defect prediction while confirming the importance of identifying the better feature vector combination for improved defect prediction performances.

The outcomes of this work ensure the significance of using developer behavior related metrics on defect prediction models. It has been rectified by the research work of Matsumoto, S et al [33] confirming the importance and the effectiveness of developer metrics utilization on developer-centric defect predictions.

Jiang, T et al [11] further confirms that the approach of personalization idea has gained many successes at different domains and the idea could be successfully applied for related domains such as personalized vulnerability prediction. This idea has been facilitated by the research contributions done by Shin, Y et al [7] and Zimmermann, T et al [10] in the evaluation on possibilities of utilizing defect prediction models trained by the classical software component metrics to predict the vulnerabilities in software components. The main objective of this research works focused on whether software defect prediction models also can be used to predict software vulnerabilities.

In the research work done by Shin, Y et al [7]; they have identified a set of traditional software metrics that are used in defect prediction modeling through literature and produced defect prediction models. Then, by utilizing the produced defect prediction models they have evaluated the vulnerability prediction performances of each source code file. The research

outcome of this study proved the hypothesis with better performance with 90% of recall measure. However, the precision measure of this study was very low. Thus, they have emphasized the identification of specialized feature metrics on software vulnerability prediction in order to achieve improved performances.

Similarly, the research work done by Zimmermann, T et al [10] followed the same approach to predict software component vulnerabilities of Windows Vista project by identifying 36 empirical software and organizational metrics that are broadly categorized into 5 categories. They have identified 30 out of 36 metrics are statistically significant when considering the correlation with the software vulnerabilities. As mentioned above; other than the software metrics they have considered measures of organizational structure as an input for prediction modeling for software vulnerabilities. As per the further analysis done by them; they have found there were several organizational feature metrics that are found to be having a significant correlation with software vulnerabilities. Addition to that, they have done another separate method to evaluate vulnerability prediction performances using the software library dependencies that have been used in particular software component. Here, they have defined a high-dimensional bit vector as the feature vector for their prediction models. Each bit in the bit vector represented a dependent software library where value 1 represented the particular library (dependency) has been used and value 0 represented the library hasn't used for the particular software component. However, the approach on predicting software vulnerabilities using software dependencies doesn't draw much attention in the scope of this study. But their work on software vulnerability prediction using classical software metrics approach further confirms the claims made by Shin, Y et al [7].

On the extensive work done by Shin, Y et al [8] to identify the indicators of software vulnerability, they have considered some specialized feature metrics of three categories. They have defined complexity, code churn and developer activity as the main categories for their evaluation and identified many feature metrics under them. Their evaluation compared two software projects and confirmed 24 out of 28 feature metrics have been supporting their hypothesis. According to them, code churn and developer activity categories have the most significant prediction performances and code complexity is the weakest performing predictor category.

According to research work done by Meneely, A. et al [34] investigating the way of developer collaboration has a significant correlated effect on the software defects. They have modeled

collaboration and developer network between contributing developers and source files in the repository and utilized network analysis mechanisms to derive developer features such a connectivity and centrality measures. But, in order to predict software defects in source files, these developer features need to be compatible with other software metrics based on source code files. Thus, they transform developer features into software features using several techniques before they apply them on file (software) based feature vector to train the models.

Instead of using software metrics or developer metrics Walden, J et al [9] proposed a novel approach for vulnerability prediction using text mining methods. There, the source code sent through a text mining process and tokenized into a vector of monograms where each monogram is followed by a count. This monogram vector is used as the feature vector in their study to compare with various prediction models and has obtained similar and higher performance than the other available prediction models. Further, they have done a comparison of prediction capabilities between text mining and software metric methods by introducing 12 software feature metrics.

But this text mining method does not help to draw any meaningful characteristics of individual developers. The text mining method involves the direct analysis of source code to extract the monogram vector data. According to the goals and objectives of this study feature metrics obtained through direct source code analysis is not desirable.

Study	Predicts For	Perspective	Technique
Ostrand, T.J. et al [12]	Defect	Developer	Developer Features
Jiang, T et al [11]	Defect	Developer	Software + Developer Features
Matsumoto, S et al [33]	Defect	Developer	Developer Features
Meneely, A. et al [34]	Defect	Software	Software Features (Transformed Developer Features)
Shin Y, et al [7]	Vulnerability	Software	Software Features
Zimmermann, T et al [10]	Vulnerability	Software	Software Features + Organizational

Shin Y, et al [8]	Vulnerability	Software	Software Developer (Transformed) Features +
Walden, J et al [9]	Vulnerability	Software	Monogram Vector

Table 2.1: Related works on defect/vulnerability predictions

As described in Table 2.1, there are several attempts have been made to develop prediction models for software vulnerabilities. In most instances, software related metrics selected as the core predictor variables in prediction models. Additionally, there are instances that employ different methods such as features of developer activity and tokenized vector of source codes as predictor variables of vulnerability prediction models. In most of the works reviewed, the attempts have been made to identify predictor variables in order to predict vulnerable software components while this study focuses on predicting vulnerable developers. Thus, further research work is required to derive the most suitable feature set for this study and analyze their discriminative power and correlation in developer-centric vulnerability prediction.

Study	Feature Type	Feature
Ostrand, T.J. et al [12]	Developer	Bug Ratio
Jiang, T et al [11]	Software	Characteristic Vector
		Word Vector
		File Age
		Source code file/path names
	Developer	Commit Hour
		Commit Day
		Cumulative Change Count
		Cumulative Buggy Change Count
Matsumoto, S et al [33]	Developer	Number of Commitments

		Number of Lines Revised
		Number of Unique Modules Revised
		Number of Unique Packages Revised
		Average Number of Faults Injected By Commit
		Number of Developers Revising Module
		Lines of Code Revised By a Developer
Meneely, A. et al [34]	Software (Transformed Developer Features)	Code Churn
		Number of Updates
		Number of Developers
		(Sum/Average/Max) of Degree
		(Sum/Average/Max) of Closeness
		(Sum/Average/Max) of Betweenness
		Number of Hub Developers
Shin Y, et al [7]	Software	Lines of Code Count
		Lines of Variable Declarations
		Count of Function Declarations
		Essential Complexity (Avg, Sum)
		Number of Conditional Statements (Avg, Sum, Max)
		Control Structure Nesting Level (Avg, Sum, Max)
		Comment Density

		Number of Inputs to a Function (Avg, Sum, Max)
		Number of Assignments to Parameters (Avg, Sum, Max)
		Number of Changes (Commits)
		Lines Changed
		Lines Inserted
		Lines Deleted
		Lines New
		Number of Prior Faults
Zimmermann, T et al [10]	Software	Total Churn
		Frequency
		Repeat Frequency
		Cyclomatic Complexity (Max, Total)
		Fan-In (Max, Total)
		Fan-Out (Max, Total)
		Lines of Code (Max, Total)
		Weighted Methods Per Class (Max, Total)
		Depth of Inheritance (Max, Total)
		Coupling between Objects (Max, Total)
		Number of Subclasses (Max, Total)
		Total Global Variables

		Incoming Direct
		Incoming Closure
		Outgoing Direct
		Outgoing Closure
		Layer Information
		Block Coverage
		Arc Coverage
	Organizational	Number of Engineers
		Number of Ex-Engineers
		Edit Frequency
		Depth of Master Ownership
		Percentage of Org Contributing to Development
		Level of Organizational Code Ownership
		Overall Organization Ownership
	Organization Intersection Factor	
Shin Y, et al [8]	Software (In addition to Shin Y, et al [7]) (Transformed Developer Features)	Consolidated Developer Network Degree Value of File (Avg, Sum, Max)
		Consolidated Developer Network Closeness Value of File (Avg, Sum, Max)
		Consolidated Developer Network Betweenness Value of File (Avg, Sum, Max)
		Consolidated Developer Network Edge Betweenness Value of File (Avg, Max)

		Consolidated Contribution Network Closeness Value of File
		Consolidated Contribution Network Betweenness Value of File
Walden, J et al [9]	Software	Monogram Vector

Table 2.2: Feature vectors of related studies

Many authors have been used a number of feature metrics in predicting software defects in the literature. As identifiable through the comparison of previous works in this domain, none of the research work has been focused topic of this study which is developer-centric vulnerability prediction. But, several authors [8], [11], [12], [33], [34] outlined that a set of developer-centric features used in defect prediction models also can be effectively utilized to do predictions on vulnerabilities.

Further, it's a notable fact that some authors have been transforming developer features into software features and achieve positive outcomes in their results. In most of the authors outlined earlier; the main goal is to predict the vulnerability of each software component (modules, packages or files) are the reason for the transformation of these developer features into software features.

This transformation has achieved through accounting developer features separately for each and every software component. Figure 1.1 shows that the same number of commits feature can be considered in different aspects. For a file, a number of commits affected can be considered as a good software feature and meanwhile number of commits done by a developer would be an important feature for a developer. For example, Shin T, et al [8] has considered all these developer and contribution network related centrality measures specific to each and every source code file. So, those features are considered as a feature of the source code file but not as an overall feature of the developer who has contributed.

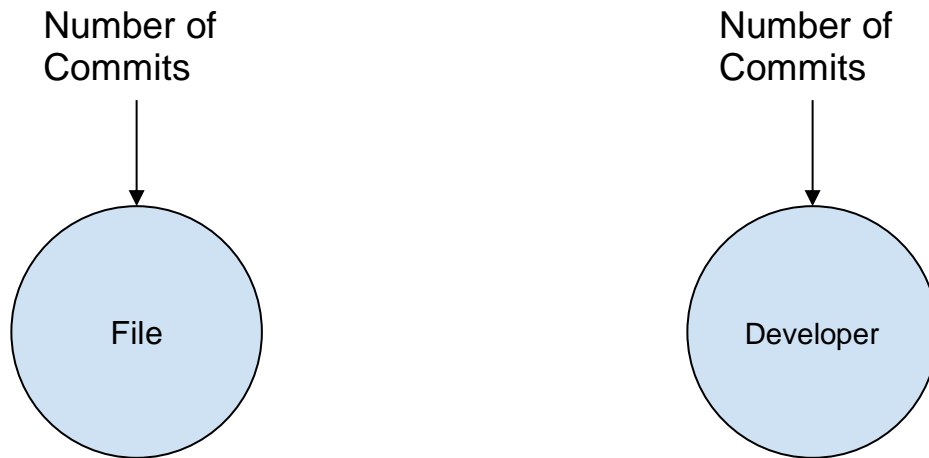


Figure 2.1: Transformed Feature vs Developer Feature

Thus, it can be assumed that this transformation can be applied vice versa to gain positive results in developer-centric vulnerability predictions. It has shown that the feature metrics that are used in software defect prediction can be used in vulnerability prediction since both defects and vulnerabilities have many similarities.

2.2 Related Work Summary

According to the reviewed literature; the attempts have been made on three different perspectives so far.

1. Defect Prediction - Software Perspective
2. Vulnerability Prediction - Software Perspective
3. Defect Prediction - Developer (Personalized) Perspective

Thus, it is clearly identifiable that there's less or no attempt has been done towards identifying vulnerability prediction methods from a developer's perspective. As outlined in the previous sections, the prediction of security and functional defects from a software perspective is possible when there is an ongoing software project which can be accessible to the source code. However, that doesn't provide a clear picture of how the individual developer contribution relates to the functional or security defects that can occur in the software.

But, methods of predicting the individual functional and security defect-proneness of developers (developer perspective) can produce better foresight about the software project risks at the early stages of software development projects.

When predicting software defects or vulnerabilities, most of the work has been focused on predicting the future potentiality of the same software components with respect to its software defect or vulnerability measures. But predicting the developer-centric vulnerability of developer is all about predicting a personalized characteristic based on the techno-behavioral practices that have been recorded in the past software project contributions.

Therefore, to extract software feature metrics; it always needs to have access to the source code of the software component. But developer features or personalized software features are always extracted through mining the past developer contribution interaction records (commits) of the associated version controlling system in the software project. Thus, in order to extract developer features; accessibility to the software code is not required.

2.3 Security Vulnerabilities

A software security vulnerability can be described as a kind of an error in a software specification, implementation or configuration which allows to exploit its security policies at the execution [7]. Specifically, the security vulnerabilities in the software implementation are considered as developer-centric vulnerabilities. Execution of a vulnerable implementation may provide functionality that is beyond its expected functionality by enabling attackers to exploit the defined policies and business logic in software.

Software defects and vulnerabilities have many similarities since both are incurred due to human mistakes. But vulnerabilities differ from defects since they are actively observed by the attackers with malicious and criminal intent while defects are exposed through the valid use cases of its functional usage.

2.4 Developer-Centric Security Vulnerabilities

The vulnerability of a software application could occur at any stage of software development life cycle. Vulnerabilities may be introduced due to invalid requirement specification, weak architectural designs, weak and vulnerable implementation techniques, algorithms and weak test scenarios executed and so on. In this study, the focus is scoped on developer-centric vulnerabilities which can be defined as the weaknesses of software developer implementations. Those have been identified as the most possible cause of security exploitations against the software security measures which has been defined by its functional and nonfunctional requirements.

Open Web Application Security Project (OWASP) is an online community, a nonprofit organization that provides documentation, methodologies, tools and technologies in the field of security in web applications. OWASP is famous for its top 10 developer-centric security vulnerabilities published at each year. In most situations, this OWASP top 10 security vulnerabilities has become the de facto checklist to ensure application security.

Common Weakness Enumeration (CWE) is another community project which aims to create and maintain a catalog of software weaknesses and vulnerabilities. It's also a community developed list for common security weaknesses in software. It has served as a common baseline for security weakness identification, mitigation and prevention efforts. According to CWE; they have documented about 700 identified weaknesses. But there's no such tool that can identify all of them. Thus, CWE has presented and a short list of top 25 issues which is known as CWE/SANS top 25 which contains mostly widespread and critical weaknesses.

In most of the security assessments, both OWASP top 10 and CWE top 25 lists have been used as the benchmarking guidelines and there are many tools has been developed to analyze software source codes by adopting OWASP and CWE as the main rule set.

2.5 Commit Log

Commit log is the detailed informational log in Git VCS which records all the changes that has done by each and every developer in the repository by reverse chronological order. This log is a very useful data source in this study since there are many rich data contents available related to developer's techno-behavioral features.

```
From 3d1b168c8235767d8070c299679b75ed55a7360a Mon Sep 17 00:00:00 2001
From: Supun Viraj <rathnaviraj@gmail.com>
Date: Wed, 1 Aug 2018 21:32:16 +0530
Subject: [PATCH] Commit Hour Frequency Issue Fix

---
src/main/java/com/rmsv/jmine/dto/Developer.java | 13 ++++++----
1 file changed, 9 insertions(+), 4 deletions(-)

diff --git a/src/main/java/com/rmsv/jmine/dto/Developer.java b/src/main/java/com/rmsv/jmine/dto/Developer.java
index c2c68e7..2744089 100644
--- a/src/main/java/com/rmsv/jmine/dto/Developer.java
+++ b/src/main/java/com/rmsv/jmine/dto/Developer.java
@@ -16,7 +16,7 @@ public class Developer implements Comparable {
    private PersonIdent data;
    private Integer encounters = 1;
    private Integer changeFrequency = 1;
-   private List<Integer> commitHours;
+   private Integer[] commitHours = new Integer[24];

    public Developer(PersonIdent data){
        this.data = data;
@@ -25,7 +25,7 @@ public class Developer implements Comparable {
        OBJECT_COUNT++;
        this.id = OBJECT_COUNT;
        DEVELOPERS.put(this.data.getName(), this);
-       commitHours = new ArrayList<>();
+       commitHours = new Integer[24];
    }

    public static Developer getDeveloper(String name){
@@ -61,12 +61,17 @@ public class Developer implements Comparable {
    }

    public Developer setCommitHours(int hour){
-       this.commitHours.add(hour);
+       this.commitHours[hour]++;
        return this;
    }

    public Integer getMostFrequentCommitHour(){
-       return Collections.max(this.commitHours);
+       int maxAt = 0;
+
+       for (int i = 0; i < this.commitHours.length; i++) {
+           maxAt = this.commitHours[i] > this.commitHours[maxAt] ? i : maxAt;
+       }
+       return maxAt;
    }

    public void setChangeFrequency(Integer changeFrequency) {
--
2.10.5
```

Figure 2.2: Detailed Structure of a Commit Log

Figure 2.2 shows a sample structure of a recorded commit log details in Git Version Controlling System (VCS). Not only are this but also there many other details captured by the VCS. Thus,

these repository mining tools are capable of capturing the statistical data about the features by analyzing through the commit logs even access to relate software components are not available or restricted.

2.6 Feature Metrics

A feature metric is a numerical representation of particular object property. In classification models, set of features which is known as feature vector used as variables of classification. Literature suggests that it's an important factor to have fine grained feature vector in order to model better performing classifier. Classifiers are the main component used in prediction models of this research work. Thus, identification of features with higher predictability with vulnerable developers is important concern.

2.7 Feature Subset Selection (FSS)

A number of features in a data set are known as the dimensionality of the data set. Many studies on prediction models report that using high dimensional data set for prediction doesn't increase the performance at all [13], [14]. Instead, performance will start to decline after an optimal amount of feature set since many features can contain noise and redundant information. Having a data set with a large set of features introduces many other problems such as complexity, higher computational overhead on training. Therefore, an optimum set of features needs to be identified in order to provide the maximum performance gain from prediction models and simplicity. Thus, there are techniques to select the best subset of features from a large feature set available in a prediction problem. There are two main approaches discussed under feature subset selection in machine learning domain which known as filter methods and wrapper methods [17].

2.7.1 Filter Methods

Filter methods are focusing on identifying the relevance of the feature based on univariate statistical measurements. They are independent of the learning algorithm and may apply in the data preprocessing phase. These methods are computationally faster and simple, but the

resulting feature subset may not be effective on the selected learning model. Below outline some filter methods that are used for FSS.

- Information Gain
- Chi-square Test
- Fisher Score
- Correlation Coefficient
- Variance Threshold

2.7.2 Wrapper Methods

Wrapper methods focus on solving the real problem by repeated optimizations of the learning model starting with all the feature sets and later selecting the optimal feature subset based on the learning model performances. Since there are many repeated learning steps and cross-validations in wrapper methods, it is more computationally expensive than filter methods. Below outline some wrapper methods that are used for FSS.

- Recursive Feature Elimination
- Sequential Feature Selection Algorithms
- Genetic Algorithms

2.7.3 Summary

The techniques used for feature selection are in a wide range and specific based on many criteria and no known technique uniformly performs well in all feature selection problems. Under this domain of vulnerability prediction Shin Y, et al [7] have used information gain ranking technique and correlation based greedy feature selection technique to select the most suitable feature vector and have obtained similar prediction performances by both techniques. Even though the wrapper methods are computationally expensive than filter methods it employs many learning steps and cross-validations. Since this study having a lesser feature vector dimension; the computational complexity problem will be less significant. Thus, the usage of the wrapper method would be the most suitable approach for this study on feature selection.

2.8 Class Balancing

In real-world classification problems, the datasets that are used to train classifiers suffer from a common weakness known as class imbalance problem. A dataset identified as an imbalanced dataset if there are data points belongs to one class more than the other classes. This causes serious negative effects on trained classifiers such as impair its best possible performances and tendency to over fit towards the majority classes. To deal with this problem there are two approaches suggested in the literature. The dataset imbalance problem either can be resolved in the algorithmic level or data level.

2.8.1 Algorithmic Level Approach

On algorithmic level, this can be resolved by applying some modifications to the generic algorithms. But these modifications are specific to each classification algorithm [25], [26] such as adjusting probabilistic estimates, decision thresholds and many other modifications to minimize the effect on class imbalance [24].

2.8.2 Data Level Approach

At the data level, this is solved by employing many different re-sampling techniques to balance the dataset classes. Addressing this problem on data level is the generic and most commonly practiced method which attempts to balance the dataset classes artificially through some techniques such as oversampling and undersampling [23].

Further, there are many data level improved derivatives of sampling techniques has been proposed on class balancing such as Random Over Sampling (ROS), Random Under Sampling (RUS), Synthetic Minority Oversampling Technique (SMOTE) [27], Condensed Nearest Neighbor Rule, One-Sided Selection (OSS). Etc.

2.8.3 Summary

Although there are many applicable versions to avoid the dataset imbalance problem, there are inherent problems in each approach. For example, oversampling techniques are known for overfitting problems due to artificially generated data points and undersampling causes for loss

of information by reduction of data points. However, Ganganwar, V. [23] confirms that still the data level solutions are more preferable to solve this problem since there are not many implementations available for all algorithms to handle this problem in algorithmic level.

According to Hulse et al. [28] selection of suitable class balancing technique mostly depends on the domain and dataset characteristics. Meanwhile, Chawla, N. et al. [24] claim that the data imbalance problem is an intrinsic characteristic of some domains such as fraud/intrusion detection, risk management, text classification and medical diagnosis/monitoring. In these classification problems, the most focused intention is to accurate detection of the minority class rather than the most common majority class. Generally, this study also can be identified as a similar variant of the above-mentioned problems where vulnerable developer class can become the minority class while non-vulnerable developer class is the majority. According to the observations of Walden, J. et al [9] the ratio between minor and major class have been around 1:100.

In the related attempts on literature [7], [8], [9] for software defect/vulnerability predictions, random undersampling (RUS) technique has been utilized to solve the class imbalance problem. Therefore, it will be worth enough to evaluate performance measures for both RUS and SMOTE techniques to get better understanding while the ROS method can be overlooked since the study evaluates with SMOTE which is an improved version of the ROS technique.

2.9 Prediction Models

Prediction modeling is a way of creating a systematic method to predict a future condition that could occur by analyzing a set of correlated predictor parameters that can be observed at the present. In the academic and research context there are a number of prediction models have been developed and employed to solve prediction problems. This study also intends to get the use of prediction models to predict vulnerable developers. Thus, it's an important thing to refer the literature to explore what are the prediction models that have been used in the related domains of this study.

There are two main streams in prediction models. Statistical prediction models coming from the mathematical background to solve prediction problems. The second approach has been

developed with the advancement of artificial intelligence (AI) and machine learning (ML) techniques in computing. The models discussed in both of these approaches solve three types of prediction problems. Regression is a supervised prediction model used to predict the value of a continuous dependent output parameter based on a set of independent parameters. Classification is the other method under supervised models used to make predictions on a discrete dependent parameter based on a set of independent parameters. Clustering is an unsupervised method in prediction models which mostly used in data mining to discover a new set of important information and facts from an available dataset.

In the literature, many authors suggest that Logistic Regression is effective as a defect and vulnerability prediction model. Logistic regression is a statistical prediction model that is used to predict on discrete binary parameter. Shin, Y et al [7], [8] employed logistic regression as the main prediction model to evaluate software metrics, code churn and developer activity measures to predict vulnerable software components and achieved good prediction performances. Additionally, they have compared the result outcomes for other effectively used prediction techniques such as Random Forests, Naïve Bayes and Bayesian Network. Most of them have performed the classification in a similar capacity while Naïve Bayes provided higher PD with higher FI than other methods. According to their approach in the logistic regression model a software component considered to be vulnerable if the probability of the output is greater than 0.5 and considered to be clean otherwise. In the study of Zimmermann, T et al [10] to predict software vulnerabilities in windows vista outputs were evaluated with logistic regression and Support Vector Machines (SVM). But it's notable that they have applied SVM only for predicting high dimensional bit vector while logistic regression only applied for classical feature vector with lower dimensionality. In the text mining approach of Walden, J et al [9] to predict software vulnerabilities techniques, they have used the Random Forest machine learning algorithm as the primary classifier and achieved better recall performance in all three case studies evaluated.

Recent studies in the literature, there are two main techniques has been employed as vulnerability prediction models. Logistic regression which is a statistical technique and Random Forests, Naïve Bayes, Bayesian networks and SVM are key machine learning techniques used in vulnerability prediction. It can be notable that these authors have been used different types of prediction algorithms to train the prediction models such as regression algorithms, tree algorithms and Bayesian algorithms. Thus, it would be worth enough to do an

evaluation for each of these algorithms and compare the most performing one on developer-centric vulnerability predictions.

2.10 Performance Evaluation

Performance evaluation describes how well and how accurate a prediction model predicts particular developer is contingent to do security vulnerable codes or not. The classification of developer profile into vulnerable class or non-vulnerable class is a binary classification. There are several approaches suggested in the literature which has been used to evaluate the performance of binary classification problems.

2.10.1 Confusion Matrix

Confusion matrix is the most common technique utilized in measuring the performance of classifiers. This technique simplifies and summarize the classifier results in order to provide a better understanding on its performance. In confusion matrix representation for a classifier results, there are four possible states.

- If the instance is POSITIVE and it is classified as POSITIVE it is counted as TRUE POSITIVE (TP).
- If the instance is POSITIVE and it is classified as NEGATIVE it is counted as FALSE NEGATIVE (FN).
- If the instance is NEGATIVE and it is classified as NEGATIVE it is counted as TRUE NEGATIVE (TN).
- If the instance is NEGATIVE and it is classified as POSITIVE it is counted as FALSE POSITIVE (FP).

In a binary classification model, there exist two possible errors. One is False Positive (FP) and the other one is False Negative (FN). True Positive (TP) and True Negative (TN) considered being two classification results that are accurately classified by the classification model.

Accuracy, Precision (P) and Recall / Sensitivity (R) are most frequently used performance measures in classification models where precision determines how many positive instances

classified are actually positive while recall determines how many positive instances are actually classified by the model. Thus, the higher the precision and higher the recall resembles the classifying model has a better performance. But precision and recall have a trade-off in between since it's difficult to expect both in higher rates at once. Thus, difficult to analyze a predictor model alone with precision and recall. According to the confusion matrix, the above-mentioned performance measures can be described as follows.

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

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

F-Measure or F1 measure is another performance measure which comes along with the above mentioned precision and recall. F1 measure described as the harmonic mean of precision and recall. An increase in F1 measure indicates an increase of both precision and recall.

$$F1 = 2 \times \frac{(P \times R)}{(P + R)}$$

Additionally, there are several custom defined or modified measures that have been used to evaluate binary classification models in the related domain of defect and vulnerability prediction.

In the attempt of personalized defect prediction models of Jiang, T et al [11] they get the use of cost-effectiveness as a performance measure on prediction models. Cost-effectiveness of prediction models describes how maximum prediction can be achieved at a minimum amount of code line inspections.

Shin, Y et al. [7] suggests two additional custom defined performance criteria, file inspection ratio (FI) and line inspection ratio (LI) on vulnerability prediction models of software sources which gives feedback on cost-effectiveness. Further, they introduce the probability of false alarm (PF) measure as the ratio of files incorrectly predicted as vulnerable to actual neutral files. However, this study doesn't intend to evaluate the performance based on such specific

criteria since it may not be applicable for developer-centric vulnerability prediction performances

The approach of Scandariato, R et al [9] to predict vulnerable software components using text mining methods suggests an additional measure called fall-out / false positive rate (ϑ) which gives the probability of false positive results that can be produced by a prediction model. This is similar to the PF measure which has been suggested by Shin, Y et al. [7]. The fall-out measure enables to do comparisons with other studies in the domain and a lesser value is preferred.

$$\vartheta = \frac{FP}{FP + TN}$$

In addition to the performance measures which has been suggested by the authors in reviewed literature on vulnerability classification, many authors on classifier modeling have employed few other important performance measures such as Receiver Operating Characteristic (ROC) and Kappa Statistic

2.10.2 Receiver Operating Characteristic

The ROC curve is a graphical representation that illustrates the classification performance in binary classifiers. This curve is created by plotting True Positive Rate (sensitivity/recall) against False Positive Rate (fall-out). Thus, the ROC curve (Figure 2.3) is a sensitivity function of fall-out.

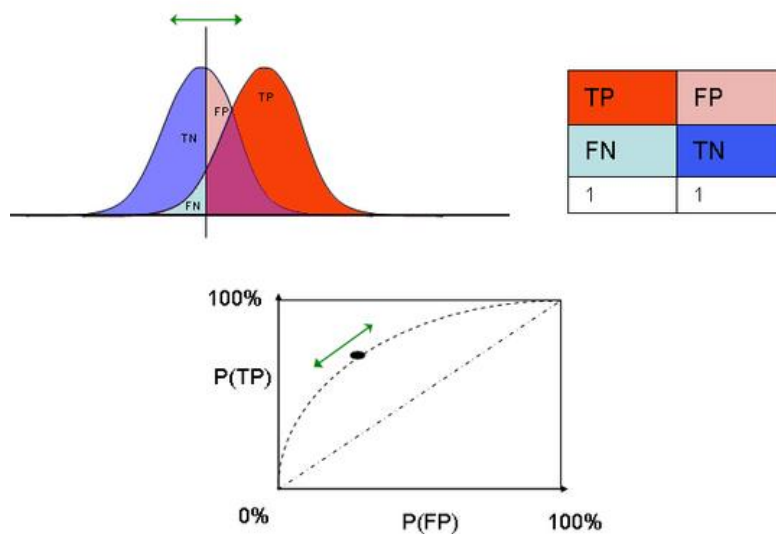


Figure 2.3: ROC curve behavior against confusion matrix

A classifier with perfect classification (no overlap in the two positive and negative distributions) has an ROC plot that passes through the upper left corner (100% sensitivity, 100% specificity). So, the overall accuracy of the classifier will be highest when the ROC plot is closer to the upper left corner. In ROC curve Area Under Curve (AUC) gives a numeric measure about the classifier performance.

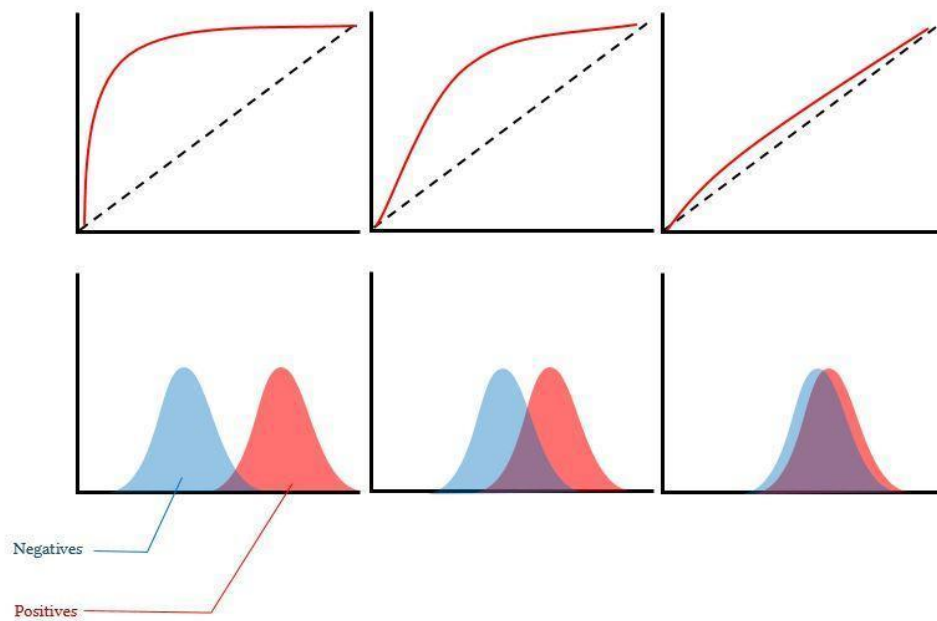


Figure 2.4: Different ROC curves and respective classification distributions

Figure 2.4 depicts the results of three different classifications and the respective ROC curves. The classification accuracies are decreasing subsequently towards the right side. As it can be observed the AUC values also decreasing with the decreasing classification performances.

2.10.3 Kappa Statistic

The Kappa statistic (or value) is a metric that compares an observed accuracy with an expected accuracy (random chance). Observed accuracy is simply the number of instances that were classified correctly throughout the entire confusion matrix. Expected accuracy is the accuracy that any random classifier would be expected to achieve based on the confusion matrix.

$$Kappa\ Statistic = \frac{(Observed\ Accuracy - Expected\ Accuracy)}{(1 - Expected\ Accuracy)}$$

2.10.4 Summary

According to the literature, the main prediction model performance measures on binary classification are accuracy, *precision*, *recall*, *F1*, *ROC* and the *Kappa Statistic*. Additionally, there are other measures that have been introduced by many authors specifically in vulnerability prediction such as cost-effectiveness and probability of false alarm to have a fine-grained comparison between different prediction models.

2.11 Error Analysis

Generally, in supervised learning, it has been identified that there are two main controllable components of errors that can effect on the trained models. Those are biased error and variance error. According to Fortmann-Roe, S. [31] total error ($Err(x)$) of a learning model can be decomposed as follows.

$$Err(x) = Bias^2 + Variance + Irreducible\ Error$$

The third component, Irreducible Error has been identified as the error that cannot fundamentally reducible by any of the models. Thus, the minimum error rate can be desirable if bias and variance components are zero.

2.11.1 Bias Error

Bias error increases with the oversimplification of the trained model when an important feature relationship of the training dataset has been ignored by the learning algorithm. Models with high bias error rates represent under fitting problem.

2.11.2 Variance Error

High variance error is caused due to the learning model oversensitivity on the training dataset. Higher variance error rates in trained models introduce random noise to the test data output which causes a higher error rate on expected vs resulted values.

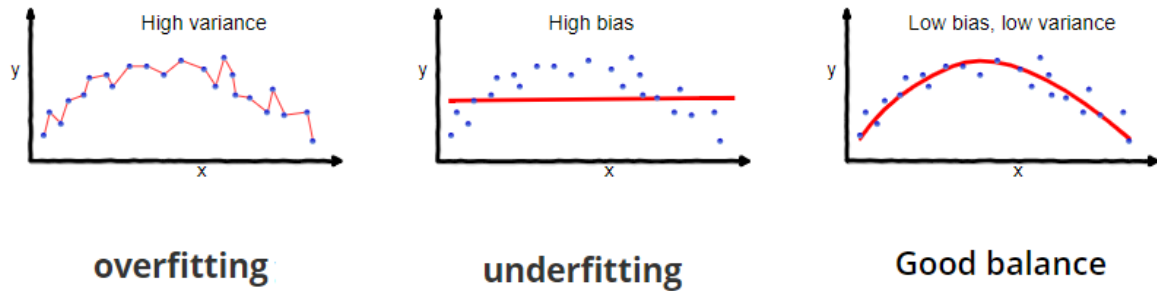


Figure 2.5: Bias-Variance Tradeoff

2.11.3 Summary

As identified, bias and variance are the two main components of the total error in a supervised learning model. Figure 2.5 shows the two problems of underfitting and overfitting scenarios that can be caused by high bias or variance. Thus, in order to produce a model with a lesser error rate, the models should be designed and trained in such a way that can balance bias-variance tradeoff.

To deal with the bias-variance tradeoff management problem, several techniques have been suggested [31] such as the use of ensemble techniques, K-fold cross-validation and feature selection. Thus, this study will focus on these identified techniques when managing the prediction model performance errors.

Chapter 03: Problem Analysis and Methodology

Generally, we can see this prediction problem as a binary classification problem that classifies whether a particular developer is vulnerable or not. However, there can be an argument on this since two vulnerable developers cannot be differentiated for their vulnerability by this approach. As per the scope, this study only focuses on predicting the vulnerable developers but not to produce a comparison between two developers who have been predicted as vulnerable. However, This research work can be taken as the underpinning work towards producing prediction models that can further differentiate each developer-centric vulnerability. Since the classifying classes are known and the dataset is available with the target classes; this can be solved through the supervised classification method discussed in the previous chapter.

Within the scope of this study, there are two main research questions to be answered in order to develop an efficient model for predicting vulnerable developer profiles based on each developer's superficial characteristics of developer behavior. First, it needs to be identified what is the most suitable set of behavioral features of developer profiles that predicts the developer-centric vulnerability. Second, it needs to be identified what are the most suitable classifiers that predict vulnerable developers effectively and efficiently and further techniques can be used to improve the performances of classifiers. In Chapter 03 we analyze the methods and approaches that can be used to answer the above research questions in detail and define the appropriate research methodologies to derive conclusions.

3.1 Candidate Feature Vector

Chapter 2 reveals that there are many metrics that have been used as the features of the defect and vulnerability prediction. As the initial step, a set of candidate developer behavioral metrics is selected based on the requirements, constraints and limitations discussed in previous chapters. Further, in selecting developer features for this study; some of the software features also have been considered to be selected as personalized software features.

As per the reviewed literature, there have been several instances that some of the authors have transformed developer feature metrics into software feature metrics. Similarly, this study focused on identifying several software features that can be measured at a personalized level. For example, the number of commits feature metric can be considered in both software component level as well as the personal level of each developer. Thus, the feature vector selected for this study is a combination of personalized software features and developer techno-behavioral features. Since these personalized software feature metrics also characterize the similar notions of developer features; ultimately the selected feature metric vector is considered as a vector of developer techno-behavioral feature metrics.

According to previous studies, different authors have been using different or similar types of feature vectors. In the selection of candidate feature vector for this evaluation, it basically hypothesizes that the developer features used by the previous works (in the literature review) can be significant for developer-centric vulnerability predictions as well with the support of claims done by Shin Y, et al [7]. Finally, the feature selection techniques discussed in Chapter 2 can be applied to derive a fine-grained feature vector which has a higher correlation with developer-centric vulnerabilities.

Data points for each developer feature metric captured through identified and custom developed mining tools on identified repositories. Based on hypotheses that may have good discriminative power on developer-centric vulnerability prediction.

In the below sections outline the detailed descriptions and hypothetical justifications on all candidate feature metrics selected for this study.

3.1.1 Comment Percentage

It's a common practice for developers to add textual comments in their source code contributions and it's a known fact that documentation on source code is best practice to avoid many issues in development including vulnerabilities and defects. Shin Y, et al [7], [8] has been used comment density of the files as a software metric. This metric can be transformed into a personalized software feature metric (a kind of developer metric) if comments added by each developer considered as a percentage of total number lines of code contributed by the developer.

Hypothetically it can be assumed that developers who do better documentation in source codes have a better understanding and background knowledge about the domain and the code. Therefore lesser vulnerable code contributions are desirable for the developers with higher comment percentages. Thus, developer comment percentage is considered as a candidate feature metric for developer vulnerability prediction.

3.1.2 Lines of Code Count

Lines of code per each developer represent the overall contribution amount of a developer in a revision of a project. Lines of code is a characteristic variable for each developer defines their implementation pattern.

Lines of Code Count has been utilized as a feature metric on software defect/vulnerability prediction studies by different authors [7], [8], [33] whose main focus to consider the lines of code counts in individual source code files or software components. In this study, this has been redefined as a count of code lines contributed by the individual developer in order to characterize it as a developer related behavior. According to the research work done by Zhang, H. [36] lines of code in software components have a positive correlation with the defects. Thus, it can be assumed that the developers who contribute more lines of codes can be susceptible to introduce security defects rather than those who do less number of code lines.

3.1.3 Change Percentage

Change percentage of a developer measures the percentage of contribution (line insertions and deletions) that have been done by a developer in the project so far. The overall percentage of the contributions that have been done by each developer to a software project can describe the role and involvement of each developer on that particular software repository.

Matsumoto, S et al [33] also suggested a similar set of metrics in his study to predict the software defects and has obtained successful outcomes. Thus, developer change percentage can be a possible candidate developer behavior feature metric correlates with individual developer-centric vulnerability. The hypothesis that has made to select this as a candidate feature is that the developer who claims for a higher percentage of changes has a higher probability to account for the vulnerabilities in the project.

3.1.4 Lines Deleted

This feature metric describes the number of lines deleted by the developer. During the contributions done by developers, some of the existing code fragments can be deleted. In the works done by Shin Y, et al [7], [8] number of deleted lines considered as a software metric. But it can be simply transformed into a developer feature by just calculating the line deletions done by individual developers by analyzing through each commit logs. Thus, lines deleted by an individual developer is considered as a behavioral characteristic which can be used in this study.

Source code lines of software may get deleted by users due to many reasons. Among the developer's under engagement, unclear requirements, problem complexities are the most common reasons. This study hypothesized that the developers who delete code lines since previous lines have some functional or non-functional weakness and the developer has a good understanding of the expected functionality. So, the assumption to select this feature is that developers who do a higher number of code lines may have a lesser developer-centric vulnerability.

3.1.5 Lines Inserted

The number of lines inserted by the developer. Same as lines deleted measure, lines of code inserted by each individual counted as a behavioral characteristic of the developer. As mentioned above; Shin Y, et al [7], [8] have used this software feature metric to predict software component vulnerabilities.

The hypothetical assumption that has made to select Lines Inserted as a candidate developer feature is similar to the lines of code count feature. Although the developer might contributing to a new feature; a higher number of code lines insertions can be a cause of a higher number of security vulnerabilities.

3.1.6 Change Stability

Change stability describes how stable the source code contributions that has been done by the developer so far in the project. This is the percentage of source code lines remaining so far from

inserted lines of code. The feature is similar to the opposite of code churn software metric used by Meneely, A. et al [34] in order to predict defects on source files.

Code churn is a metric that counts the evolving rate of code in a software component. Thus, it can be assumed that the change stability feature is the transformed developer feature of code churn software metric which will support to predict developer-centric vulnerability. The initial hypothetical basis for selecting this feature is the assumption that the developers who have higher change stability may have done less security vulnerable code contributions.

3.1.7 Average Cyclomatic Complexity

Cyclomatic complexity is a software metric that is used to measure the complexity of the source codes. The expectation is to measure the developer-specific complexity of the code contributions of each developer. Since the complexity measures will vary from one code fragment commit to another done by an individual developer, an average value of all contributions will be calculated as the developer's cyclomatic complexity for convenience.

Cyclomatic complexity has been used as a software metric by several authors in their studies. Zimmermann, T et al [10] used the total and average cyclomatic complexities of software components and a similar variant of this feature has been used by Shin Y, et al [7], [8] to predict the vulnerabilities in software components.

The hypothetical assumption that has made in selecting this as a candidate feature is that the developers who contribute with higher average cyclomatic complexity may contain security vulnerabilities in their contributions due to overlooked scenarios that developers might miss to cover up.

3.1.8 Developer Contribution Age

Developers are starting to contribute to software projects at different times. Thus, their knowledge of the project differs based on the time that they started to contribute to the software project. Developer contribution age is the metric that counts a number of days so far since particular developer's first commit. It can be assumed that the developers with higher contribution age are experienced enough to understand the security standards in the project do fewer vulnerability defects in their code contributions.

In the research work of Jiang, T et al [11], have used a software metric for file age in order to do predict software defects. Thus, it can be identified that the developer contribution age feature metric is the transformed version of the file age metric which can be used to predict developer-centric vulnerability.

3.1.9 Developer Closeness

Closeness is a centrality metric of a developer network that is created by identifying developer interconnections. The closeness of a developer is measured by calculating the average distance to any other developer in the network graph.

This developer feature has been used as a transformed software component feature in the works done by Shin Y, et al [8] which can indicate whether the particular software component has been contributed by central or non-central developers of the developer network.

This metric measures how closely these developers are interconnected when collaborating to the software project and it supports to understand and capture the collaborative behavior of developers which can be important on vulnerability prediction. Hypothetically it can be concluded that the developers with higher collaboration behavior with other developers; will share the security standards and norms towards avoiding security defects. So, developers with higher collaboration behavior will have reduced developer-centric vulnerability measures.

3.1.10 Developer Betweenness

Betweenness is another centrality metric that can be derived from developer network graphs. In a network graph, the path is a non-repeating sequence of adjacent nodes. The shortest path between two nodes is identified as the Geodesic path.

As same as the developer closeness, betweenness also a developer feature metric used by Shin Y, et al [8] as a transformed feature for software component files to indicate particular software component has been contributed by the central or non-central developer of the developer network graph.

The developer betweenness is the number of geodesic paths that include a particular developer over a total number of geodesic paths in the network graph. As same as closeness, betweenness

also measures the developer interconnectivity and possibly correlated feature for developer-centric vulnerability predictions. As similar to developer closeness; this feature also describes another aspect of developer-developer collaboration behavior. Thus the hypothesis made on developer closeness also would be valid for this feature as well.

3.1.11 Number of Commits

A number of commits by the developer also a primary behavioral feature. Many authors [7], [8], [11], [33], [34] who have done related works have been suggested to utilize a number of commits as a feature metric. Thus, this study also intends to use this feature in the final feature vector on predicting developer-centric vulnerability. It can be assumed that a higher number of commits may contain a higher number of code lines per each developer. As same as the hypothetical assumption discussed in lines of code feature; a number of commits that have been contributed by a developer also considered as a developer-centric feature for this analysis.

3.1.12 Frequent Commit Hour

In the study of Jiang, T et al [11] they have been using the commit hour as a feature metric on software defect prediction. As we can identify this feature can be transformed to developer's feature by selecting the frequent commit hour of the day on each developer. Developers do commits to the source control system in different times in a day. Due to natural reasons human developers working skills, the mood may get affected (impaired or enhanced) based on the time that they work. So the expected coding quality may get affected and could be a cause to contribute to vulnerable codes. Thus, there is a hypothetical assumption to select frequent commit hour as a developer feature metric that can describe the most frequent hours that the developer has worked.

3.1.13 Average Commit Interval

Developers are committing their code changes to the version control system from time to time. Some developers are frequent committers and some are not. In the study of Weicheng, Y et al [21] to analyze the developer commit patterns they have used average time distance in between commits per developer. They have omitted the distances more than 10 days since they are making some unnecessary noise in the data set. The average distance between commit intervals of each developer also can be a candidate feature measure in this study as well since it captures

some behavioral patterns of the developer. The hypothesis that is considered on this feature is the developers who have lesser commit intervals may not take time to think or re-evaluate on the security standards or security defects on their contribution and commit frequently. Thus, the developers with lesser average commit intervals may be having a higher developer-centric vulnerability.

Following the outcomes from Weicheng, Y et al [21] this study adopts an average commit interval feature as a vital developer behavioral metric that can cause an effect on vulnerable code contributions.

3.2 Repository Selection for Dataset

As defined in the scope of this study, we are focusing on software repositories that are openly available in GitHub social coding platform to capture the required dataset. Since there are many open source repositories available, it's required to select the most suitable repositories considering the requirements, scope and the limitations of this study.

In the selection of software repositories for this study, the following characteristics have been considered.

3.2.1 Implementation Language

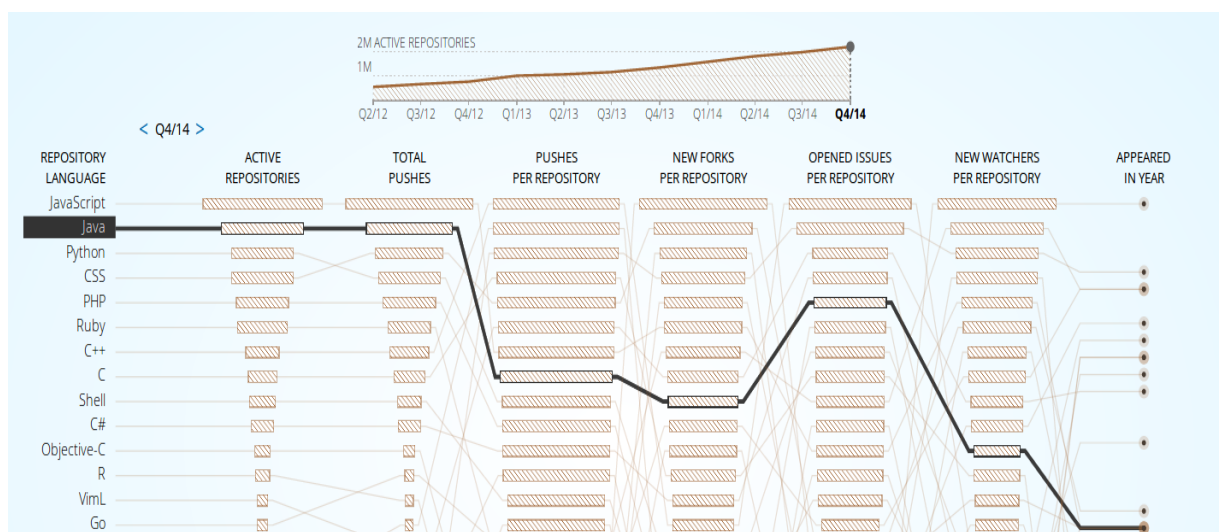


Figure 3.1: Top Active Languages in GitHub (source: <http://github.info/>)

GitHub has software repositories implemented in many languages. But we have considered the repositories that are implemented in Java language since it is popular, widely used and Figure 3.1 shows that Java has been ranked as the second most active repository holding language in GitHub. The availability of many supported repositories analyzing tools for Java language is another plus point for this decision.

3.2.2 Build Tool

Java projects in Github uses several types of build management tools such as Ant, Maven, Gradle, etc. But SonarQube tool that is used to filter out vulnerable developer profile is only supported for the projects developed using Maven and Gradle build tools. Thus, the projects that are built with Maven or Gradle build management tool will need to be filtered out from Github.

3.2.3 Number of Contributors

In order to train and develop a good prediction model, there should be richer and complete data set for training and verifications. When a repository has a higher number of contributors increases the chances of having a good dataset to develop a high performing prediction model. Thus, having a higher number of contributors is strongly considered a point for repository selection.

3.2.4 Number of Commits

A higher number of commits represents more interactions of the developers with the repository. That enriches the extracting dataset which also helps to train the high performing prediction model.

3.2.5 Number of Releases

A higher number of releases represents the stability and continuity of the software. Thus, having a higher number of releases imposes the importance and the significance of the repository to be chosen as a candidate repository for data extraction.

3.2.6 Selected Repositories

According to the above criteria following repositories has been examined to capture the features that are identified.

Repository	Description	Build Tool	Contributors	Commits	Releases
Apache Curator	Apache Curator is a Java/JVM client library for Apache ZooKeeper	Maven	74	2,571	108
Apache Ignite	Memory-centric distributed database, caching and processing platform	Maven	198	25,375	83
Guava	Google Guava is an open-source set of common libraries for Java	Maven	170	4,860	170
Netty	Netty is an asynchronous event-driven network application framework	Maven	357	9,098	199
Nifi	Easy to use, powerful, and reliable system to process and distribute data.	Maven	213	4,898	67
OkHTTP	An HTTP & HTTP/2 client for Android and Java applications	Maven	171	3,454	55
RxJava	Reactive Extensions for the JVM	Gradle	233	5,505	209
Spring Boot	Spring Boot is an open source Java-based microservices framework	Maven	531	19,973	115
WSO2 - IS	WSO2 Identity Server is an open source Identity and Access Management solution	Maven	96	3,715	117
WSO2 - Test Grid	TestGrid provides the enterprise customers confidence in the products and updates WSO2 ship	Maven	14	1,619	44

Table 3.1: General information about selected Github repositories

3.3 Data Extraction

Data extraction of the features on selected data repositories is an important and critical point of this study. The accurate set of data is expected to get a valid prediction model. To extract the data from repositories, we use a set of tools that are already available and custom developed scripts for certain features when there are no tools available to extract those features from the repository. Below describes the tools and techniques that are used in this study to extract the features discussed in 3.1

3.3.1 Gitinspector

Gitinspector is a statistical analysis tool for Git repositories that originally developed to fetch repository statistics from student projects at Chalmers University of Technology and Gothenburg University [15]. Gitinspector tool provides a wide range of detailed and useful statistical data about the Git repository and its developer characteristics that are inspected and provides sophisticated options to manipulate the analysis. It supports for several output formats such as JSON, XML, HTML and plain text to export the generated analytic data of a repository.

The JSON output format (See Appendix A) of statistical data will be used in this study and will be further processed in the analysis phase.

3.3.2 Jmine

Jmine is a custom-built tool by the author of this study using jGit java library to mine repositories for identifying developer relationships in a software project based on file modifications that have been done on .java file extensions. If two developers have done modifications in the same java file, it's considered that there's a connection in between them. These relationships aim to build a developer network of each repository to identify the inter-developer interaction behaviors and that can derive networking measures such as centrality, closeness and clusters.

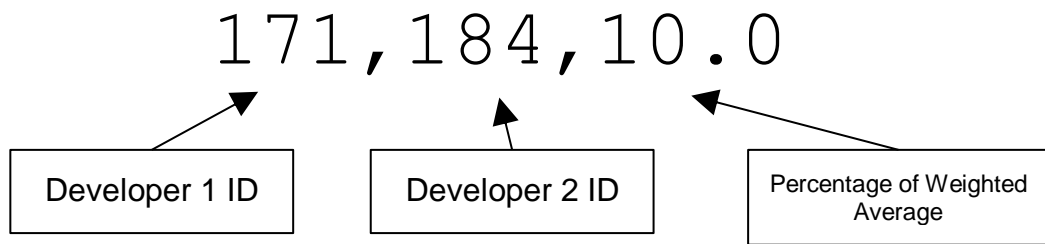


Figure 3.2: Output Data Set Format of *Jmine* Analyzer

This tool produces a dataset in a comma separated value (CSV) format containing three values. The first and second value of the series represent unique developer ids that are assigned by the tool representing a developer-developer non-directed connection and last value represents the percentage of the weighted average value of that connection. Generally in the dataset; the developer email id has been considered as the unique identifier of the developer data. But in order to produce the developer network graph, another unique key had to be introduced for convenience. Thus, in Figure 3.3 shows that the developer network connections that indicate the collaboration links between different developers in a software project. In this graph, each edge represents a software component (source file) collaboration and the edges represent the developers.

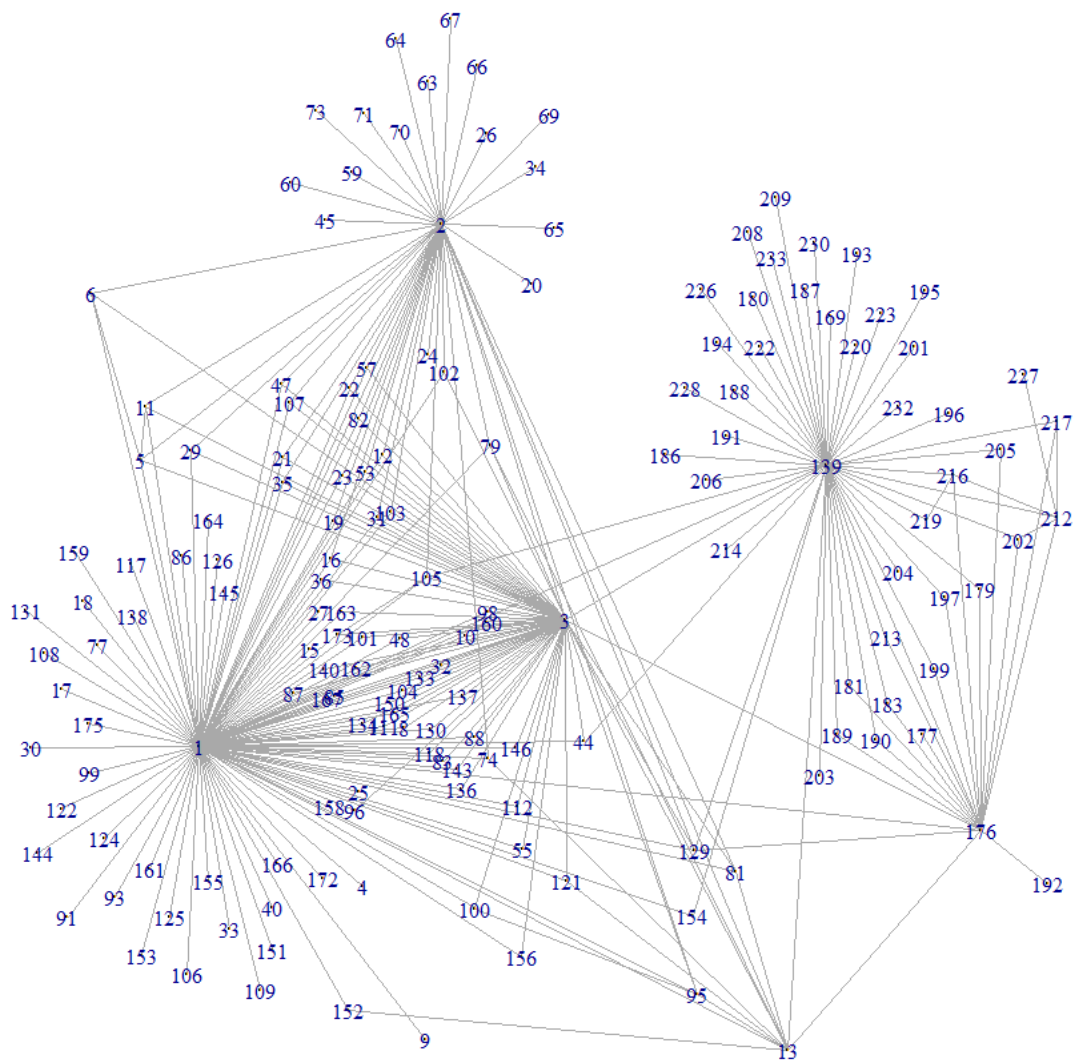


Figure 3.3: Developer Network Graph of Metrics GitHub Project

3.3.3 R and R Studio

R is a well-known programming language for statistical computing which supports in applications of statistics, data mining and data analytics. R Studio is an Integrated Development Environment (IDE) for R language which supports R developers by providing a sophisticated development environment.

In this study, R and R Studio is used to produce developer network graphs using igraph [20] package and get the statistical measures of developer networks. The CSV output of the Jmine tool is the input source for R and igraph package to produce the developer network. The known

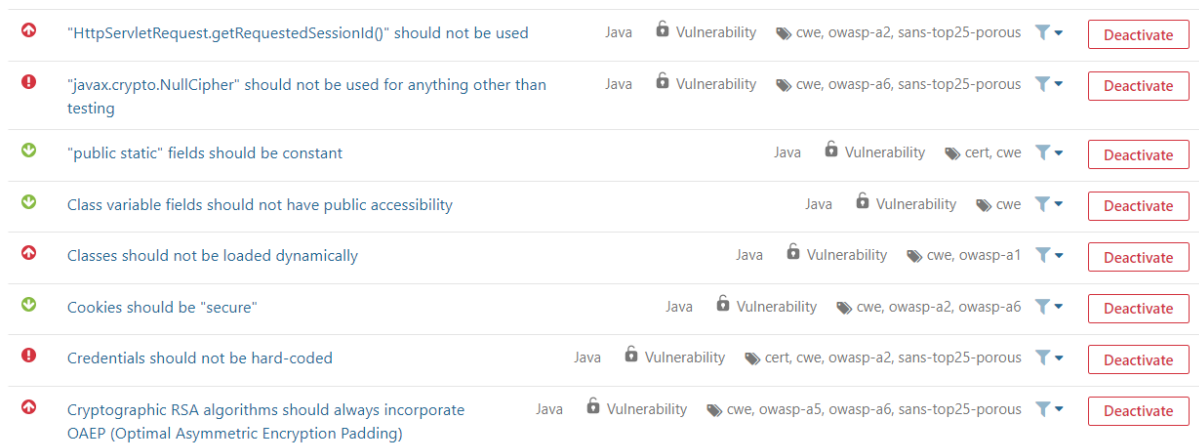
popularity, support and availability of sophisticated packages are the primary reasons for adopting R and R Studio as the main experimenting environment.

3.3.4 SonarQube Tool

SonarQube is a tool used to manage continuous code quality measures and allows to do analysis on software projects with more insightful details regarding bugs and vulnerability issues. SonarQube supports for OWASP, CWE and many other vulnerability rules and it is supported for Gradle and Maven build tools.

SonarQube Quality Profiles feature facilitates a sophisticated way to select set of defect, vulnerability, technical debt and code smell rules from a predefined standard (by referring OWASP, CWE, etc.) and verified rule set collection. These rules set collection can be filtered by the programming language (Java, Javascript, Python, etc.) and type (Bug, Vulnerability or Code Smell).

In the process of scanning for software vulnerable codes, a custom configured scanner quality profile created which only contain only security rules of Java programming language with the type labeled as Vulnerability. According to this custom-created quality profile; there are 33 rules that are used to do the software code vulnerability scanning.



The image shows a screenshot of a SonarQube interface displaying a list of security vulnerability rules. Each rule entry includes a status icon (red for error, green for success), the rule name, the programming language (Java), the rule type (Vulnerability), associated standards (like CWE, OWASP, or Sans), and a 'Deactivate' button.

Status	Rule Name	Language	Type	Standards	Action
⊕	"HttpServletRequest.getRequestId()" should not be used	Java	Vulnerability	cwe, owasp-a2, sans-top25-porous	Deactivate
⊕	"javax.crypto.NullCipher" should not be used for anything other than testing	Java	Vulnerability	cwe, owasp-a6, sans-top25-porous	Deactivate
⊕	"public static" fields should be constant	Java	Vulnerability	cert, cwe	Deactivate
⊕	Class variable fields should not have public accessibility	Java	Vulnerability	cwe	Deactivate
⊕	Classes should not be loaded dynamically	Java	Vulnerability	cwe, owasp-a1	Deactivate
⊕	Cookies should be "secure"	Java	Vulnerability	cwe, owasp-a2, owasp-a6	Deactivate
⊕	Credentials should not be hard-coded	Java	Vulnerability	cert, cwe, owasp-a2, sans-top25-porous	Deactivate
⊕	Cryptographic RSA algorithms should always incorporate OAEP (Optimal Asymmetric Encryption Padding)	Java	Vulnerability	cwe, owasp-a5, owasp-a6, sans-top25-porous	Deactivate

Figure 3.4: Sample set of security vulnerability rules

Thus, all other issues such as Bugs, Technical Debts and Code Smells will be ignored through the scan (Figure 3.5). The data set that captured through this tool will be used as the dependent data set to train the prediction models.

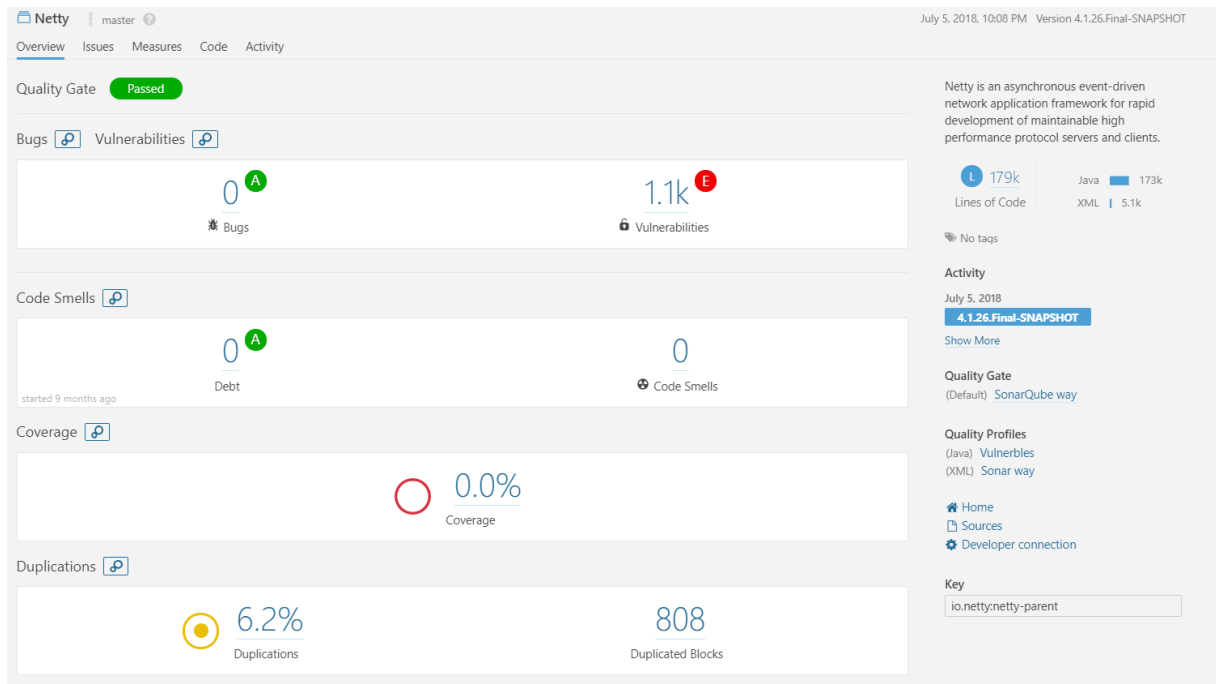


Figure 3.5: Overall SonarQube Assessment Sample of Netty GitHub Project

Figure 3.5 describes the top level statistics about the project which has been scanned. Since bugs, technical debts, code smells have been ignored in the custom defined scanning profile it shows only a number of vulnerabilities (1.1K = 1100 vulnerabilities according to the figure) which has been detected by the scanning process.

3.3.5 Technique

In the process of data extraction; each repository identified in Table 3.1 mined using the tools to extract all features in the feature vector. Meanwhile, in order to identify each feature extracted developer's class (has contributed vulnerable code or not), the software repository scanned separately through the SonarQube tool. At the end of the successful scanning process; the SonarQube tool persists all identified vulnerable code fragments (Figure 3.6) into a relational database with the particular developer information who has committed the security weakness.

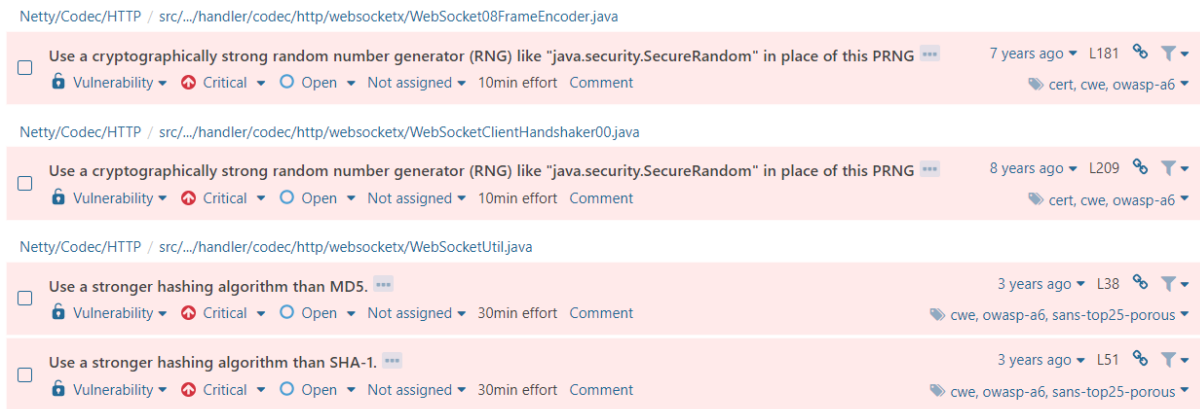


Figure 3.6: Identified Vulnerabilities of Netty GitHub Project

These persisted data manually extracted through SQL query and exported to JSON representation (Figure 3.7, the image has been distorted due to privacy concerns of the developers) for further processing and association purposes through the custom developed Jmine tool. Both extracted feature data and vulnerability class and other details exported from the SonarQube tool are fed into Jmine tool and it maps and consolidates each developer's feature vector with identified class. Finally, it produces the finalized dataset for the particular software repository.

```

1  [
2  { "id": "b1e7e2e2i@gmail.com", "issues":1 },
3  { "id": "notoai1@google.com", "issues":1 },
4  { "id": "suneaya@gmail.com", "issues":4 },
5  { "id": "lichun1_i1@apple.com", "issues":1 },
6  { "id": "irgan@google.com", "issues":6 },
7  { "id": "lgablwgv_r@twitter.com", "issues":5 },
8  { "id": "ed...@gatling.io", "issues":4 },
9  { "id": "yhatowrnable@gmail.com", "issues":1 },
10 { "id": "...@debian-desktop.home.org", "issues":3 },
11 { "id": "varex@jelen.biz", "issues":2 },
12 { "id": "...@motd.kr", "issues":259 },

```

Figure 3.7: Exported Sample JSON Data Set of Vulnerable Developers

The abstract functionality of the Jmine tool is to iterate through all feature extracted developers and cross check with the vulnerable developer list (Figure 3.7) to identify whether a particular developer has an entry there. If the entry was found in the vulnerable list; the class of that developer identified as vulnerable while others who haven't found an entry are decided as non-vulnerable developers. To produce the complete dataset; all finalized repository datasets are combined together (Figure 3.8).

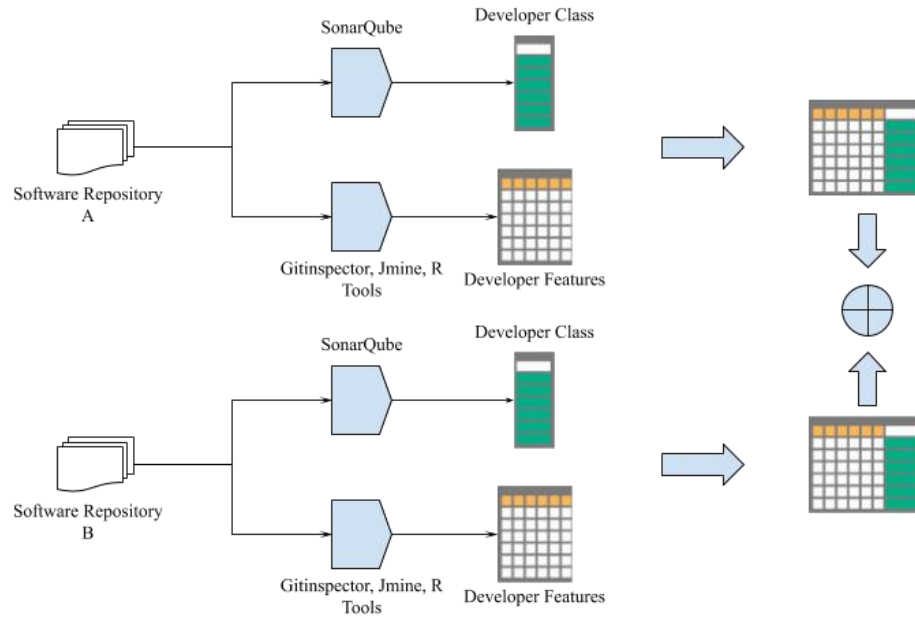


Figure 3.8: Feature Data Set Creation Process

3.4 Data Set

As described in previous Section 3.3 complete dataset collected through various existing and developed mining tools and techniques mentioned and analyzed and merged together into one main dataset at the end.

The final dataset that is collected by mining 10 GitHub repositories consists of 13 characteristic features of 1827 developers. According to the dataset (Figure 3.9) there are 383 identified developers who have contributed security vulnerable codes while others aren't contributed such codes.

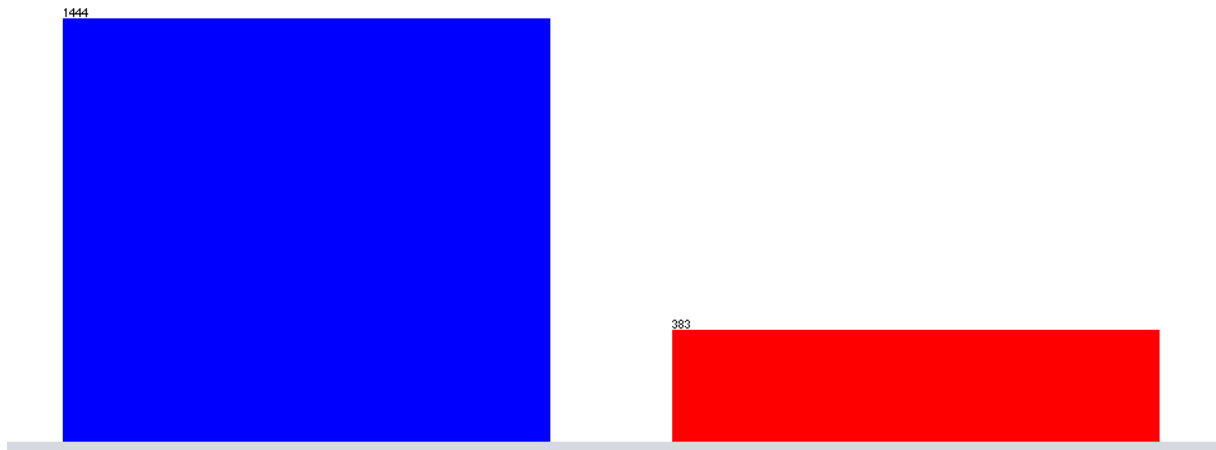


Figure 3.9: Class distribution histogram of collected dataset

Table 3.2 describes some basic statistical information about the dataset.

Feature	Missing Value (%)	Min	Max	Mean	Standard Deviation
lines_of_code_count	15	1	163460	1673.116	8178.325
average_commit_interval	42	0.2	185	44.822	33.085
frequent_commit_hour	2	0	23	11.699	7.715
lines_inserted	0	0	245008	2284.212	13263.096
lines_deleted	0	0	128530	1194.859	7944.188
developer_age	15	0.1	486.2	101.618	76.494
average_cyclomatic_complexity	41	51	1059	190.523	160.589
change_stability	15	0.2	4414.3	95.248	191.811
comment_percentage	15	51	100	20.856	28.47
developer_closeness	15	0.001	0.05	0.003	0.004
developer_betweennes	15	0	5736.206	81.492	407.337
number_of_commits	0	1	2123	16.066	100.016
change_percentage	0	0.001	66.02	0.293	2.436

Table 3.2 : General statistics of complete collected dataset

3.5 Data Set Preprocessing

As discussed in Section 3.3, several approaches have been utilized to fetch the developer related data for the selected feature vector. After that, all the datasets coming from various sources need to be combined into a single dataset and preprocessed before where it can be used as the training and testing dataset of the prediction model. Preprocessing of data is an essential step before using them as the core data set for model training since it ensures the quality of the data set which leads to the accuracy of trained models.

3.5.1 Missing Values

It's an obvious situation that a data set can have missing values. Here, there are such situations that the tools and techniques used to extract developer features aren't capable of capturing the data for all the developers or the data may not exist in the repository at all. Thus, there are a set of developers in the collected dataset who doesn't have a complete data set for all the feature vector. Although there are missing values in the dataset the tools and algorithms that are used on classification techniques are capable of handling the missing values in data points. So, there is less possibility for any negative impact on the experimental methodology.

3.5.2 Standardization

Standardization or z-score normalization of feature data is a general data preprocessing requirement in many machine learning problems. In the standardization process, feature data will be rescaled to be centered on 0 with a standard deviation (σ) of 1. Standardization is an important requirement when there are different types of feature data measured in different kinds of units.

3.5.3 Normalization

Normalization is an alternative method for standardization which is also known as min-max scaling. This approach scales the data into a fixed range which 1 - 0 range chosen typically. Normalization produces a data set with a lesser standard deviation while suppressing the effects of outliers.

3.5.4 Sampling

As identified in Chapter 2 sampling techniques will be applied to resolve the class imbalance problems of the collected dataset. Here, the dataset will be refined by the RUS technique which is the most preferred technique used in previous works as well as another preferred oversampling technique known as SMOTE. Both techniques are applied separately to the dataset and the performance will be compared in order to select the better technique.

3.5.5 Outliers

Outliers are the data that are abnormal observations in the data set which could have very higher or lesser values than the general observed population. These outliers cause deterioration of correct results on experiments. According to the statistical studies, the data observations that are beyond interquartile range can be identified as outliers of the dataset and normally those are excluded during the data preprocessing stage.

3.5.6 WEKA

WEKA (Waikato Environment for Knowledge Analysis) is a tool with a huge collection of data mining, machine learning algorithms and data preprocessing filters developed by the University of Waikato, New Zealand. WEKA tool flexibly supports for almost all the requirements of machine learning tasks for dataset preprocessing, classification, clustering, etc. It is used to preprocess the collected dataset and train different classifiers and evaluate the results in this study.

WEKA tool itself contains a collection of data preprocessing filters that can be readily applicable to the data set. Thus data preprocessing filters such as normalization, sampling are intended to be used on the data set to improve the prediction performances in the later stage classifier models. Additionally, some of the abnormal extreme values (obvious outliers) that have been observed in the data set had to be manually removed before the data preprocessing phase.

3.6 Feature Selection

In the feature selection process for this study, we have decided to go with wrapper methods. Since this study expects to compare four different types of learning models, the feature selection method should be unique and independent from the models in order to make a clear comparison between each. Further, wrapper methods are optimized and easy to configure. Although they have higher computational overheads, that's less significant for this study since it has a small feature vector.

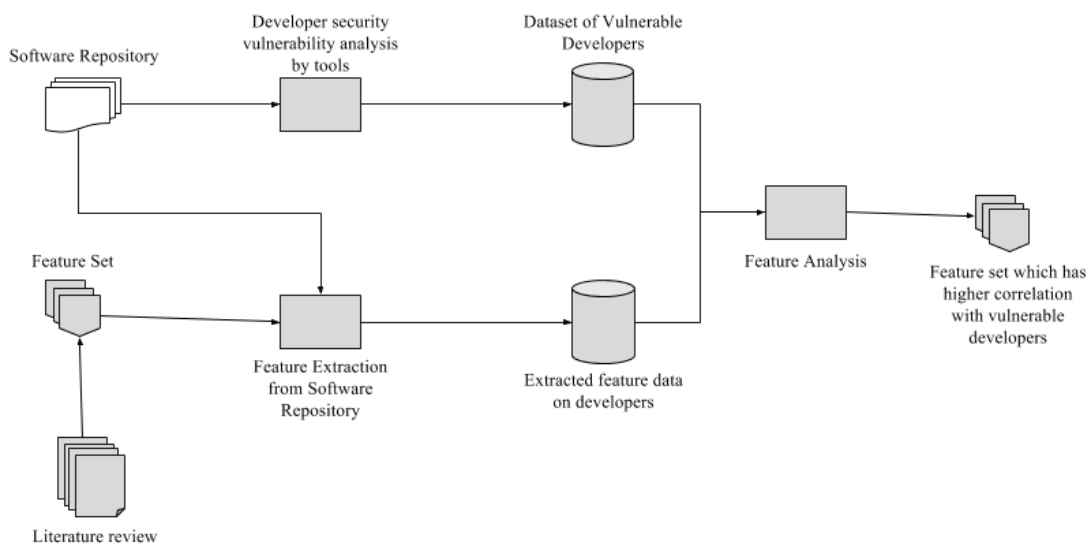


Figure 3.10: Feature Selection Approach Diagram

To select the most suitable feature subset from the identified feature set, the recursive feature elimination (RFE) technique will be employed. It is a wrapper method that has been identified in Chapter 2. Although the wrapper methods are computationally expensive than filter methods it's more convenient and advantageous to use a wrapper method since those handles the complexities of each and every feature of the data set. Additionally, according to the literature, the selection method doesn't affect much on the prediction performance. Thus, in order to select the most suitable feature vector for this study, an RFE technique will be used.

The Classifier Subset Evaluator in the WEKA tool provides a decent implementation of the Wrapper method [35]. For each & every feature analyzed, it provides a clear comparison of performance measures such as Root Mean Squared Error (RMSE), Mean Absolute Error

(MAE), etc. Then, based on a performance measure threshold, a subset of features can be selected for further training on selected prediction models.

3.7 Prediction Model Selection

The requirements and characteristics of this problem which identified in previous sections conclude that the most suitable prediction model for this problem is a regression model. In this study, we expect to focus on four different models from three different varieties and train them for prediction and evaluate them to identify the best performing model. Based on the suggestions in previous studies, related works, scope constraints and the findings of problem analysis Logistic Regression, Naive Bayes, Decision Tree and Random Forest algorithms were selected from a pool of classification algorithms available in WEKA tool as the better options to be evaluated as candidate model for this research problem. In Figure 3.11, M1 to M4 represents the candidate prediction models that are trained using four different candidate algorithms.

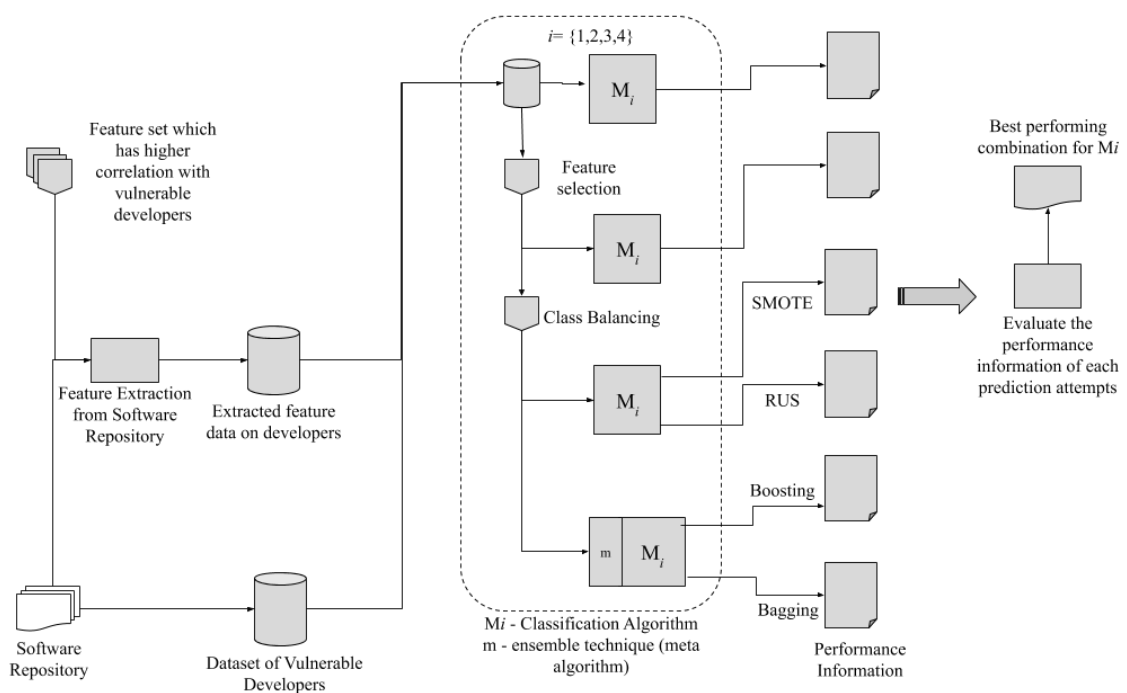


Figure 3.11: Prediction Model Selection Approach Diagram

3.7.1 Logistic Regression

The logistic regression is initially used in the field of statistics but later adopted in machine learning domain as well. It basically supports binary classification problems where the prediction target is categorical. But it can be applicable for multiclass classifications with some derivative works. It assumes the input variables are numeric and have a Gaussian (bell curve) distribution. But even the input variables do not show a Gaussian distribution still this algorithm achieves good results. Further, it assumes conditions such as less noisy, non-existence of multiple highly correlated features and less missing values in the dataset to produce better performances on prediction accuracy.

The algorithm learns a coefficient for each input value, which is linearly combined into a regression function and transformed using a logistic (S-shaped) function. Logistic regression is a fast and simple technique.

In WEKA tool logistic regression classifier can be used by selecting Logistic in classifier tab classifier collection list under the functions category.

3.7.2 Decision Tree

Decisions trees will create a tree to evaluate an instance of data start at the root of the tree and moving towards to the leaves (roots) until a prediction can be made. The process of creating a decision tree works by greedily selecting the best split point in order to make predictions and repeating the process until the tree is a fixed depth. After a decision tree is constructed the concept 'Pruning' can be used to reduce the size of the tree by removing the sections of the tree that provide very little power to classify the instances.

Decision tree models produce very intuitive rules which can be easy to explain and understand. Further, it efficiently handles non-linear feature correlations in the datasets which can be obviously contained in the dataset of this study. But decision trees are very vulnerable to overfitting problems with the training dataset. However, cross-validation techniques are there to overcome from overfitting problem in the Decision Tree algorithm.

In the WEKA tool, the decision tree classifier can be used by selecting REPTree in classifier tab classifier collection list under tree category.

3.7.3 NaïveBayes

Naïve Bayes is a simple and commonly used classifier algorithm in machine learning. It is a probabilistic classifier and it employs the Maximum A Posteriori decision rule to derive the classifications. In any of the probabilistic classifier attempts to determine the probability of each feature (x_1, x_2, \dots, x_n) occur in each class (c_1, c_2, \dots, c_n) and finally identifies the class which is most likely to occur. Therefore, the probability of each class needs to calculate for a given feature vector that occurred $[P(c_i | x_1, \dots, x_n)]$.

In order to calculate these probabilistic values for each class, the Bayes rule is applied.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$$P(c_i | x_1, \dots, x_n) = P(x_1, \dots, x_n | c_i) * P(c_i) / P(x_1, \dots, x_n)$$

A, B components are replaced with the class and the feature vector components respectively. Since it is difficult to calculate P(B) component which is $P(x_1, x_2, \dots, x_n)$, the Bayes rule redefined as a proportional relationship by ignoring that component.

As identified; Naïve Bayes is a simple implementation and it is computationally fast. Thus, it is capable of handling high dimensional datasets. But in the specification of the Naïve Bayes algorithm; it relies on several assumptions about the dataset which may become false on typical datasets.

In the WEKA tool, naive Bayes classifier can be used by selecting Naive Bayes in classifier tab classifier collection list under the Bayes category.

3.7.4 Random Forest

Random forest is an algorithm that can be used for both classification and regression problems. This algorithm creates a forest with several decision trees. The larger the number of trees we can produce the more accurate the results will be. The rationale behind this algorithm is using a combination of learning models will increase the classification accuracy. This concept is known as bagging. Bagging generates diverse classifiers only if the base learning algorithm is unstable. That is small changes in the training set to result in large changes in the learned classifier. Since neural networks and decision trees are unstable classifiers, they are good candidates to apply the technique.

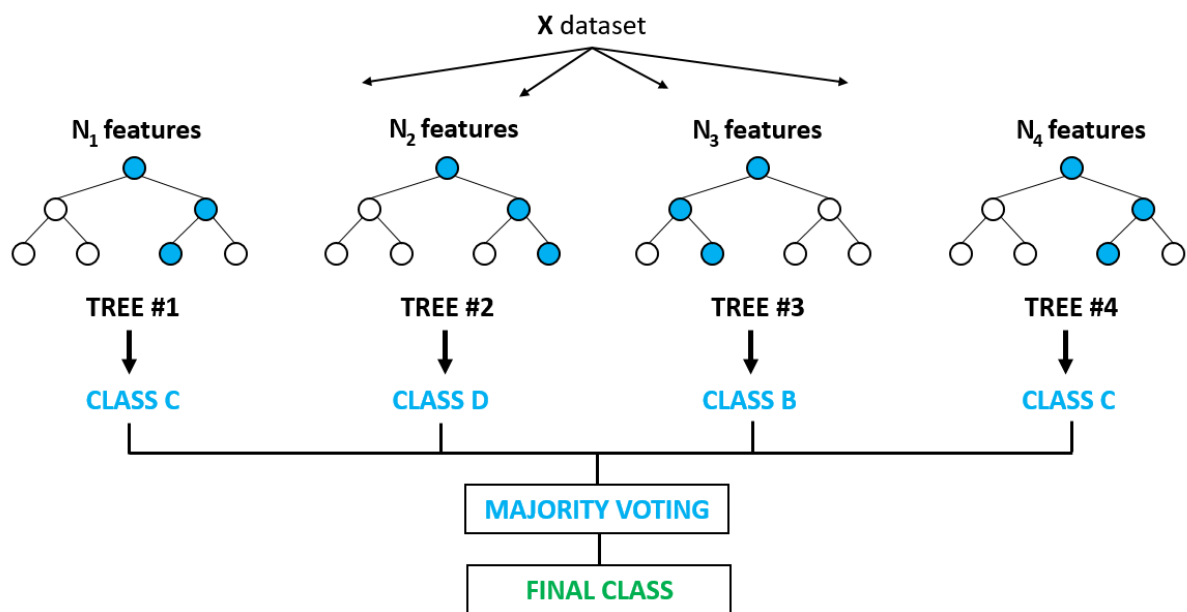


Figure 3.12: Random Forests Algorithm

Overfitting is one critical problem in most of the other algorithms that may make the results worse, but for the Random Forest algorithm, if there are enough trees in the forest, the classifier won't overfit the model. Further, the Random Forest algorithm is immune to the issues when dealing with a set of features that may have inter-correlations.

To classify a given instance it will use the rules of each randomly created decision tree to predict the outcome and then calculates the votes for each of the predicted targets. Then it will highest be voted predicted target as the final prediction.

In the WEKA tool random forest classifier can be used by selecting RandomForest in classifier tab classifier collection list under tree category.

3.7.5 Support Vector Machine (SVM)

Support Vector Machine is also an algorithm that can be applied for both supervised classification and regression problems. In the classification problems, SVM attempts to identify a hyperplane in an N-dimensional feature space that can classify the data points into the target classes. When there are multiple hyperplanes can be identified for a classification problem; SVM attempts to find out the hyperplane that can produce Maximum margin in between data points of classes (Figure 3.13).

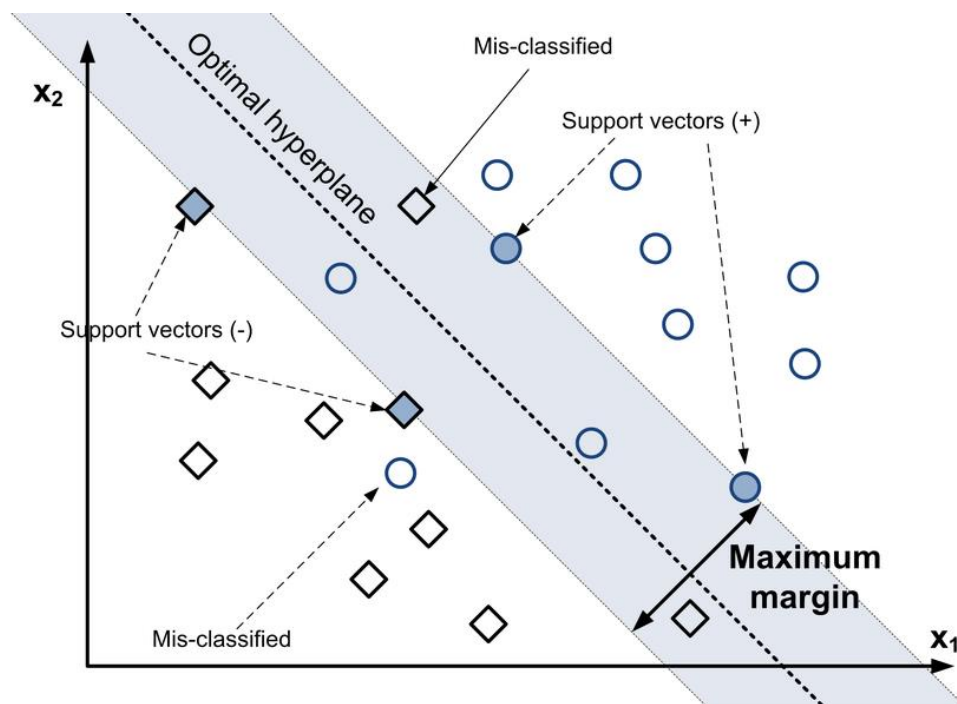


Figure 3.13: SVM for 2-dimensional Feature Space

SVM is capable of handling high dimensional datasets, but it performs well with smaller and clean datasets. Thus, the performance will get affected if the dataset is large and noisier.

In the WEKA tool, two implementations of the SVM algorithm are available to select under functions algorithm category collection. SMO or LibSVM can be selected in the experiments to get the use of SVM classification. However, LibSVM is a third party implementation of SVM which doesn't contain in the WEKA tool by default. So, SMO implementation is being used as the SVM implementations in the study experiments.

3.8 Training

Supervised predictive modeling is always involved in two phases. The first phase is the learning or the training phase where the prediction model is trained by a set of data that is already available with known independent and dependent parameters. The next phase is to evaluate the success of the learning phase through a verification and validation process. Training and evaluation of the models are about supervising the learning process of the models from a training dataset and validating the effectiveness of the learning process through a testing dataset.

3.8.1 Train-Test Split

The traditional approach to get these two datasets, the collected data set needs to be divided into two parts which can be named as training dataset and testing dataset. In supervised machine learning; the heuristic ratio for this separation is 7:3 where 70% from the whole dataset is taken into a training dataset and the remaining 30% as testing dataset.

But with this approach, the performances of the classifiers would be suboptimal. Since there's always a tradeoff between selecting the training : testing ratio from the full dataset. Further, there is some uncertainty of bias effects on classifier performance due to some noise or randomness in training or testing datasets.

3.8.2 K-Fold Cross Validation

K-Fold Cross Validation is another approach that is used to train and validate classifier models which can avoid the difficulties that can be experienced with the previous train-test split method. As shown in Figure 3.14 the dataset is divided into K number of folds/bins and K-1 number of folds are selected as training dataset while the other one is used as the testing dataset.

The process will go through a K number of iterations and the average performance of all iterations will be considered as the final classifier performance. The value of K is an arbitrary number decided based on the dataset size and time constraints of the training-validation process. Heuristically, the value of K has been selected as 10 in most experiments.



Figure 3.14: K - Fold Cross Validation

3.8.3 Summary

This study gets the use of K-fold cross-validation technique when training and validating the classifier performances and K value selected as 10 as suggested in most of the other examples. But fine tunes on this value and method may have to be considered based on any new knowledge and findings throughout the study.

Chapter 04: Developer Vulnerability Prediction and Evaluation

This chapter discusses the techniques and experimentation approaches that have been carried out to produce the prediction models. Further the mechanisms on evaluating and comparing their performances.

4.1 Benchmark

It's important to have benchmarking models to compare the performance measures of the classifier models that are produced as this research outcome. As per the evaluations are done in the literature review, there's no predefined classifier model to be taken as the benchmarking model.

Hence, Zero Rule (ZeroR or 0-R) classifier which is a basic and simple classifier model employed as the benchmark. ZeroR classifier algorithm always classifies the instances into the majority class of the trained dataset ignoring all the predictor features.

Here, the basic model will be trained using the complete feature vector and the performance measures will be taken as the benchmarking figures for the improved versions of classifier models.

Table 4.1 shows the benchmark performance measures of the ZeroR classifier.

Classifier	Accuracy	F-Measure	ROC (AUC)	Area	Kappa
ZeroR	79.0367%	0.883	0.496		0

Table 4.1: Benchmarking classifier performance measures

4.2 All Feature Vector Performance

At this stage, the initial performances of all selected classifiers are evaluated without applying any data preprocessing technique or feature selection technique. These results produce a general idea about how each classifier algorithm behaves with the raw dataset.

	ZeroR	Naive Bayes	Logistic Regression	Decision Tree	Random Forest	SVM
Accuracy (%)	79.0367	83.1081	83.7297	85.5676	85.2432	82.1018
F-Measure	-	0.779	0.808	0.848	0.840	0.770
AUC	0.496	0.672	0.792	0.838	0.890	0.583
Kappa	0	0.2846	0.3814	0.5322	0.4987	0.2344
Precision	-	0.828	0.84	0.854	0.848	0.829
Recall	0.79	0.835	0.843	0.861	0.858	0.821

Table 4.2: Classifier performance measures on raw dataset

According to the classifier performances (Table 4.2) the Random Forest classifier produces the best results on the full feature set by using K-fold cross-validation. For this experiment, the K value was selected as 10 which is the default value in the referred literature as well.

4.3 Performance through Feature Selection

As the next step dataset evaluated through feature subset selection process discussed in Chapter 3 and the most significant predictors will be identified. The classifiers are trained using filtered dataset and performance measures are evaluated.

The feature subset is selected through a wrapper method by employing the Bagging meta-algorithm as the learning scheme with a full dataset in the WEKA tool. Here in the wrapper method, attempts to find out the best possible feature subsets by traversing through the feature subset space by using a greedy best first search algorithm. Next, the performances of each feature subset will be evaluated by the K-fold cross-validation technique. The final output of

this process will produce the average significance rank of each feature attribute identified by the forward and backward elimination searches in the best first search algorithm.

Then the performances of each classifier can be evaluated while step by step elimination of least significant features of the feature vector.

=== Run information ===

```
Evaluator:   weka.attributeSelection.ClassifierSubsetEval -B weka.classifiers.meta.Bagging -T
-H "Click to set hold out or test instances" -E DEFAULT -- -P 100 -S 1 -num-slots 1 -I 10 -W
weka.classifiers.trees.REPTree -- -M 2 -V 0.001 -N 3 -S 1 -L -1 -I 0.0
Search:      weka.attributeSelection.BestFirst -D 1 -N 5
Relation:    FinalDataProcessed_v3-weka.filters.unsupervised.attribute.Remove-R1
Instances:   1827
Attributes:  14
             lines_of_code_count
             number_of_commits
             average_commit_interval
             frequent_commit_hour
             lines_inserted
             lines_deleted
             developer_age
             change_stability
             average_cyclomatic_complexity
             comment_percentage
             change_percentage
             developer_betweenness
             developer_closeness
             issues
Evaluation mode:   evaluate on all training data
```

=== Attribute Selection on all input data ===

```
Search Method:
  Best first.
  Start set: no attributes
  Search direction: forward
  Stale search after 5 node expansions
  Total number of subsets evaluated: 111
  Merit of best subset found:   0.926

Attribute Subset Evaluator (supervised, Class (nominal): 14 issues):
  Classifier Subset Evaluator
  Learning scheme: weka.classifiers.meta.Bagging
  Scheme options: -P 100 -S 1 -num-slots 1 -I 10 -W weka.classifiers.trees.REPTree -- -M
2 -V 0.001 -N 3 -S 1 -L -1 -I 0.0
  Hold out/test set: Training data
  Subset evaluation: classification error
```

```

Selected attributes: 1,3,4,5,6,7,9,10,13 : 9
    lines_of_code_count
    average_commit_interval
    frequent_commit_hour
    lines_inserted
    lines_deleted
    developer_age
    average_cyclomatic_complexity
    comment_percentage
    developer_closeness

```

Experiment 4.1: Classifier Subset Evaluation with Bagging

As per the experiment 4.1 results on feature subset selection process, 4 out of 13 feature matrices have been eliminated due to less significant correlation with target attribute (issues). developer_betweenness, change_percentage, number_of_commits, change_stability appears to be having the least predictive power while others are having better predictive power on developer vulnerability classification. Thus, the performance of each classifier re-evaluated to identify how the classifier performances are affected by the feature selection technique.

	ZeroR	Naive Bayes	Logistic Regression	Decision Tree	Random Forest	SVM
Accuracy (%)	79.03	83.1965	84.0175	85.3859	85.6596	82.0471
F-Measure	-	0.796	0.809	0.850	0.843	0.768
AUC	0.496	0.732	0.800	0.861	0.893	0.580
Kappa	0	0.3253	0.3687	0.5391	0.4945	0.2289
Precision	-	0.828	0.839	0.848	0.847	0.83
Recall	0.79	0.832	0.84	0.854	0.857	0.82

Table 4.4 : Classifier performance measures on feature selected dataset

```
=== Run information ===
```

```

Scheme:          weka.classifiers.trees.RandomForest -P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V
0.001 -S 1
Relation:          FinalDataProcessed_v3-weka.filters.unsupervised.attribute.Remove-R1-
weka.filters.unsupervised.attribute.Remove-R2,8,11-12
Instances:        1827
Attributes:        10
                  lines_of_code_count

```



```

    average_commit_interval
    frequent_commit_hour
    lines_inserted
    lines_deleted
    developer_age
    average_cyclomatic_complexity
    comment_percentage
    developer_closeness
    issues
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

RandomForest

Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 1.03 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances      1565           85.6596 %
Incorrectly Classified Instances    262           14.3404 %
Kappa statistic                    0.4945
Mean absolute error                 0.1828
Root mean squared error             0.3194
Relative absolute error             55.1226 %
Root relative squared error        78.4662 %
Total Number of Instances          1827

=== Detailed Accuracy By Class ===

      TP Rate  FP Rate  Precision  Recall  F-Measure  MCC      ROC Area  PRC Area  Class
      0.961    0.538    0.871     0.961    0.914     0.517    0.893    0.965    FALSE
      0.462    0.039    0.760     0.462    0.575     0.517    0.893    0.702    TRUE
Weighted 0.857    0.433    0.847     0.857    0.843     0.517    0.893    0.910
Avg.

=== Confusion Matrix ===

      a    b  <-- classified as
1388  56 |  a = FALSE
 206 177 |  b = TRUE

```

Experiment 4.2: Random Forest classifier evaluation on feature selected dataset

As per the experiment 4.2, the feature selected dataset is the best performing subset for the Random Forest classifier with 10-fold cross-validation. Further, the identified feature subset is evaluated with other classifiers to understand and compare the performance gain or loss on Table 4.4

In Table 4.4 performance values compared against previous all dataset training performances to evaluate the effects of feature selection in each classifier. As per the figures Naive Bayes, Logistic Regression and Random Forest classifiers show slight improvements on the accuracy while other classifiers show some accuracy drop with the refined feature subset.

4.4 Performance through Class Balancing

As identified in previous chapters, the class imbalance problem may produce a negative effect on classifier performances. Thus, this stage is attempting to evaluate the classifier performances by correcting the class imbalance problem to improve the classifiers. Here in Table 4.5, the performances will be evaluated over two (RUS and SMOTE) approach.

In the RUS approach majority class reduced by randomly selecting 383 data points to match the number of data points in the minority class. Then the total number of data points (Figure 4.2) will be reduced to 766 in the dataset. In order to apply RUS to the dataset SpreadSubsample filter can be utilized which is available under WEKA tool supervised instance filter collection.

Name: issues		Distinct: 2		Type: Nominal
Missing: 0 (0%)				Unique: 0 (0%)
No.	Label	Count	Weight	
1	FALSE	383	383.0	
2	TRUE	383	383.0	

Figure 4.1: RUS Class Balanced Dataset Statistics in WEKA

By applying the SMOTE filter in WEKA, minority class has been increased by 276% to match the majority class which was containing 1444 data points. Now complete dataset (Figure 4.3) consists of 2884 data points. In the WEKA tool's supervised instance filter collection list; the SMOTE filter is available to oversample the minority class.

Name: issues		Distinct: 2		Type: Nominal
Missing: 0 (0%)				Unique: 0 (0%)
No.	Label	Count	Weight	
1	FALSE	1444	1444.0	
2	TRUE	1440	1440.0	

Figure 4.2: SMOTE Class Balanced Dataset Statistics in WEKA

	ZeroR	Naive Bayes	Logistic Regression	Decision Tree	Random Forest	SVM
RUS						
Accuracy (%)	49.6084	68.2768	75.8486	80.5483	81.9843	63.9687
F-Measure	0.475	0.653	0.753	0.805	0.820	0.670
AUC	0.495	0.759	0.836	0.848	0.898	0.640
Kappa	-0.0078	0.3655	0.517	0.611	0.6397	0.2794
Precision	0.495	0.779	0.786	0.806	0.822	0.645
Recall	0.475	0.683	0.758	0.805	0.820	0.640
SMOTE						
Accuracy (%)	50.0693	65.4646	77.2191	86.8585	89.3551	69.8682
F-Measure	?	0.613	0.766	0.869	0.894	0.699
AUC	0.499	0.811	0.865	0.925	0.952	0.699
Kappa	0	0.3087	0.5442	0.7372	0.7871	0.3973
Precision	-	0.771	0.804	0.869	0.894	0.699
Recall	0.501	0.655	0.772	0.869	0.894	0.699

Table 4.5: Classifier performance measures on class balanced dataset

The performance figures in Table 4.5 shows that SMOTE class balancing supported to outperform most of the classifiers while RUS class balancing affected negatively to some of the previous classifier performances. According to the overall performance results of these classifier algorithms, the Random Forest algorithm has been showing a higher classification capacity which is around 39% improvement compared to the benchmarking ZeroR algorithm.

Meanwhile, the Decision Tree algorithm also shows closely followed performance gains with the improvement done to the dataset and training methods.

4.5 Ensemble Techniques

Ensemble technique is a way of combining multiple classifier algorithms together to achieve higher classification performances as a combined system. This approach of combining a set of classifier algorithms can support to manage the bias-variance tradeoff for better classification performance.

This study attempts to analyze the performance behavior of developer vulnerability prediction classifiers by employing two well-known ensemble techniques named as Boosting and Bagging (Bootstrap Aggregating).

4.5.1 Boosting

Boosting is an ensemble method which converting weak learner to strong learners by fitting them to a sequence. Boosting is a two-step method. It initially produces averagely performing models and improve (boost) their performance by combining them together as the second step. The predictions are combined through a weighted classification or weighted regression in order to produce the final result. The algorithm known as adaptive boosting (AdaBoost) is the most widely used algorithm as a boosting ensemble technique.

4.5.2 Bagging

Bagging is a way of reducing the variance of the learner prediction by generating additional data for training from the original dataset. Although increased dataset size doesn't improve the predictive power it supports to reduce the variance while narrowly tuning the prediction to an expected outcome.

	ZeroR	Naive	Logistic	Decision	Random	SVM
--	-------	-------	----------	----------	--------	-----

		Bayes	Regression	Tree	Forest	
Boosting (AdaBoost)						
Accuracy (%)	50.0693	65.4646	77.6006	88.3495	89.5284	65.8807
F-Measure	?	0.613	0.770	0.883	0.895	0.623
AUC	0.499	0.811	0.841	0.943	0.930	0.718
Kappa	0	0.3087	0.5518	0.767	0.7906	0.317
Precision	-	0.771	0.807	0.883	0.895	0.756
Recall	0.501	0.655	0.776	0.883	0.895	0.659
Bagging						
Accuracy (%)	50	66.4355	77.9126	88.835	89.251	65.9501
F-Measure	0.5	0.627	0.774	0.888	0.893	0.624
AUC	0.5	0.817	0.870	0.948	0.951	0.69
Kappa	0	0.3281	0.5581	0.7767	0.7857	0.3184
Precision	0.5	0.774	0.806	0.889	0.893	0.755
Recall	0.5	0.664	0.779	0.888	0.893	0.66

Table 4.6: Classifier performance measures on ensemble techniques

In this experiment, ensemble techniques were applied to the class balanced dataset using SMOTE which has shown better performance results in the previous step. According to the results, both ensemble techniques applied here produce a minor improvement on most classifier models. Random Forest is still the best performing classifier with closely followed performance gains on both ensemble techniques evaluated here.

4.6 Experiment Summary

The main goal of the experiments conducted in this chapter is to answer the second research question of this study. The study used some general classifier algorithms which have been utilized to classification prediction models in the related domain as also suggested in the

literature. According to the performance measures of the classifier model evaluated, it has been clearly identifiable that developer-centric vulnerability can be predictable with better performance by their behavioral characteristics. Below figures show the ROC figures of final classifier performance respectively to the benchmarking Zero - R classifier. The figures are captured through the experiments done in the WEKA tool.

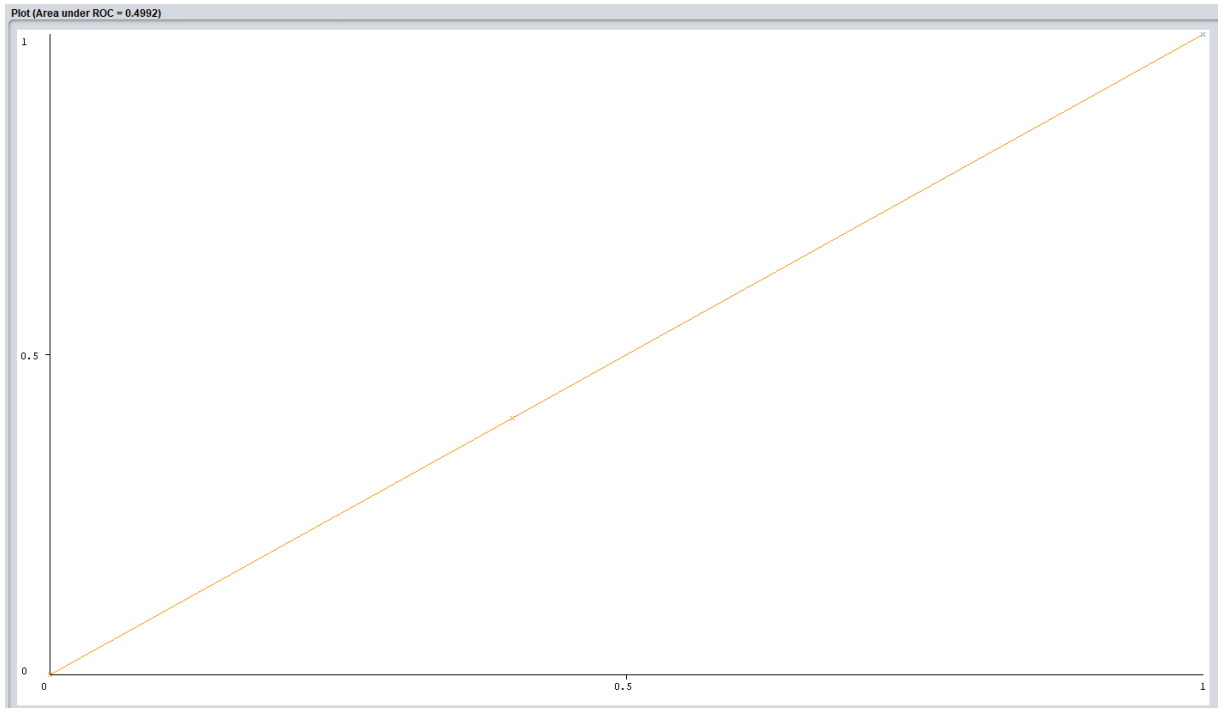


Figure 4.3: ROC curve of Zero - R Classifier

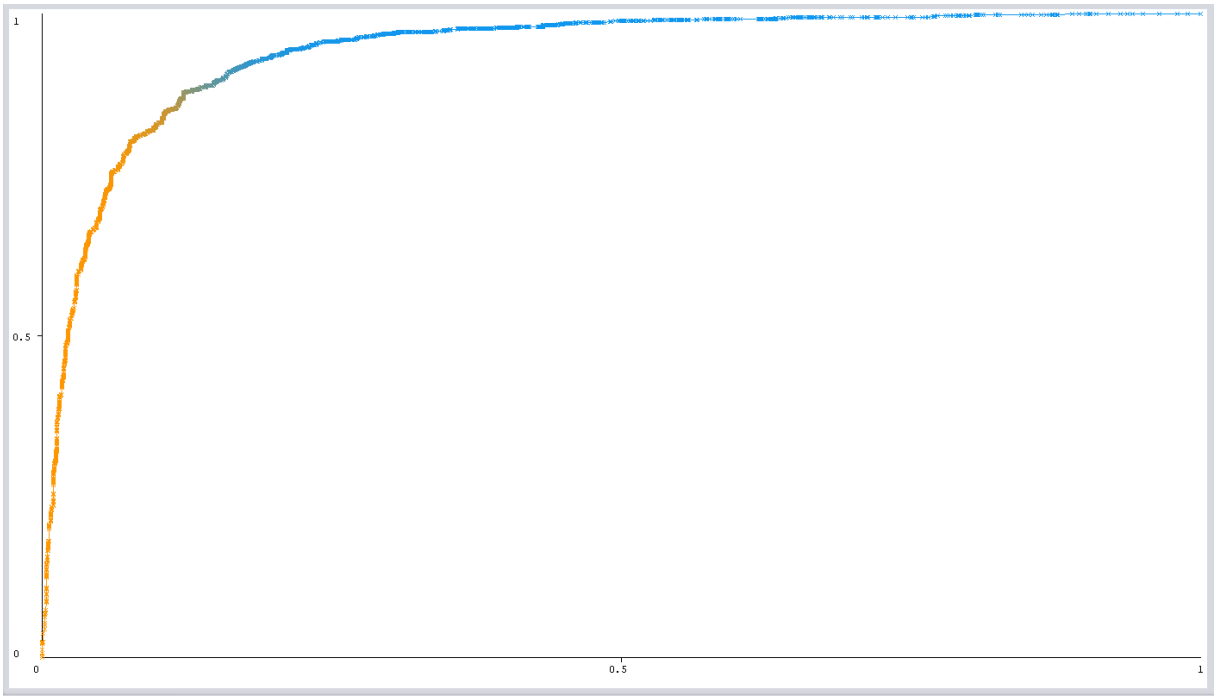


Figure 4.4: ROC curve of Random Forest Classifier

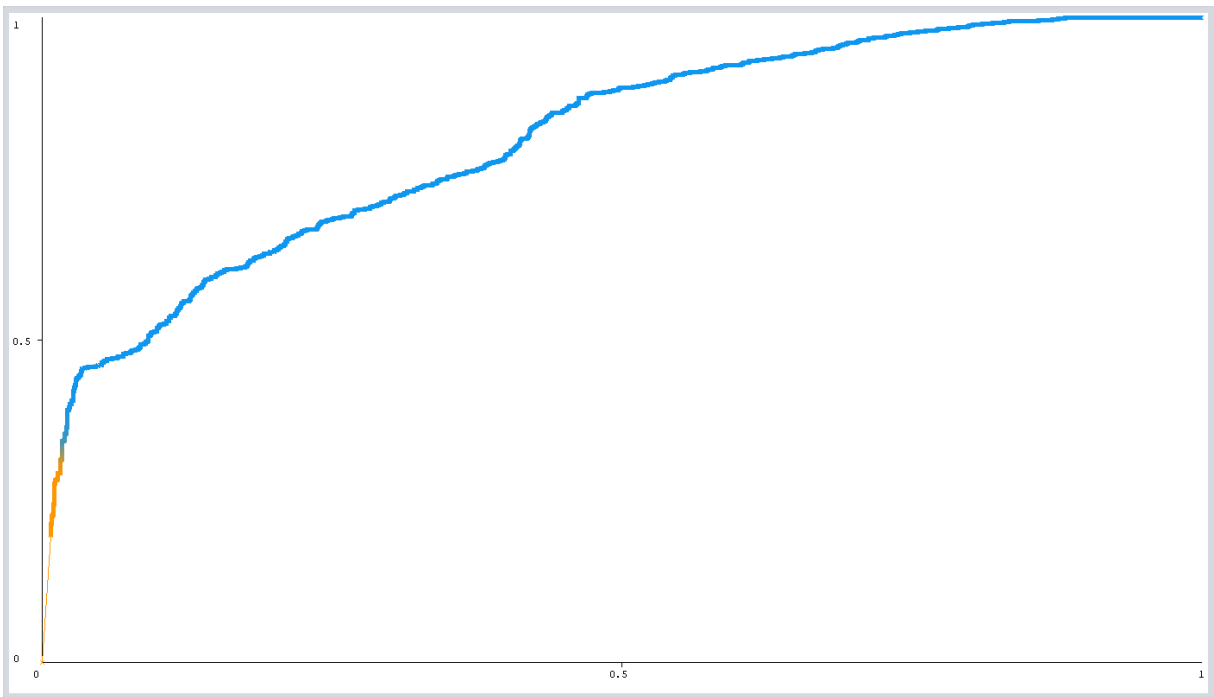


Figure 4.5: ROC curve of Naive Bayes Classifier

Chapter 05: Discussion

This chapter intends to discuss the abstract summary of the works that are done in this study so far and the final outcomes of this study. Further, it will review the future research areas that can be done based on this study.

5.1 Summary

Many related studies that have been done on this domain area are mostly attempted to focus on predicting defect/vulnerabilities in software components. This research work motivated by some of the preliminary attempts done by several authors who attempted to see this prediction problem from the perspective of developers and attempted to identify the human factors which can be correlated with defects and vulnerabilities in software components. Based on the results and hints from those research works; this study was conducted in order to address two main research questions that were prevailing under the software defect/vulnerability prediction domain.

First, this study attempted to identify if there exist any superficial features of developer behavior that can be mined through software repositories without analyzing the syntactic level of source code which can be correlated to intentional or unintentional vulnerable contributions done by developers. Secondly, the study focused on identifying algorithms and techniques that can be applied in order to train a better performing classification model for developer-centric vulnerability prediction based on the identified techno-behavioral feature vector of developers.

Considering literature reviews, available repository mining tools and other project constraints a candidate feature vector for developer behavior identified. Then 10 open source software repositories are selected from GitHub social coding platform based on several criteria considered in Chapter 02. All repositories mined using identified and custom-built mining and analyzing tools. The mined and preprocessed dataset contained data points of 1827 developers

with 383 developers have contributed at least one vulnerable code while others haven't contributed for any security vulnerable code to the repositories.

To analyze the prediction performance of developer-centric vulnerability by the selected feature vector, it has been decided to evaluate the performances with 4 different classifier models while keeping the Zero - R classifier performances as the benchmarking performance values. The study has utilized Naive Bayes, Logistic Regression, Decision Tree, Random Forest and Support Vector Machine algorithms to train the classifier models and applied feature selection, class balancing and ensemble techniques to analyze their effects on classifier performances. As per the results, a random forest classifier performed best with all the techniques applied produced 89% classification accuracy performance with 10 - fold cross-validated training.

The highest performance was recorded with the Random Forest algorithm when it applied with the AdaBoost ensemble technique while the feature selected dataset with SMOTE class balancing is used for model training. It is more than 39% performance in accuracy gain compared to the benchmarking Zero-R classifier performance. Meanwhile, it can be noted that the performance of the Decision Tree algorithm trained models are closely following the performance accuracies of the Random Forest algorithm. The obvious reason for this could be both algorithms are classified into tree algorithms. Surprisingly; SVM algorithm didn't produce a significant performance on developer-centric vulnerability predictions

Finally, it has been identified that potential contributions of vulnerable codes to software projects can be predicted through analyzing the developer's techno-behavioral features. This is more beneficial in understanding each developer-centric vulnerability (whether a developer is contingent to contribute security vulnerable code fragments) when there is limited or restricted access to the source code repository. Further, it has been identified that general classifier models can predict developer-centric vulnerability with better accuracy, precision and recall with the application of general ML techniques.

5.2 Findings

As identified through the feature selection phase, there were several feature metrics have been eliminated from the feature vector due to an insignificant correlation between the target

variable. But 9 developer features retained through the feature elimination process. Thus, it is important to find out and discuss the relationship between developer-centric vulnerability. In order to get a better idea on the relationship with developer-centric vulnerability, it's worth to look into the decision tree of decision tree classifier.

REPTree

=====

```

lines_inserted < 133.15
|  lines_of_code_count < 58.14
|  |  lines_deleted < 50.5 : FALSE (613.51/12) [309.59/15]
|  |  lines_deleted >= 50.5
|  |  |  frequent_commit_hour < 8.5 : FALSE (11.29/0) [3.48/0]
|  |  |  frequent_commit_hour >= 8.5
|  |  |  |  frequent_commit_hour < 12.56 : TRUE (4.65/0) [1/0]
|  |  |  |  frequent_commit_hour >= 12.56 : FALSE (13.94/1.65) [3.83/1.82]
|  |  lines_of_code_count >= 58.14
|  |  |  lines_inserted < 35.1 : FALSE (28.53/1) [12.94/3]
|  |  |  lines_inserted >= 35.1
|  |  |  |  lines_of_code_count < 136.61 : FALSE (92.58/29.6) [50.24/11.15]
|  |  |  |  lines_of_code_count >= 136.61
|  |  |  |  |  comment_percentage < 18.47 : TRUE (11.44/0.37) [5.28/2.26]
|  |  |  |  |  comment_percentage >= 18.47
|  |  |  |  |  |  lines_of_code_count < 847 : FALSE (2.08/0.01) [3.05/1]
|  |  |  |  |  |  lines_of_code_count >= 847 : TRUE (3.79/0.77) [1.41/0.41]
lines_inserted >= 133.15
|  lines_of_code_count < 338
|  |  developer_closeness < 0
|  |  |  lines_of_code_count < 98 : FALSE (49.97/3.73) [19.23/2.19]
|  |  |  lines_of_code_count >= 98
|  |  |  |  comment_percentage < 18.02
|  |  |  |  |  developer_age < 148
|  |  |  |  |  |  lines_of_code_count < 278
|  |  |  |  |  |  |  average_cyclomatic_complexity < 200.76
|  |  |  |  |  |  |  |  developer_age < 60.2 : FALSE (11.85/2.76) [5.96/2.05]
|  |  |  |  |  |  |  |  developer_age >= 60.2 : TRUE (25.51/11.3) [6.56/3.22]
|  |  |  |  |  |  |  |  |  average_cyclomatic_complexity >= 200.76
|  |  |  |  |  |  |  |  |  |  lines_inserted < 170.54 : TRUE (7.22/0.12) [2/2]
|  |  |  |  |  |  |  |  |  |  lines_inserted >= 170.54
|  |  |  |  |  |  |  |  |  |  |  lines_inserted < 207.05 : FALSE (2.48/0) [2.37/0]
|  |  |  |  |  |  |  |  |  |  |  lines_inserted >= 207.05
|  |  |  |  |  |  |  |  |  |  |  |  average_cyclomatic_complexity < 515.58 : TRUE (7.55/0.6)
|  |  |  |  |  |  |  |  |  |  |  |  |  [3.47/1.15]
|  |  |  |  |  |  |  |  |  |  |  |  |  |  average_cyclomatic_complexity >= 515.58 : FALSE
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  (2.45/1.25) [1.29/0.1]
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  lines_of_code_count >= 278 : TRUE (6.99/0.3) [5.77/0.74]
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  developer_age >= 148
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  comment_percentage < 10.85
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  comment_percentage < 9.98 : FALSE (9.55/2.14) [9.25/6.04]
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  comment_percentage >= 9.98 : TRUE (2.12/0.09) [1.06/0.05]

```

```

| | | | | | | comment_percentage >= 10.85 : FALSE (6.11/0.09) [1.16/0.02]
| | | | | | | comment_percentage >= 18.02
| | | | | | | lines_deleted < 12.5 : FALSE (6.9/0) [8.28/0]
| | | | | | | lines_deleted >= 12.5
| | | | | | | | lines_inserted < 184
| | | | | | | | | comment_percentage < 26.41 : FALSE (4.7/1) [1/0]
| | | | | | | | | comment_percentage >= 26.41 : TRUE (4.09/0.09) [0/0]
| | | | | | | | | lines_inserted >= 184
| | | | | | | | | | lines_of_code_count < 255.5 : FALSE (15.69/0.35) [8.82/1.68]
| | | | | | | | | | lines_of_code_count >= 255.5
| | | | | | | | | | | lines_of_code_count < 275 : TRUE (2.15/0.1) [1.06/0.05]
| | | | | | | | | | | lines_of_code_count >= 275
| | | | | | | | | | | | lines_inserted < 2348 : FALSE (9.62/1.18) [1.33/0.04]
| | | | | | | | | | | | lines_inserted >= 2348 : TRUE (2.22/0.14) [0.03/0]
| | | | | | | | | | | | | developer_closeness >= 0
| | | | | | | | | | | | | | lines_deleted < 19.66 : FALSE (18.95/5.41) [8.35/2]
| | | | | | | | | | | | | | lines_deleted >= 19.66
| | | | | | | | | | | | | | | developer_age < 222.9 : TRUE (102.02/18.58) [55.94/7.73]
| | | | | | | | | | | | | | | developer_age >= 222.9 : FALSE (4.03/0.15) [4.09/1.05]
| | | | | | | | | | | | | | | | lines_of_code_count >= 338
| | | | | | | | | | | | | | | | | lines_deleted < 289.33 : TRUE (230.32/53.44) [119.86/32.66]
| | | | | | | | | | | | | | | | | lines_deleted >= 289.33
| | | | | | | | | | | | | | | | | | frequent_commit_hour < 22.99
| | | | | | | | | | | | | | | | | | | developer_age < 12.81 : TRUE (17.54/5.17) [4.17/2.06]
| | | | | | | | | | | | | | | | | | | developer_age >= 12.81
| | | | | | | | | | | | | | | | | | | | lines_deleted < 3810.14 : TRUE (360.61/22.81) [193.1/18.4]
| | | | | | | | | | | | | | | | | | | | lines_deleted >= 3810.14
| | | | | | | | | | | | | | | | | | | | | average_cyclomatic_complexity < 864.87
| | | | | | | | | | | | | | | | | | | | | developer_closeness < 0
| | | | | | | | | | | | | | | | | | | | | | lines_deleted < 11976.71
| | | | | | | | | | | | | | | | | | | | | | | lines_deleted < 9240.18 : TRUE (6.61/0.01) [7/1]
| | | | | | | | | | | | | | | | | | | | | | | lines_deleted >= 9240.18 : FALSE (2.5/1.09) [0/0]
| | | | | | | | | | | | | | | | | | | | | | | lines_deleted >= 11976.71 : TRUE (13.19/0.02) [0.11/0.01]
| | | | | | | | | | | | | | | | | | | | | | | | developer_closeness >= 0 : TRUE (198.41/0.29) [96.81/2.15]
| | | | | | | | | | | | | | | | | | | | | | | | | average_cyclomatic_complexity >= 864.87 : TRUE (3.32/1.02) [0.09/0]
| | | | | | | | | | | | | | | | | | | | | | | | | | frequent_commit_hour >= 22.99
| | | | | | | | | | | | | | | | | | | | | | | | | | | average_cyclomatic_complexity < 248.7 : TRUE (2.78/0.74) [1.02/0]
| | | | | | | | | | | | | | | | | | | | | | | | | | | average_cyclomatic_complexity >= 248.7 : FALSE (2.74/0.01) [2/1]

```

Size of the tree : 79

Experiment 5.1: Generated Decision Tree of Decision Tree Classifier

According to the decision tree, it can be identified that what are the possible ways of determining whether a developer has a potential vulnerability or not. As per the decision tree rules, it seems to be difficult in identifying specific threshold value for each feature separately. Therefore, the selection of developer-centric vulnerability is based on the combined output of several threshold values.

But, the class distribution of each feature can produce a general understanding of how these features correlate with developer-centric vulnerability. Below boxplots describes the data distributions of each feature.

5.2.1 Number of Rows

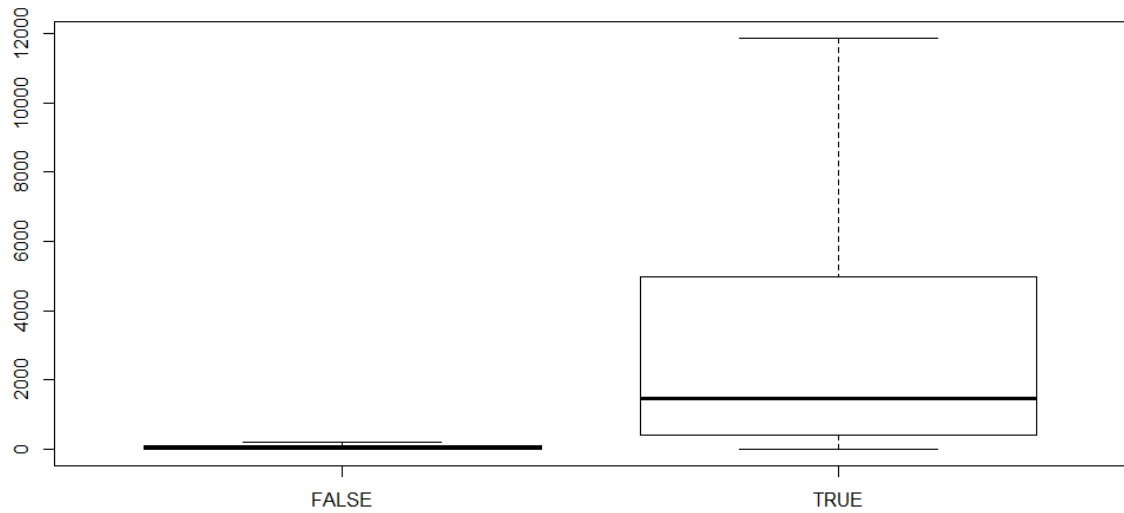


Figure 5.1: Number of Rows Distribution

The figure shows that most developer who has contributed vulnerabilities done a higher number of code lines in the project. This could be an obvious reason since mostly software defects are also proportional to the lines of code.

5.2.2 Average Commit Interval

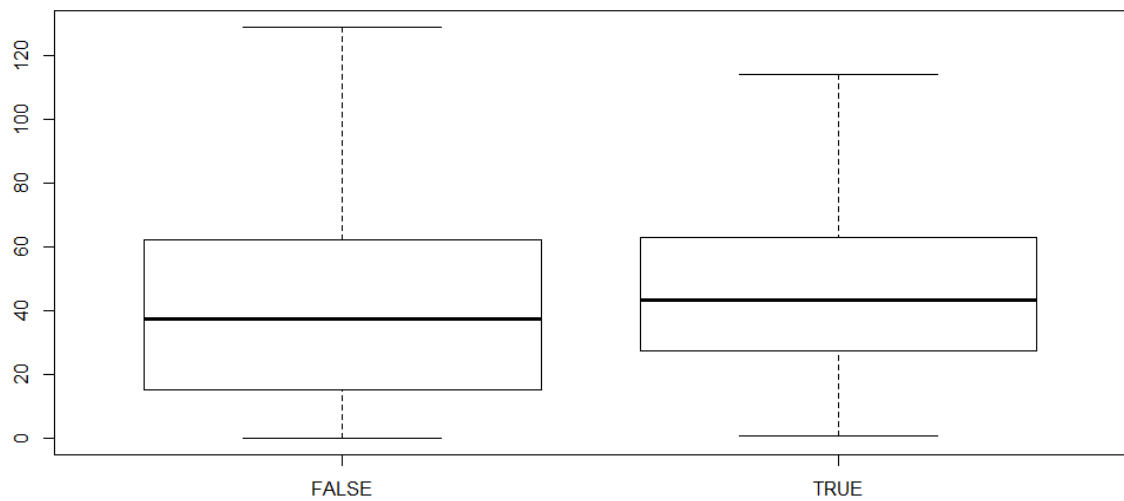


Figure 5.2: Average Commit Interval Distribution

According to the boxplot even though both classes have closely similar median values; as per the data distribution comparison, it can be identified that the majority of the developers who have done their commits in lesser time intervals are the ones who have most security vulnerable codes. Thus, presumably, these developers may not take time to evaluate and analyze their source code for security vulnerabilities and push their code changes in lesser time intervals.

5.2.3 Frequent Commit Hour

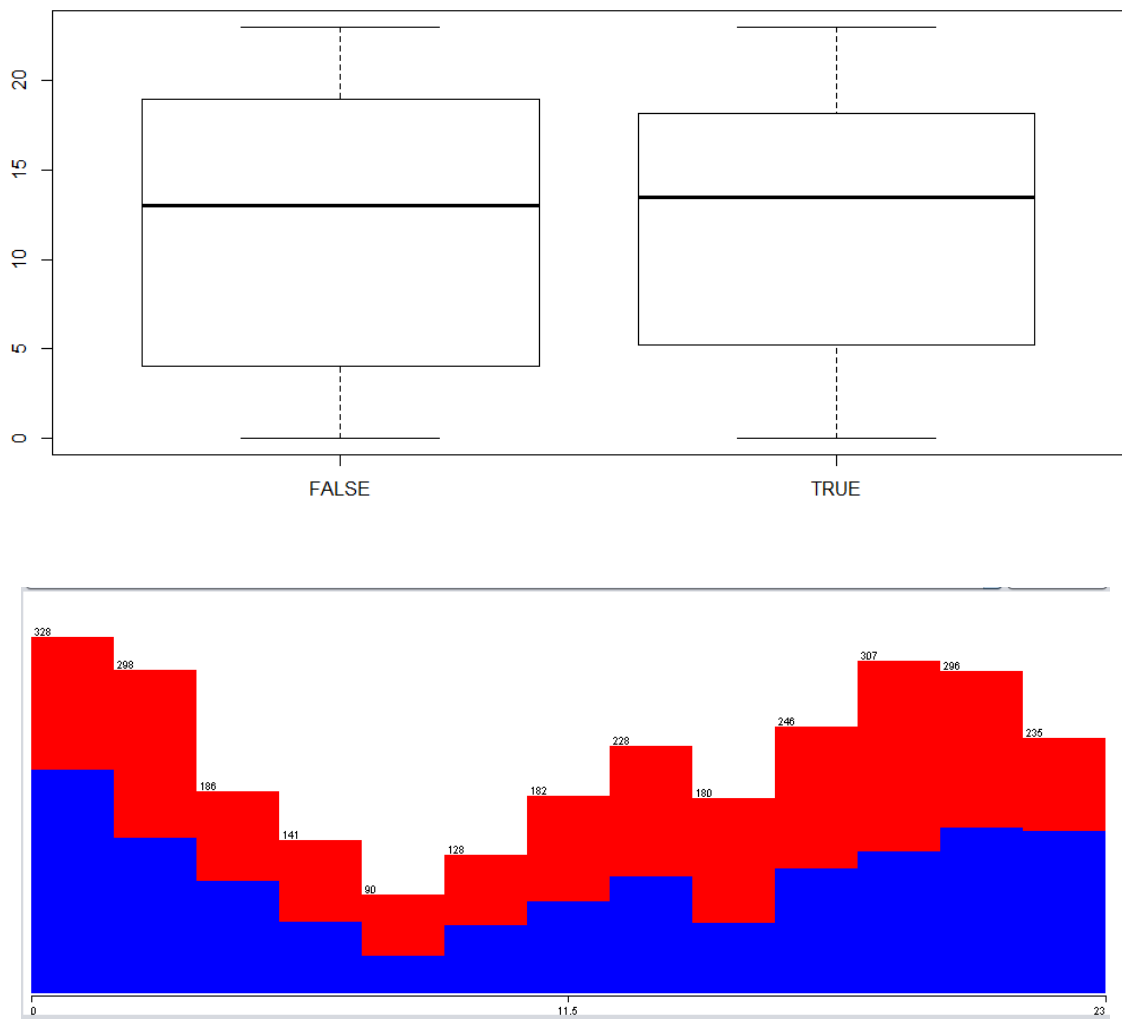


Figure 5.3: Frequent Commit Hour Distribution Histogram

It's a little bit difficult to determine the developer behavior based on the frequent commit hour distribution in the box plot. But the distribution histogram can produce a better idea on the developer-centric vulnerability. According to the distribution histogram frequent commit hour can be identified as a kind of time series data where the developers who have done commits in between 5 am to 11 am having less potential to do vulnerable codes into repositories.

5.2.4 Lines Inserted

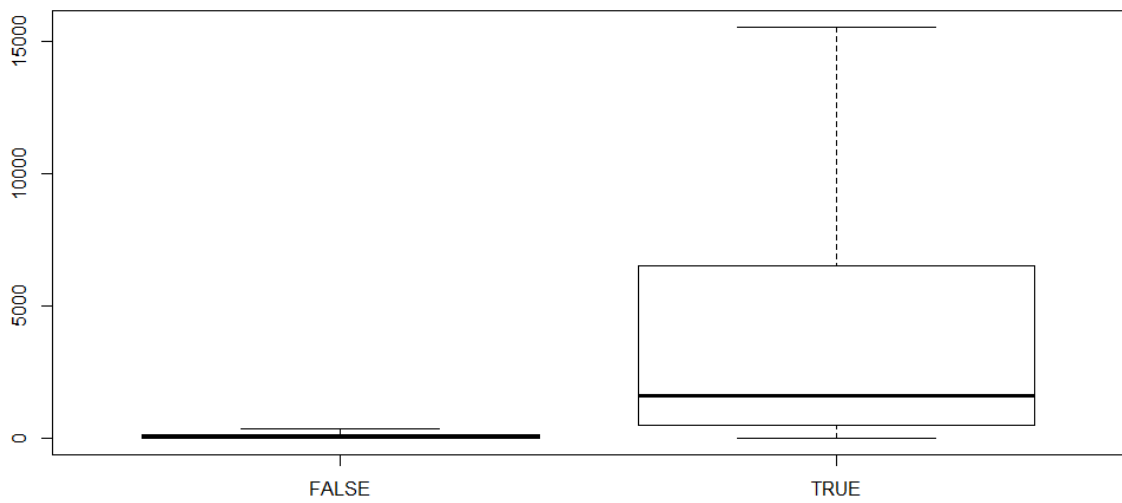


Figure 5.4: Lines Inserted Distribution

Similar as lines of code developers with higher number of lines inserted are more contingent to do security vulnerabilities to the code.

5.2.5 Lines Deleted

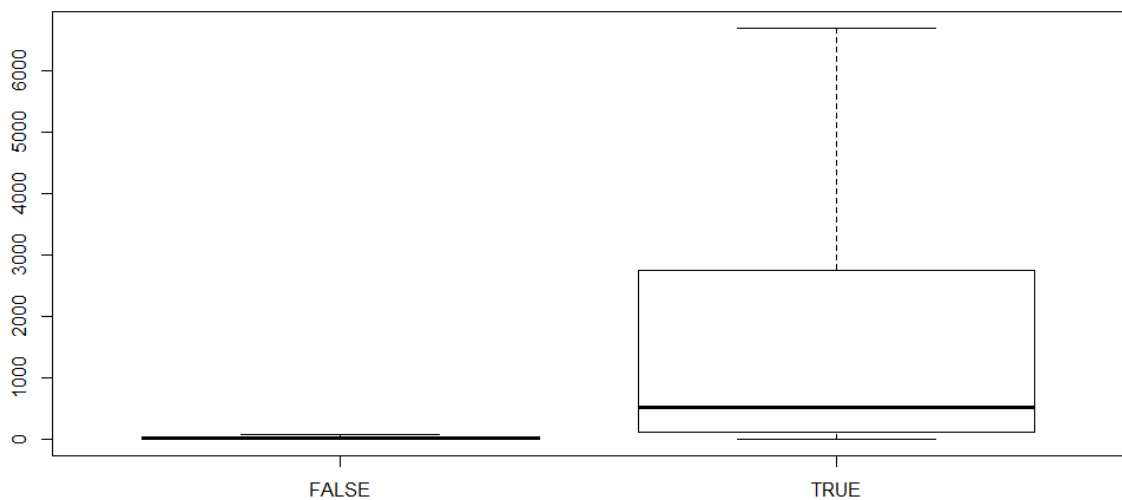


Figure 5.5: Lines Deleted Distribution

Similar as lines of code developers with higher number of lines deleted are more contingent to do security vulnerabilities to the code.

5.2.6 Developer Age

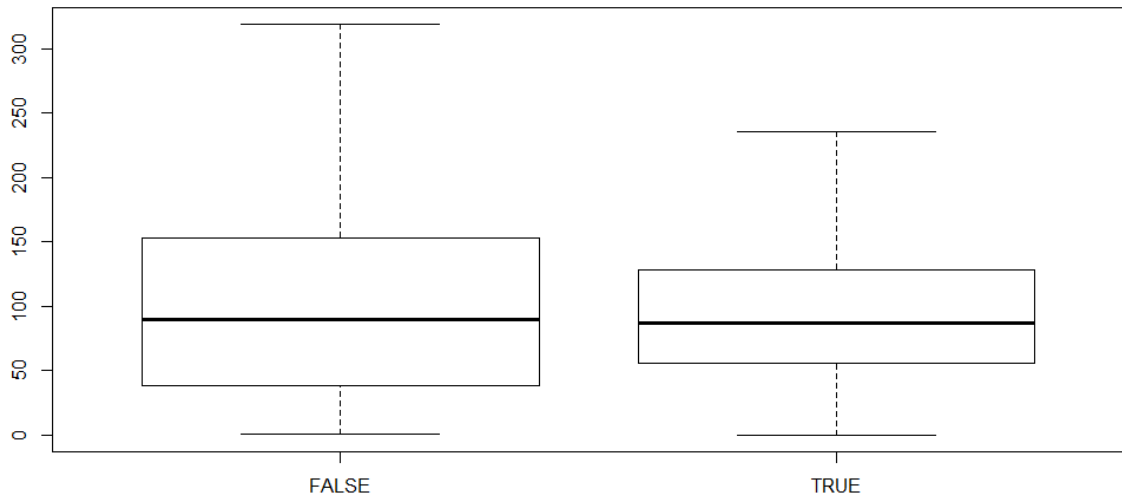


Figure 5.6: Developer Age Distribution

This is another important observation in the dataset that mined from repositories. Although the median values are closely similar; the interpretation of the box plot data distribution suggests that the developers who have done security vulnerable codes have more centering towards lesser developer contribution age with lesser interquartile range. This observation concludes the developers who are new to the projects, do more vulnerable code contributions.

5.2.7 Average Cyclomatic Complexity

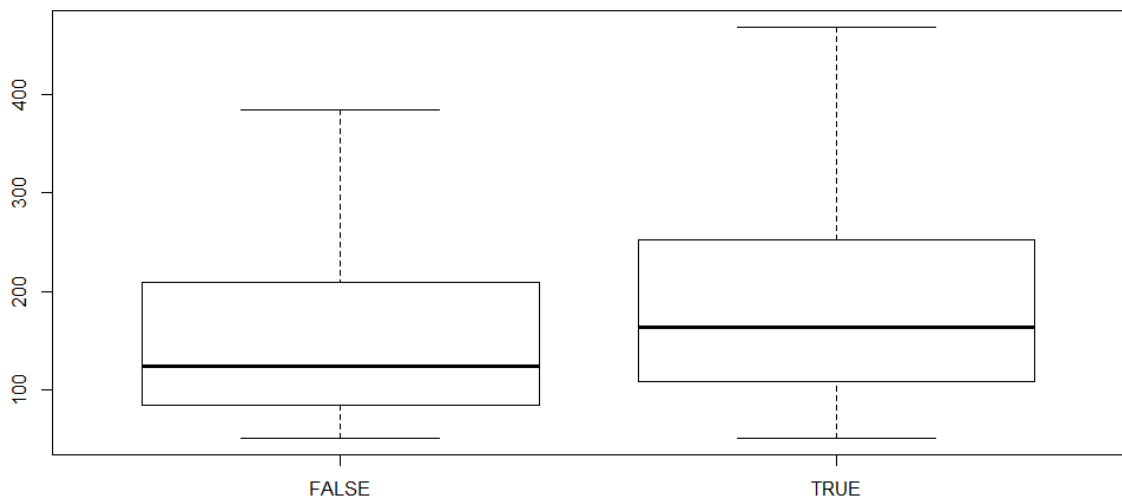


Figure 5.7: Average Cyclomatic Complexity Distribution

According to the box plot most of the developers who tend to work with lesser average complexity codes are having minor tendency to do vulnerable codes than others.

5.2.8 Comment Percentage

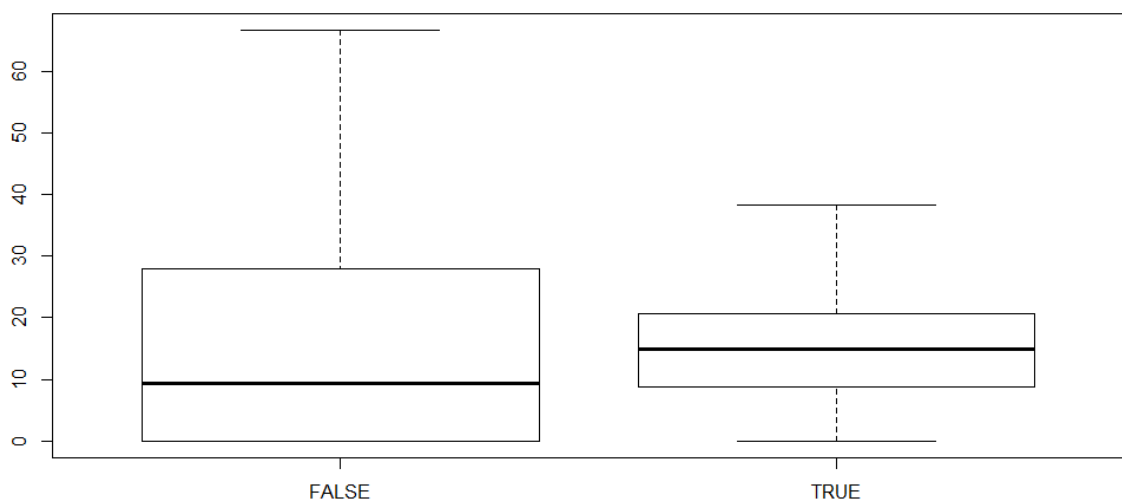


Figure 5.8: Comment Percentage Distribution

As identifiable in the plot, developers who has done less comment percentage in their source contributions have done more vulnerable code contributions compared to the developers who has done more comments.

5.2.9 Developer Closeness

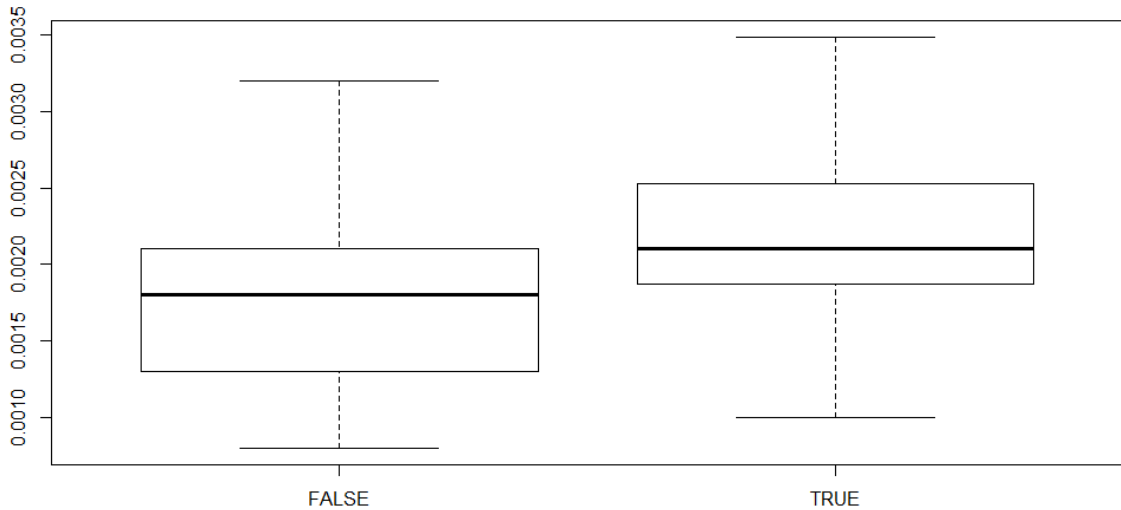


Figure 5.9: Developer Closeness Distribution

Developer closeness describes how much a developer is interconnected with other developers. Developers with higher closeness distance averages mean they are not closely connected and not collaborating enough with other developers. As per the plot, it can be clearly identified that most of the developers with higher closeness distance have contributed more security vulnerable source codes to the repositories. Thus, it can be concluded that the developers who don't closely collaborate with the other developers tend to do more security vulnerable codes.

5.2.10 Summary

According to the evaluated graphs, it can be identified that most of the features in the feature vector shows positive or negative correlation patterns with developer-centric vulnerability except frequent commit hour.

Feature	Correlation
Number of Rows	Positive
Average Commit Interval	Negative
Frequent Commit Hour	-
Lines Inserted	Positive
Lines Deleted	Positive
Developer Age	Negative
Average Cyclomatic Complexity	Positive
Comment Percentage	Negative
Developer Closeness	Positive

Table 5.1: Feature Vector Correlation Summary

In Table 5.1, a feature is identified as positively correlated if the developer-centric vulnerability increases with the increment of the feature value and negatively correlated if developer-centric vulnerability decreases with the increment of the feature value. Frequent Commit Hour feature doesn't have an identifiable linear correlation.

5.3 Future Work

The outcomes of this study confirm that the vulnerability of individual developers can be predicted by their behavioral patterns on the development process. Due to the limitations and constraints, the number of behavioral features analyzed in this study was limited or elementary. But there exist many other developer behavior related features that are complex to mine from software repositories yet effective on developer-centric vulnerability prediction. Thus, this research work can be extended further through the development of advanced repository mining tools and the identification of more effective techno-behavioral features.

Further, this study attempts to classify developers into two classes which are vulnerable and non-vulnerable. But there is a potential that developers can be classified into more classes which describes the severity levels of the developer-centric vulnerability of each developer. Thus, it allows doing a more detailed comparison between two or more developers to support a flexible assessment of developer-centric vulnerability.

This study trained and evaluated classifications using general purpose classifier algorithms and techniques. As discussed previously, some of the classifiers performed well in this setup and produced better classification accuracy. But there are many opportunities to do future works to improve these classification performances with the application of more focused and customized techniques on this classification problem.

References

- [1]. The Department for Business, Innovation and Skills (BIS) . (2014). INFORMATION SECURITY BREACHES SURVEY 2014 | technical report. Available: <https://www.pwc.co.uk/assets/pdf/cyber-security-2014-technical-report.pdf>. Last accessed 1st July 2017.
- [2]. United States General Accounting Office. (1992). MISSION-CRITICAL SYSTEMS Defense Attempting to Address Major Software Challenges. Available: <http://www.gao.gov/assets/220/217352.pdf>. Last accessed 1st July 2017.
- [3]. I. Gorton (2011). Essential Software Architecture. 2nd ed. USA: Springer-Verlag Berlin Heidelberg. P23-38.
- [4]. Dan Suceava. (2005). The Importance of Defect Tracking in Software Development. Available: https://www.axosoft.com/downloads/axowp_importance_of_defect_tracking.pdf. Last accessed 2nd July 2017.
- [5]. "Build Software Better, Together". GitHub. N.p., 2017. Web. 28 May 2017.
- [6]. van der Stock, A., Gonçalves, I.R. and Correa, J. (2015). OWASP top Ten cheat sheet. Available: https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet. Last accessed 11th July 2017
- [7]. Shin, Y. and Williams, L., 2013. Can traditional fault prediction models be used for vulnerability prediction?. Empirical Software Engineering, 18(1), pp.25-59.
- [8]. Shin, Y., Meneely, A., Williams, L. and Osborne, J.A., 2011. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Transactions on Software Engineering, 37(6), pp.772-787.
- [9]. Walden, J., Stuckman, J. and Scandariato, R., 2014, November. Predicting vulnerable components: Software metrics vs text mining. In Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on (pp. 23-33). IEEE.
- [10]. Zimmermann, T., Nagappan, N. and Williams, L., 2010, April. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In Software Testing, Verification and Validation (ICST), 2010 Third International Conference on (pp. 421-428). IEEE.
- [11]. Jiang, T., Tan, L. and Kim, S., 2013, November. Personalized defect prediction. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (pp. 279-289). IEEE Press.
- [12]. Ostrand, T.J., Weyuker, E.J. and Bell, R.M., 2010, September. Programmer-based fault prediction. In Proceedings of the 6th International Conference on Predictive Models in Software Engineering (p. 19). ACM.

- [13]. Chandrashekar, G. and Sahin, F., 2014. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1), pp.16-28.
- [14]. Guyon, I. and Elisseeff, A., 2003. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar), pp.1157-1182.
- [15]. The statistical analysis tool for git repositories. (2017). Ejwa Software. Available: <https://github.com/ejwa/gitinspector>. Last accessed 9th September 2017.
- [16]. Five Data Characteristics: Building a Great Data Analytics Solution. (2015). Ashish Soni. Available: <https://www.linkedin.com/pulse/five-data-characteristics-building-great-analytics-solution-soni>. Last accessed 9th September 2017.
- [17]. Tang, J., Alelyani, S. and Liu, H., 2014. Feature selection for classification: A review. *Data Classification: Algorithms and Applications*, p.37.
- [18]. Hall, M.A., 2000. Correlation-based feature selection of discrete and numeric class machine learning.
- [19]. "dropwizard/metrics", GitHub, 2017. [Online]. Available: <https://github.com/dropwizard/metrics>. [Accessed: 12- Oct- 2017].
- [20]. "igraph R package", Igraph.org, 2017. [Online]. Available: <http://igraph.org/r/>. [Accessed: 12- Oct- 2017].
- [21]. Weicheng, Y., Beijun, S. and Ben, X., 2013, December. Mining GitHub: Why Commit Stops--Exploring the Relationship between Developer's Commit Pattern and File Version Evolution. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific (Vol. 2, pp. 165-169)*. IEEE.
- [22] HEALTH CARE INDUSTRY CYBERSECURITY TASK FORCE. (2017). REPORT ON IMPROVING CYBERSECURITY IN THE HEALTH CARE INDUSTRY. Available: <https://www.phe.gov/Preparedness/planning/CyberTF/Documents/report2017.pdf>. Last accessed 28th July 2018.
- [23] Ganganwar, V., 2012. An overview of classification algorithms for imbalanced datasets. *International Journal of Emerging Technology and Advanced Engineering*, 2(4), pp.42-47.
- [24] Chawla, N.V., Japkowicz, N. and Kotcz, A., 2004. Special issue on learning from imbalanced data sets. *ACM Sigkdd Explorations Newsletter*, 6(1), pp.1-6.
- [25] Tang, Y., Zhang, Y.Q., Chawla, N.V. and Krasser, S., 2009. SVMs modeling for highly imbalanced classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(1), pp.281-288.

- [26] Liu, W., Chawla, S., Cieslak, D.A. and Chawla, N.V., 2010, April. A robust decision tree algorithm for imbalanced data sets. In *Proceedings of the 2010 SIAM International Conference on Data Mining* (pp. 766-777). Society for Industrial and Applied Mathematics.
- [27] Chawla, N.V., Bowyer, K.W., Hall, L.O. and Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16, pp.321-357.
- [28] Van Hulse, J., Khoshgoftaar, T.M. and Napolitano, A., 2007, June. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning* (pp. 935-942). ACM.
- [29] Dietterich, T.G., 2000, June. Ensemble methods in machine learning. In *International workshop on multiple classifier systems* (pp. 1-15). Springer, Berlin, Heidelberg.
- [30] Kotsiantis, S.B., Zaharakis, I. and Pintelas, P., 2007. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160, pp.3-24.
- [31] Fortmann-Roe, S., 2012. Understanding the bias-variance tradeoff.
- [32] Wurster, G. and Van Oorschot, P.C., 2009, August. The developer is the enemy. In *Proceedings of the 2008 New Security Paradigms Workshop* (pp. 89-97). ACM.
- [33] Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K.I. and Nakamura, M., 2010, September. An analysis of developer metrics for fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering* (p. 18). ACM.
- [34] Meneely, A., Williams, L., Snipes, W. and Osborne, J., 2008, November. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (pp. 13-23). ACM.
- [35] Witten IH. 2014. More data mining with Weka. The university of Waikato, <http://www.cs.waikato.ac.nz/ml/weka/mooc/moredataminingwithweka/slides/Class4-MoreDataMiningWithWeka-2014.pdf>
- [36] Zhang, H., 2009, September. An investigation of the relationships between lines of code and defects. In *2009 IEEE International Conference on Software Maintenance* (pp. 274-283). IEEE.
- [37] Krsul, I.V., 1998. *Software vulnerability analysis*. West Lafayette, IN: Purdue University.

Appendix

Appendix A: Gitinspector JSON Output Format Sample

```
{
  "gitinspector": {
    "version": "0.5.0dev",
    "repository": "metrics.git",
    "report_date": "2017/08/19",
    "changes": {
      "message": "The following historical commit information, by author, was found",
      "authors": [
        {
          "name": "xxxx",
          "email": "xxxx@example.com",
          "gravatar": "",
          "commits": 2,
          "insertions": 15,
          "deletions": 2,
          "percentage_of_changes": 0.02
        }, {
          "name": "xxxx",
          "email": "xxxx@example.com",
          "gravatar": "",
          "commits": 1,
          "insertions": 1,
          "deletions": 1,
          "percentage_of_changes": 0.00
        }
      ]
    },
    "blame": {
      "message": "Below are the number of rows from each author that have survived
and are still intact in the current revision",
      "authors": [
        {
          "name": "xxxx",
          "email": "xxxx@example.com",
          "gravatar": "",
          "rows": 14,
          "stability": 93.3,
          "age": 221.7,
          "percentage_in_comments": 0.00
        }, {
          "name": "xxxx",
          "email": "xxxx@example.com",
          "gravatar": "",
          "rows": 6,
          "stability": 100.0,

```



```

        "age": 150.0,
        "percentage_in_comments": 0.00
    ]]
},
"timeline": {
    "message": "The following history timeline has been gathered from the
repository",
    "period_length": "week",
    "periods": [
        {
            "name": "2010W49",
            "authors": [
                {
                    "name": "xxxx",
                    "email": "xxxx@example.com",
                    "gravatar": "",
                    "work": "-+++++"
                }
            ],
            "modified_rows": 1215
        }, {
            "name": "2010W50",
            "authors": [
                {
                    "name": "xxxx",
                    "email": "xxxx@example.com",
                    "gravatar": "",
                    "work": "--++++"
                }
            ],
            "modified_rows": 1148
        }
    ],
},
"metrics": {
    "violations": [
        {
            "type": "estimated-lines-of-code",
            "file_name": "metrics-
jmx/src/main/java/com/codahale/metrics/jmx/JmxReporter.java",
            "value": 676
        }, {
            "type": "cyclomatic-complexity",
            "file_name": "metrics-
core/src/main/java/com/codahale/metrics/MetricRegistry.java",
            "value": 135
        }, {
            "type": "cyclomatic-complexity",
            "file_name": "metrics-
core/src/test/java/com/codahale/metrics/InstrumentedScheduledExecutorServiceTest.java",
            "value": 128
        }
    ],
},
"responsibilities": {
    "message": "The following responsibilities, by author, were found in the current
revision of the repository (comments are excluded from the line count, if possible)",

```

```

    "authors": [
      {
        "name": "xxxx",
        "email": "xxxx@example.com",
        "gravatar": "",
        "files": [
          {
            "name": "metrics-
core/src/test/java/com/codahale/metrics/MetricRegistryTest.java",
            "rows": 8
          }
        ],
      }, {
        "name": "xxxx",
        "email": "xxxx@example.com",
        "gravatar": "",
        "files": [
          {
            "name": "metrics-
servlet/src/main/java/com/codahale/metrics/servlet/AbstractInstrumentedFilter.java",
            "rows": 6
          }
        ]
      }
    ],
    "extensions": {
      "message": "The extensions below were found in the repository history",
      "used": [ "java" ],
      "unused": [ "*", "ai", "conf", "css", "erb", "handlers", "html", "io", "js",
"less", "markdown", "md", "prefs", "properties", "py", "r", "rb", "rst", "scala", "schemas",
"xml", "xsd", "yaml" ]
    }
  }
}

```