Software Defect Severity Level Prediction Using Machine Learning Techniques

By: Miraj Zekiyi Jemal

A Thesis Submitted to the Department of Computer Science and Engineering

School of Electrical Engineering and Computing.

Presented in Partial Fulfillment of Requirement for the Degree of Masters of Science in Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

Adama, Ethiopia,

January 2021

Software Defect Severity Level Prediction Using Machine Learning Techniques

By: Miraj Zekiyi Jemal

Advisor: Mesfin Abebe(PHD)

A Thesis Submitted to the Department of Computer Science and Engineering

School of Electrical Engineering and Computing.

Presented in Partial Fulfillment of Requirement for the Degree of Masters of Science in Computer Science and Engineering

Office of Graduate Studies

Adama Science and Technology University

Adama, Ethiopia,

January 2021

Declaration

I hereby declare that this MSC thesis is my original work and has not been presented as a partial degree requirement in other university, and all sources of materials for the thesis have been duly acknowledged.

Name: Miraj Zekiyi Jemal

Signature: _____

This MSc thesis has been submitted for examination with my approval as a thesis advisor.

Name: Mesfin Abebe(PhD.)

Signature: _____

Date of Submission:_____

Approval of Board of Examiners

We, the undersigned, members of the Board of Examiners of the final open defense by Miraj Zekiyi have read and evaluated his/her thesis entitled" Software Defect Severity Level Prediction Using Machine Learning Techniques" and examined the candidate. This is therefore, to confirm that the thesis has been accepted in partial fulfillment of the requirement of the Degree of Masters of Science in Computer Science and Engineering.

Supervisor/Advisor	Signature	Date
Chairperson	Signature	Date
Internal Examiner	Signature	Date
External Examiner	Signature	Date

Acknowledgment

I am grateful to the almighty for blessing me, giving me good health and strength through the thesis journey. My sincere gratitude to my advisor Dr.Mesfin Abebe for his willingness to advise on the research, his encouragement, guidance, support, and critical comments improved this thesis work.

I would like to thank members of the Smart Software SIG of the CSE department, who sat on each phase of presentations for their input to the improvement of the thesis. I would like to thank my classmates for their support until the completion of the thesis.

Last but not the least; I would like to thank my family for their continuous support and blessing, thanks to all my friends who supported me during the completion of the thesis.

Table of Contents

List of Tables

List of Figures

List of Equation

List of Abbreviations

AI	Artificial Intelligence
CPU	Central Processing Unit
DT	Decision Tree
DP	Defect prediction
F1	F1-score
PITS	Project Issue tracking system
KNN	K-nearest Neighbor
LOC	Lines of Code
LR	Linear Regression
ML	Machine Learning
MDP	Metrics Data Program
ODC	Orthogonal Defect Classification
OVR	One-Versus-Rest
RF	Random Forest
SDP	Software Defect Prediction
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
WCSS	Within Cluster Sum of Square

Abstract

Software defects are errors that prevent a software product to function properly; they have a huge impact on software quality and reliability. Identifying and fixing defects early will result in better quality and is cost-effective. Thus, software defect managing approaches have gained much attention in the arena of software engineering. After defects are detected knowing their magnitude is very important, the severity level of defects shows the effect the software defect can cause to the end-user, knowing the stage of defect is now primary interest. Defects with the highest severity level should have to be maintained before defects with lower severity, it helps to assign more resources to the most defective part of the software. Various research works have been done in defect prediction, but it's also important to know the stage of defects. Some works suggested severity level prediction for defect reports from bug tracking software. But none of the studies addressed the severity level for the software module.

This study proposed Software Defect Severity level Prediction for software modules using NASA's MDP repository dataset that contains software defect metrics such as McCabe Metrics, Halstead metrics, and LOC Metrics which determines the defectiveness of module. The study used a clustering machine learning algorithm to label the datasets to appropriate classes of severity level. Then after it used RF, DT, KNN, and SVM machine learning classification algorithm to perform the classification task, the algorithms are applied to five datasets from the repository named CM1, KC1, KC3, PC, and PC3 to develop a model, the performance of each algorithm is measured and compared to select a better model. Based on the accuracy and f measure, the result shows that the RF model performed well on each dataset compared to the DT, KNN, and SVM models which resulted in maximum testing accuracy of 97.4% on PC1 and a minimum testing accuracy of 90.1% on KC1 .datasets also the f measure is 97% on PC1 and 90% on KC1. Generally, the result of the study shows software defect metrics can determine the severity level of a defective module and RF model can be recommended for software defect predictions.

Keywords: Software defect, severity level, software defect metrics, clustering, machine learning classifier, NASA MDP dataset.

CHAPTER ONE

1. Introduction

1.1. Background of the Study

Technology is rapidly growing in this modern era. The numbers of software produced are increasing daily as the number of software users are increasing. Multiple industries are using software technology in their day to day activities for solving big problems in their company. According to “Evans Data Corporation”, in 2019 26.4 million software developers were there, 27.7 million software developers are expected in 2023[1]. Along with the production of software, software defects are very much part of the process. Most software companies are facing difficulties in producing quality software. Some large software products are delivered without appropriate functionality. Software testing has been a hot topic of discussion; because the need for quality software by software users makes software testing a fast-growing area.

Even though the software is tested, the probability of defect existence in software cannot be rejected. It can be confronted while using the software product or under-test. Software defects that are detected can have an impact on software reliability, quality, and maintenance cost. Software failure can result in financial loss and can cost even our life; they can be caused by incomplete or wrong methods, unreliable environment, peoples without proper training, or incomplete requirements[2].

Since defects cannot be eliminated, testing and fixing defects will be difficult as the software becomes larger and complex. Managing software defects in the software development process will help the software developing companies in reducing a given type of defect by identifying the cause of the defect; preventing and fixing defects to improve the quality of software product also achieve user satisfaction by improving the performance of software product. Early management of defects is cost-effective and adds value to the reliability, availability, and maintainability of software artifacts[3].

Software defect detection is an important phase of software testing that helps to identify defective software modules[4]. The area of software defect detection involves using software

metrics as an input to characterize the given software modules. Software metrics are quantitative data that used to determine if a given software is defective or not, LOC, McCabe, Halstead metrics are the most used metrics in software defect detection [5]. Various researches have been done to address software defect problems[6][7] [8] by using machine learning techniques.

After defects are detected major activity in software defect management is classifying defects by identifying critical risk the defect can cause, that is categorizing the defect based on their severity level. Defect severity is the highest risk the defect can cause, shows the seriousness of software defect[9]. Defects are fixed based on their severity level, the impact they cause on the software product. The following are standard classification of defects based on their severity level.[9]

- Critical: That causes software damage, causes harm to end-user, and stops the software.
- Major: Major functionality dysfunction
- Minor: Change the behavior of the software but still works
- Trivial: it won't cause any harm or crash to the system

Research works have been done to solve software defect problems like predicting whether a software module has a defect or not. But after defects are detected it's important to know their severity level, severity levels help to give priority to the software module with the highest number of defects. This study focuses on addressing software defect problems by assessing their severity level and classifying them as critical, major, and minor. This research work proposed software defect severity level prediction for software modules using NASA's MDP repository dataset that contain software defect metrics such as McCabe Metrics, Halstead metrics, and LOC Metrics which determines the defectiveness of module and proposed multiple machine-learning algorithms to compare and select the best prediction model.

1.2. Motivation of the Study

The researcher, as a software engineer, observed some software quality issues that result in a high risk in human life and also caused big financial loss. Recently Boeing, a US-based

aircraft manufacturer launched new model max 737, in October 2018 the airplane crashed in Indonesia and the same model is crushed in Ethiopia in March 2019 due to software failure which result in the killing of 346 people in the accident, British airlines canceled more than 100 flights due to software crash in August 2019 which result in major financial loss[10]. Also, software testing is an evolving and interesting active research area from the software engineering point of view. Currently, most software companies are producing software product and delivering software with inappropriate functionality due to a lack of proper software testing. Software quality becomes a challenging issue for most software developers. If defects are detected and fixed in an early stage of software development, it will decrease maintenance cost, and quality software which is defect-free software can be delivered to the customer. Software quality should have to be a requirement for successful software development. So the researcher was inspired to study software defect severity prediction because addressing defect issues will improve software quality which is a critically important and sensitive issue currently.

1.3. Problem statement

Hidden defects can cause strong damage to the software product. Early detection and fixing of those defects is an important task in the software development process. To deal with software defect problems it's possible to conduct different types of research based on software defect metrics. The first one is predicting if a given software module is defective or not. The second one is predicting the magnitude of the defect that is the effect the defect has on the software such as severity level. After the defects are reported their severity level will be analyzed and will be maintained. Maintaining all defects at the same time is difficult, so knowing their severity level is important to give priority to the most defective parts of the software. It assists the software quality assurance team to give attention to defect-prone areas of the software, help to put more effort into the highly defective module, and effectively allocate resources for fixing the defect. Also improves the efficiency of the defect detection system which improves the quality of software produced.

Most studies on software defect prediction are done on the prediction of if the software is defective or not, this work helps to predict the magnitude of the defect which is severity level

by using machine learning algorithms. Some works have been done for bug reports from bug tracking software to predict the severity of software bugs using text classification techniques, these bugs are only reported from bug tracking software, but what if the defect is per software module? After software modules are tested and detected with defects, severity level assessment for defective software modules is also needed. This study predicts severity levels for software modules by addressing the problem using a dataset with defect metrics that determine the defectiveness of a software module and multiple machine learning techniques. The study addresses the problem by answering the following questions:

- How to balance biased classes of the existing dataset?
- How to prepare the data set for software defect severity prediction using the existing data sets which do not have severity level?
- Which machine learning algorithm has better performance for severity prediction?

1.4. Objectives of the Study

1.4.1. *General Objective*

The general objective of this work is to predict the severity of software defects using machine learning's clustering and classification techniques.

1.4.2. *Specific Objective*

The specific objectives are:

- To review the literature on software defects predictions and machine learning's application in software engineering
- To collect data from open source data repositories
- Preprocessing data set for effective noise removing and resampling the dataset for tackling the class imbalance
- To cluster the dataset to appropriate classes for the prediction
- To develop models by using a different machine learning algorithm
- To evaluate the performance of the model.

1.5. Scope and Limitations of the Study

1.5.1. Scope of the Study

The study focused on predicting the magnitude of software defect which is severity by using data from NASA‘s MDP which has McCabe Metrics, Halstead Metrics, and LOC Metrics. The dataset with defective modules is clustered into critical, major, and minor. The model is trained and tested based on those classes, also the performance of the model is measured with Accuracy and precision, recall, and confusion matrix.

1.5.2. Limitations of the Study

As the time is given not enough this work is only limited to report how severe the defect is, after the severity level of the defect known the next action will be giving priority to and fixing them, but it’s not going to give priority to defects of each module that has a high severity level. Another method is required to deal with those issues, but the time given is not enough to add another method.

1.6. Application of the Study

The main beneficiaries of this study are software developers mainly quality assurance teams can identify the most defective parts of the software and can put more effort into defective modules to allocate limited resources for fixing software products. They can also easily analyze defects that have a high risk on the software, can track all defects with their severity level. Software testers can identify which software module cannot function correctly and the degree of an effect they have on each software module by using defect metrics.

1.7. Organization of the Thesis

The research paper organized under seven chapters in the following ways: *Chapter one* discussed above, and includes the introduction of the study, the motivation, and the statement of problems, the research questions that would be answered in the proposed solutions, the scope and limitation, the objective of the study, the methodology, and the application results of the study.

Chapter two: presents the literature review and related works on Software defect severity prediction, the definition of Defect, Categories of a defect, Software metrics used in the study, Machine learning used in software defect prediction, and finally, it discussed the related work.

Chapter three, discusses the research methodology used in this research work, methods used to develop Software defect severity prediction models which are preprocessing, resampling, clustering, machine learning classification algorithms, and finally, it presents the evaluation methods.

Chapter four discusses the proposed solution for software defect severity prediction which is the proposed preprocessing, machine learning training, and testing.

Chapter five discusses the implementation and experimentation of the proposed solution including working environment, dataset description, also the implementation of preprocessing, clustering, models implementations, and evaluation models.

Chapter six discusses the result of sampling, clustering, models evaluation result *Chapter seven* presents conclusion, recommendation and identifies future works for further research direction on this research topic

CHAPTER TWO

2. Literature review

This chapter reviews relevant literature to further understand and investigate the research problem. Provide technical review by defining software defects, followed by a review of the cause of software defects, defect categories, followed by explaining defect severity, software metrics, machine learning techniques that have been used, and related works studied on the problem.

2.1. Software Defect

Most of the time researchers cannot differentiate between software defects, bugs, faults, failures, wrong, errors in software testing. This study reviewed the definition of software defects given by different sources. The following are some of the definitions of software defect and their sources:

- ***Definition 1:IEEE Guide to Classification for Software Anomalies,"*** A defect is an imperfection or deficiency in a work product where that work product does not meet its requirement or specifications and needs to be either repaired or replaced"[9].
- ***Definition 2: The Practical Guide to Defect Prevention,(Marc Mcdonald, Robert Masson & Ross Smith, 2007),"*** a deficiency in a software product that causes it to perform unexpectedly"[11].
- ***Definition 3: Software Testing and Quality Assurance,(K. Naik &P. Tripathy, 2008),"*** A software defect is a fault, error or failure in a software"[12].
- ***Definition 4: Foundation of Software Testing,(E.Van, Isabel. E)*****,"** if someone makes an error or mistake in using the software, this may lead directly to a problem-the software is used incorrectly and so does not behaves as we expected. This is called defect"[13].
- ***Definition 5:Software Defect Classification Based on Mining Software Repository, Hui Wang 2014***," Software defects are programming errors which cause the different behavior compared with the expectation"[14].

- ***Definition 6: ISTQB***,“ a flaw in a component o system that can cause the component or system to fail to perform its required function”[15].

All the above definitions can be summarized to define software defects for this research work. **Software defect:** Prevent software product to function in the desired way or expected way, error in code, requirement, or design that makes software inefficient to meet user requirement.

Some of the concepts related to software defects are described in Table 2.1

Table 2.1 Types of defects [9]

Types of defects	***Description***
Bugs	Error in codes
Wrong	When requirements implemented in the wrong way
Error	An error occurred by the misunderstanding of the system by software developers
Fault	Incorrect process in the software program
Failure	The Inability of software to perform the required functionality

Many reasons cause defects in software development. While designing, building, or coding the software, developers can make a mistake that causes the software to be defective. Defects can be caused by [16];

***Deficiency in requirement*;** inaccurate and unsatisfactory requirement.

***Coding error*;** errors caused by programmers while codding the system.

***Lack of logic*;** poor design by software developers.

Inadequate Software testing; Improper testing by software testers before releasing the software to the customer.

Human error**;** incorrect input by the user to the system

2.1.1. Software Defects Classification

After defects are detected, defect classification is an important task. Several defect classification tools have been used in different studies. Orthogonal Defect Classification (ODC) which is proposed by IBM companies is the most used defect classification technique[17]. Defects are classified based on types of defect, sources, impact, trigger, severity, and phase found. The other one is which is proposed by HP, classifies defects based on origins, types, and modes[18]. Other approaches also classified defects based on maintainability, standards, user interface. Defects can be categorized based on their severity level. According to IEEE "Classification for Software Anomalies", Defect severity shows the degree of impact the defect can cause to the software product[9]. The severity level of defect gives insight into the risk the defect has on the end-user; it shows the quality of the defect under test. They play important role in the release decision of software products[19].

Defects can be classified based on their severity level into the following[3]

- ***Critical*****:** Defect that causes immediate damage to the system, complete crush software that risk end-users safety and result in loss of main components of the system. This type of defect should have to be fixed quickly.
- ***Major*****:** its high severe defect resulted when major functionality of the software is not working, but some parts of the software are still functional
- ***Minor*****:** Which does not affect the main functionality of the software, cause unwanted behavior, but has a low impact on the implementation of the software.
- ***Trivial*****:** it won't cause any harm or crash to the system it's just a "cosmetic" problem, does not need fixation.

2.2. Software Metrics

IEEE Standard of Software Quality Metrics Methodology, defined metrics as " software metrics is a function, with input as the software data, and output is a value which could decide on how the given attribute affect the software"[20].

Software Metrics are characteristics used in measuring the performance of software products, they describe specific features of software[21]. In software development, it's important to define metrics as they are used to measure the quality of the software, estimating the cost, planning work items, identifying defects in code, predicting defects and risks in project management [22][5].

Software defect prediction metrics have the most significant role to build a prediction model that aims to improve software quality by predicting as many software defects as possible.

Most defect prediction models used various software metrics.

Lines of code were the first metrics used in early studie**s.** Lines of Code is size metrics used to measure the number of lines in a given code includes classes and methods[23]**.** LOC can be used to check the size of code units; methods cannot be easily understandable if there are more than 20 lines of code in the method. It can be also used to estimate the size of the project. With larger LOC values it takes more effort and will be difficult to maintain, code duplication and larger LOC values are not recommendable within a single module as it results in bugs.[24].Many studies proved that LOC can be useful in determining software quality, it's one of the simplest metrics in predicting software defect[25].

The other metrics introduced in the early 1970s is complexity metrics which is **McCabe Complexity Metrics** proposed by McCabe. IEEE Transactions on Software Engineering, Dec 1976, Thomas McCabe Defined " a set of metrics to characterize the complexity of a software modules internal control flow logic, it's a quantitative measure of independent paths in source code of the program [26]. This metrics application was to limit the complexity of the software module during programming. Whenever the complexity limit exceeds the modules should have to be split into submodules to reduce complexity as highly complex modules will result in a defect. Cyclomatic complexity is one of the McCabe metrics that used to define the

complexity of a single module by measuring the extent of decision logic of software modules. It quantity independent paths in the software program, Cyclomatic complexity within a software program can be calculated by using control flow graphs. Figure 2.1 shows a flow diagram for normal sequence, while, if-else, and until statements.

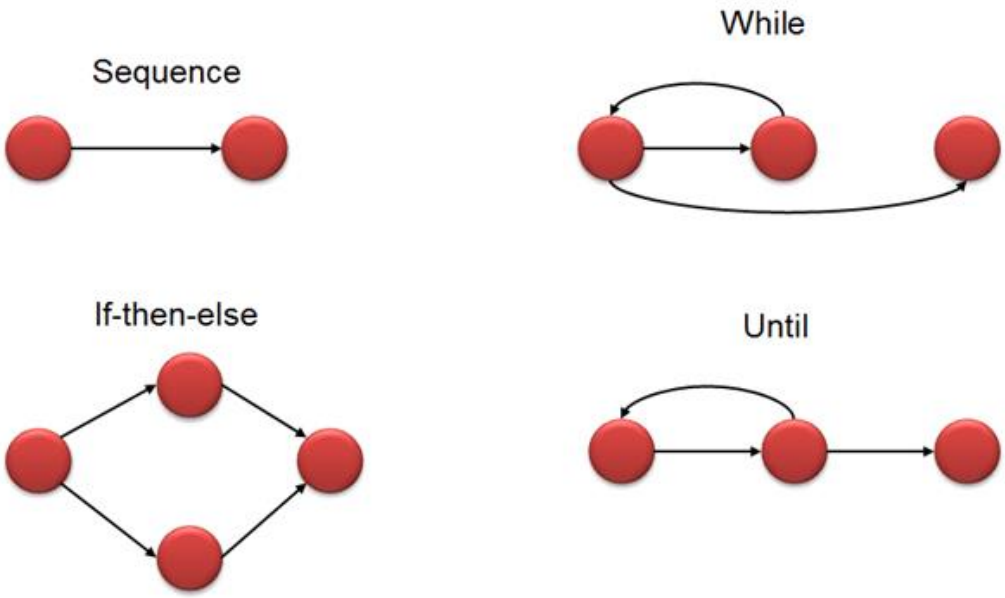


Figure 2.1 flow graph notation for a program[27]

Mathematically, cyclomatic complexity is calculated by the equation;

$$\mathbf{V(G)} = \boldsymbol{E - N + P} \qquad 2.1$$

Where,

V(G)= cyclomatic number

E= number of edges

N= number of nodes

P= number of connected components

Properties of cyclomatic complexity

- V(g) is the maximum number of independent paths in the program
- V(g)>=1
- V(g)=1 shows G will only have one independent path
- Complexity should have to be minimized to 10

The other metrics are Essential complexity which measures unstructured constructs in a code shows the structuredness of a code [28]. The other is design complexity measure the interaction between modules in the software. To build a software defect prediction model, previous studies have been using McCabe complexity metrics[29].

Another software metrics related to complexity is **Halstead metrics** which is proposed by Halstead[30]. Shows how difficult to understand a given program, using redundant operators and operands will result in difficulty in writing and understanding the program. Halstead metrics focus on the counting of tokens and determines the number of operators and operands, total number of occurrences of operators, total number of occurrences of operands in a program, also shows program time, volume, length, effort, and difficulty [30].

By counting tokens and identifying which are operators and operands basic measures can be composed as;

n1 = the number of unique operators

n2= the number of unique operands

N1= the total number of operators

N2= the total number of operands

From these numbers different Halstead metrics can be calculated;

- Halstead program length is calculated by the equation;

$$\mathbf{N = N1 + N2} \qquad 2.2$$

- Halstead vocabulary is calculated by the equation;

$$\mathbf{n} = \mathbf{n1} + \mathbf{n2} \quad (2.3)$$

- Halstead volume is calculated by the equation;

$$\mathbf{V} = \mathbf{N} * \mathbf{log2n} \quad (2.4)$$

- Halstead difficulty is calculated by the equation;

$$\boldsymbol{D} = \frac{\boldsymbol{n1}}{\mathbf{2}} * \frac{\boldsymbol{N2}}{\boldsymbol{n2}} \quad (2.5)$$

- Halstead effort is calculated by the equation;

$$\mathbf{E} = \mathbf{D} * \mathbf{V} \quad (2.6)$$

2.3. Machine Learning

Machine learning is one of the fast-growing sub-area of Artificial Intelligence (AI) where computer systems automatically find a solution by recognizing patterns from a learning experience**. Machine Learning, Tom Mitchell 2011**, described " Machine learning is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"[31]. Machine learning observes data from direct experience, instructions, or examples to make the decision by looking at patterns from observed experience. Machine learning aims to make the computer system learn automatically from a given large amount of data without human intervention, and assist by giving the desired output.[32]. Machine learning is used in finding relevant data, predicting patterns based on analyzed data, computing probabilities, recognizing patterns, adapting certain development.[33]. Machine learning algorithms accept input data, analyze the data to predict new output values based on learned examples. Machine learning techniques can be classified into Supervised, unsupervised, semi-supervised based on the classifier.

***Supervised learning*:** The algorithm use labeled data as an input to predict the output based on learned experience. Supervised learning is commonly used in classification problems as the

algorithm maps input to the desired output[34]. This type of learning, model the correlation between the target variable and the feature used as an input such that they can assist in predicting new values based on the relation between the values learned from the dataset[35]. After sufficient training, the learning algorithm gives targets for new input, and the performance of the algorithm is evaluated by comparing the output using test data. Supervised learning techniques classified to classification; used to predict discrete values and regression; which are used to predict continuous value.

***Unsupervised learning*:** Unsupervised ML techniques used to give an insight into unlabeled data, used when we train data that has no label or classification. Unsupervised techniques describe and infer the hidden structure of the data from unlabeled data. The aim is to have a learning system to analyze data and draw inference without telling it how to do describe the structure[36]. They can be used as an initial step before using supervised learning. The hidden structure of the data can give an inference on how to better predict the output[37]. We have clustering and dimensionality reduction in unsupervised learning.

***Semi-supervised learning*:** it's a mixture of supervised and unsupervised learning. Used when your training data have few labels when there is no enough resource to label the dataset. They use classification for identifying the data and clustering for grouping the data into different groups, the semi-supervised model can increase the size of the unlabeled data[38].

2.3.1. Machine Learning in Software Engineering

Machine learning appears in many disciplines as its intersection between computer science, engineering, and statics. Machine learning can be used to solve many problems, any field can use machine learning to interpret and work on data[38]. Software engineers can use machines that assist them in software development. Machine learning-based solution can be developed for software engineering problems.ML has been applied to predict, estimate, classify, generate software engineering processes[39].

Table 2.2Machine Learning Application in predicting and estimating SE artifacts

Software engineering tasks	Machine learning Applications
Software defect prediction	Identifying defect prone software modules, predicting and classifying their magnitude such as severity, priority, and probability[40].
Software quality prediction	ML models have been used for predicting internal quality attributes of software [41]
Software reliability prediction	ML models are used in estimating and predicting system failing in given time spam[42].
Software cost and effort estimation	ML models have been used to predict the resource and effort required to develop a software system such as budget and schedule[43].
Software Size Estimation	Estimating size of software component using LOC metrics[44].
Software reusability prediction	Identifying the probability of reusing existing software components[45].

2.3.2. Machine Learning in Software Defect Prediction

Several Machine Learning algorithms were used by researchers to identify defect prone software modules and also for categorizing the amount of the defect. Most of the researches that are reviewed used supervised Machine learning algorithms such as; Decision tree classification, Support vector machine, K-nearest Neighbor, Naïve Bayes, Random forest, neural network, Linear Regression, Logistic Regression. Shruthi P, Pranav D. et al[46]. Applied multiple linear regression model to predict numbers of defects on a single software

module. They used historical data from Bugzilla, CVS projects, to determine the number of defects in the future. A.Hammouri, M..Hammad et al.[8] Applied Decision Tree, Naïve Bayes, Neural network classification algorithm software defects based on historical data. They used three different datasets from different software projects to detect defects. Sushant K, Ravi.B.Mishra et al.[47].Applied Bayesian Network classifier to classify defects, using different Open -source datasets. Yan N Soe, Paulus Santosa et al. [48]Applied Random forest algorithm on PROMISE public dataset A . Alsaeedi, M. Zubair Khan et al. [49]Applied SVM, RF and decision tree on 10 NASA defect dataset to predict defects.

Table 2.3 frequently used machine learning algorithm

Algorithm	***Pros***	***Cons***
RF	Reduces overfitting and improves accuracy, work in high dimensional data, automates missing values in the dataset.	Complexity happen because of RF creates a lot of trees, requires more time for modeling.
DT	Easier to understand, requires less code, does not requires scaling, model non-linear decision boundaries.	Higher time for modeling, a small change in the data can cause a huge change in the structure of the DT, Easy to overfitting, instability.
SVM	Effective with high dimensional spaces, handle non-linear, imbalanced data, and result in high accuracy.	Does not perform well when there are noise and overlapping data, not suitable for large data.
KNN	No assumption of data, easy to implement, no training data period, high accuracy.	Does not work well with high dimensional data, needs feature scaling, sensitive to imbalanced data set, poor interpretability.
Naïve	Easy to implement, the training	The assumption of independent

Bayes	period is less, work with small data.	predictors, assumes all attributes are mutually independent.
Linear Regression	Easier to implement and interpret the output, overfitting can be reduced by regularization.	Sensitive to outliers, assumes a linear dependency between dependent and independent variables, prone to outliers and overfitting.

2.4. Related works

Tim Menzies et al.[50] Introduced the first automated method SEVERIS that assign severity levels of software bugs. This method is applied to NASA's Project and Issue Tracking System(PITS), to extract relevant features they used text mining techniques. They classified severity levels of bugs into Severity 1, severity 2 severity 3, severity 4.

K. K. Chaturvedi et al. [51] Assign severity level from 1 to 5, 1 is assigned for the most severe bug report and 5 is assigned to the lowest severe to bug reports from NASA's PITS dataset. They studied different machine learning algorithms to demonstrate the application of machine learning in determining the severity level of software bugs. They applied Support Vector Machine, K-nearest Neighbor, Naïve Bayes, and J48 to the dataset. The algorithms are evaluated using different measures and the accuracy was obtained was between 56.11% and 79.21%.

R. Malhotra, N. Kapoor et al.[52] Assessed the severity of defects and categorized defect reports from NASA's PITS dataset into NEW, EXAMINED, CONFIRMED, RESOLVED, VERIFIED, and ACCEPTED, used text classification techniques and, have applied Support Vector Machine of machine learning to classify documents on the basis of severity levels. Accuracy measure was between 72 to 75% accuracy.

R.Jindal et al.[53] They used bug reports from PITS A, PITS C, PITS D, and PITS E datasets from NASA's Software Verification and Validation (IV & V) Program and classify severity into 1 to 5. Range from low severe to high severe, Multi-nominal Multivariate Logistic

Regression, Multi-layer perception, and Decision tree machine learning algorithm are used and, the accuracy measure lies between 0.83 and 0.88.

G. Sharma et al.[54] They presented an approach for assessing bug severity; they classified bug reports from Eclipse and Bugzilla bug repository by creating a dictionary of critical terms. The report has defined severity level as a blocker, critical, Enhancement, major, minor, normal, trivial. They classified critical and blocker as severe and minor and trivial as non – severe. They used Naïve Bayes Multinomial and K-nearest neighbor algorithms for the classification of bug reports. The accuracy of the algorithm obtained in this study lies between 69 % to 75 %.

A. Baarah et al.[55] They build a prediction model for the severity level using software bug reports in a closed source project. They used a dataset from a local company located in Jordan using the JIRA bug tracking system that is related to closed-source projects, that have severity levels lowest, low, medium, high and highest, they classified lowest and low as non-severe, highest, and high to high severe. They applied Support Vector Machine, Decision tree, Naïve Bayes, Naïve Bayes Multinomial, Random Forest, K-nearest neighbor, and Decision rules machine learning algorithm. They evaluated and compared each algorithm the best accuracy was obtained 86.31%.

Table 2.4 Summary of Software defect severity prediction related works

Author	***Title***	***Dataset***	***ML method and Accuracy***
K. K. Chaturvedi et.al [51]	"Determining Bug Severity Using Machine Learning Techniques"	Bug reports from NASA's PITS datasets.	Support Vector Machine, Naïve Bayes, k-Nearest Neighbor, J48. Between 56.11% and 79.21%.
R. Malhotra, N.	"Severity Assessment of	Bug reports from	Support Vector

Kapoor et.al [52]	Software Defect Reports using Text Classification"	NASA's PITS dataset.	Machine between 72 to 75% accuracy.
R.Jindal, Abha Jain et. al[53]	"Prediction of Defect Severity by Mining Software Project Reports"	Bug reports from NASA's PITS dataset	Logistic Regression, Multilayer Perceptron, and Decision Tree between 0.83 and 0.88.
G. Sharma , S Sharmaa et al[54]	" A Novel Way of Assessing Software Bug Severity Using Dictionary of Critical Terms"	Bug reports from Eclipse and Bugzilla	Naïve Bayes Multinomial and K nearest neighbor between 69 % to 75 %.
A Baarah et.al [55]	" Machine Learning Approaches for Predicting the Severity Level of Software Bug Reports in Closed Source Projects"	Software Bug Reports in Closed Source Projects.	Naïve Bayes, Support Vector Machine, Decision Tree, Random Forest, Decision Rules, K-Nearest Neighbor, and Naïve Bayes Multinomial The Accuracy obtained is 86.31%.

All the above mentioned works assign severity level for bug reports. Works such as Tim Menzies et al.[50], K. K. Chaturvedi et al. [51], and R.Jindal et al.[53] classifies software bug reports from NASA PITS dataset into severity level 1 to 5, R. Malhotra, N. Kapoor et al.[52] classifies bug reports as NEW, EXAMINED, CONFIRMED, RESOLVED, VERIFIED, and ACCEPTED. Works by G. Sharma et al.[54], and A. Baarah et al.[55] classifies bug reports from Bugzilla, Mozilla, and JIRA bug tracking software to severe and non-severe. Thus none of the above mentioned research works are done on software module defects. The proposed solution will address the prediction of software defects severity level of software modules, based on standard classification of software defects that are critical major, minor and trivial.

CHAPTER THREE

3. Research Methodology

This chapter describes the methodology used in conducting the study. In this section, we discuss the detail of the dataset used, model selection, and evaluation methods for the algorithms to achieve the objective of the study.

3.1. Data Source Description

Software defect datasets from NASA's MDP repository which are available online are used[56]. MDP repository contains data of 13 projects, those datasets contains software defect prediction metrics such as;

Line of Code Metrics (loc_blank, loc and comment, loc comments, loc executable, number of lines, loc total),

McCabe Metrics (cyclomatic complexity, design complexity, essential complexity, cyclomatic density, essential density, normalized cyclomatic complexity, essential complexity),

Halstead Metrics (num_ of _operands, num_operators, num_unique_operands,halstead content, halstead difficulty, halstead effort, halstead_error_est, halstead length, halstead level, halstead _prog_time, halstead volume).

Others (node_count, edge count, branch count, call pairs, condition_count, decision count, parameter count, global_data_complexity, global_data_density, modified_condition_count, multiple_condition_count).

The dataset has the dependent attribute "defective" which contains 'Y" or "N" values. "Y" means that a given module is defective and "N" describes that a certain module is non-defective. Details of the datasets are described in Table 3.1.

Table 3.1 NASA's MDP repository dataset overview

Data sets	***Programming***	***No of modules***	***Defective modules%***
CM1	written in C	671	10%
JM1	written in C	10875	19%
KC1	written in C++	3257	15%
KC3	written in java	394	9%
KC4	written in Perl	125	49%
MC1	written in C and C++	9466	0.7%
MC2	written in C	161	32%
MW1	written in C	403	8%
PC1	written in C	1414	7%
PC2	written in C	5589	0.4%
PC3	written in C	2152	10%
PC4	written in C	1458	12%
PC5	written in C++,	17186	3%

3.2. Software Defect Severity Level Prediction Modeling

3.2.1. Preprocessing

Data preprocessing is an essential task for making data appropriate for a machine learning model which also increases the performance of a machine learning model. Preprocessing

methods have been used for cleaning the dataset, handling missing values, balancing the biased classes, and clustering the dataset to the appropriate group of class labels for prediction.

3.2.1.1. Dataset Balancing Methods

Resampling approaches are mainly used in balancing biased classes for classification. Class imbalance problems happen when the number of values in one class is much smaller than the number of values in the other class. Prediction models with imbalanced data are biased towards the majority class, therefore the model will produce inaccurate results[57]. The nature of the dataset used in this study is highly imbalanced between the instances of the defective and non-defective instances. As described in Table 2 percentage of the defective modules for each dataset is less than the non-defective modules. The data has a representation of the majority class as non-defective while the minority classes are represented by defective classes. In this study combination of under-sampling and oversampling methods are used in balancing defective classes with non-defective classes of the dataset

***Under-sampling*:** is one of the most used techniques for imbalanced datasets. In this method, the algorithm attempts to balance by randomly removing the majority class instances from the datasets. Consequently, the majority class of examples will be reduced from the dataset. This method will be repeated until the number of instances in each class becomes equal[58][59].

***Oversampling*:** this technique randomly duplicates minority classes to balance the dataset. It selects random instances of minority class with replacement and providing the training data with multiple duplicates of the minority class instances. Oversampling is a non-heuristic method that randomly replicates examples in the minority class

This study uses the combination of both method which result in better performance, the concept is applying some amount of oversampling to the minority class to improve the bias of minority class, applying some ration of under-sampling to reduce bias in the majority class

3.2.2. Machine Learning Clustering Algorithm

The data sets used are not labeled with severity information, for predicting severity levels, the model requires labeled data to train, that is the data should have a target value that describes

the severity level of a software module. From the dataset modules with only defective classes are needed, and then the defective classes should have to be labeled with severity information. For labeling defect dataset, clustering methods are used to group the defectcive software modules based on the values of software metrics. The Clustering algorithm can be used to organize data points that have similar features into the same group, instances within the same clusters are similar and those found on the other clusters are different, the algorithm will process the data and find the groups of the data point that exist in the data, after applying the algorithm it gives insight which group the data point will be categorized by analyzing our data [60][61].

K-means clustering algorithm has been used in this study to cluster the data set to appropriate classes. It's the most used clustering algorithm because it's easy for implementing, interpreting, and simply presenting the result. K-means algorithm categorizes the dataset into k clusters, where k values are predefined. It classifies each data point into only one group, where data points that belong to the same group are similar. For each cluster a single point called centroids will be defined, these points of centroids should have to be placed carefully, it will give better result if they are placed far away from each other. The following step will be associating each data point to the nearest centroid. An early grouping will be done in this stage when there is no point pending, k new centroids will be re-calculated as centers of the clusters resulting from the prior step. After we get k new centroids, a new association will be done between data points and new k centroids. An Iterative loop will be generated; as a result, those centroids will change their place until no changes are done, until they stop moving anymore[62][63].

3.2.3. Machine Learning Classification Algorithm

The objective of the study is to predict the severity level of defective software modules using machine learning algorithms. For achieving the objective multiple supervised machine learning algorithm are used and their performance has been compared. The following machine learning algorithms are used to build the prediction model. The algorithms are chosen based on their good performances and popularity in software defect prediction problems on previous work results[64][65][40].

Decision Tree: Decision trees use a branching structure to map the output of the decision. The tree has three main components such as the decision node, branches, and leaves. The features of the dataset represented by an internal node, decision rules are represented by and outcomes are represented by leaves. For predicting a given class of the datasets, by starting from the root node the algorithm compares the node with actual features of the dataset it then follows a branch based on the comparison. The algorithm will again compare the attributes with the sub-nodes until it reaches the leave nodes[66][38].

Random Forest: is a supervised machine learning that creates forests by building and merging multiple decision trees. It uses bagging when building the tree. While splitting the tree instead of searching for the most important features, it finds the best feature from a random subset of features, it finds the best feature from a random subset of features. In RF it's easy to measure the importance of each feature in the prediction, it is fast and can handle a large data number of attributes in the datasets[34][38].

***Support Vector Machine*:** Support vector machine is widely used in classification problems. The algorithm classifies the data by constructing n-dimensional decision boundaries called hyperplanes which separate the data points into two categories. To create a hyper-plane the algorithm chooses extreme data points called support vectors. The main aim of the algorithm is to find the hyper-plane that helps to separate those vectors in such a way one group of the target variable is categorized on one side plane and the other side will have the second target group[37][66]. In this study, we used multiclass implementation of SVM which is called one-versus-the-rest to classify our dataset, because SVM is a binary classifier and the dataset we are using has multiple labels. In the one-versus-the-rest method, the number of class's c will be constructed as binary SVM classifier, the classifier will reduce to a two-class problem by differentiating one class from the other[67].

***K-nearest Neighbor*:** To classify new instances KNN algorithm assigns data points to the nearest neighbors, which are the most labeled instances. KNN algorithm is non-parametric, does not make an assumption on data and it's also a lazy learner algorithm, it acts on the data set at the time of classification by storing the dataset, instead of learning from the training set. The algorithm works by allocating the number of k-neighbors and then computes the Euclidean distance between the k-neighbors. Based on the result of the distance the k nearest

neighbor will be taken. From these neighbors, the number of the data point in each category will be counted and the new instance will be assigned where the category has the maximum number of k neighbors[68][38].

3.3. Model Evaluation

Model evaluation is an important aspect that helps us to choose the best algorithm for our data. It helps us to know how well our algorithm will perform in the future. Different types of evaluation metrics such as; accuracy, precision, recall, f-measure, and confusion matrix are used in this study to evaluate the performance of the algorithms. Those metrics are described by the following terms;

- True Positive (TP): when the actual observation is True but, is predicted to be True.
- False Negative (FN): when the actual observation is positive but, is predicted negative.
- True Negative (TN): when the observation is negative but, is predicted to be negative.
- False Positive (FP): when the observation is negative but, is predicted positive

3.3.1. Accuracy

Accuracy describes the value of correct predictions of the classification divided by all predictions[69]. It's given by the equation:

$$\textbf{Accuracy} = \frac{\boldsymbol{TP + TN}}{\boldsymbol{TP + TN + FP + FN}} \quad .3.1$$

3.3.2. Precision

Precision answers how many are actually correct from which are classified as relevant.it shows how many times the model predicts true[69]. It's calculated by the equation:

$$\textbf{Precision} = \frac{\boldsymbol{TP}}{\boldsymbol{TP + FP}} \quad 3.2$$

3.3.3. Recall

Recall shows the number of proportions that are truly relevant[69]. It's calculated by the equation:

$$\textbf{Recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FN}} \tag{3.3}$$

3.3.4. F-Measure

F-measure defined as the harmonic mean of precision and recall: is calculated by the equation:

$$\textbf{F} - \textbf{measure} = 2 * \frac{(\boldsymbol{Recall} * \boldsymbol{Precisoin})}{(\boldsymbol{Recall} + \boldsymbol{Precision})} \tag{3.4}$$

3.3.5. Confusion Matrix

A confusion matrix is a table that shows the prediction results of a classification model. It summarizes the value of correct and incorrect prediction by comparing the actual values with the predicted values by the classifier. It also shows the errors, types of errors that are being made, and the way the classifier is confused during prediction[70].In this study, a confusion matrix is used for four class models(Critical, Major, Minor, N) of severity levels, the following table shows a sample format of the confusion matrix for those classes.

Table 3.2 Confusion matrix: an overview

		Predicted			
		Critical	Major	Minor	N
Actual	Critical	True Critical	False Major	False Trivial	False N
	Major	False Critical	True Major	False Trivial	False N

	Minor	False Critical	False Major	True trivial	False N
	N	False Critical	False Major	False Trivial	True N

CHAPTER FOUR

4. Proposed Approach for Software Defect Severity Prediction

In this chapter, the proposed solution of Software Defect Severity level prediction using machine learning techniques that are suitable for problems stated under this study is discussed.

4.1. Proposed Software Defect Severity Level Prediction Framework

The proposed architecture is used to predict severity levels of software defect into Critical, Major, and Trivial. Based on the frame shown in Figure 4.1, the proposed architecture takes as an input the datasets from NASA's MDP repository which has software defect metrics that determine the defectiveness of a software model and then preprocessed the datasets by cleaning the dataset and removing irrelevant records from the dataset. After that, a combination of the under-sampling and oversampling method is used to balance the defective and non-defective classes of the dataset. After balancing the dataset the defective classes of the dataset have been clustered using k-means clustering machine learning algorithm to group the defective classes of the dataset into severity level (Critical, Major, and Minor). The resulted group of the severity level of defective modules combined with the rest of the non-defective module which contains 'N' class labels means no defect. After we have the training set ready with the desired class labels, Models developed using RF, SVM, DT, and KNN Machine learning algorithms. The best model is selected by evaluating the models using different evaluation metrics discussed in the previous chapter. Finally, new input with software defect metrics will be classified as Critical, Major, Minor, and N (no defect) by the selected model.

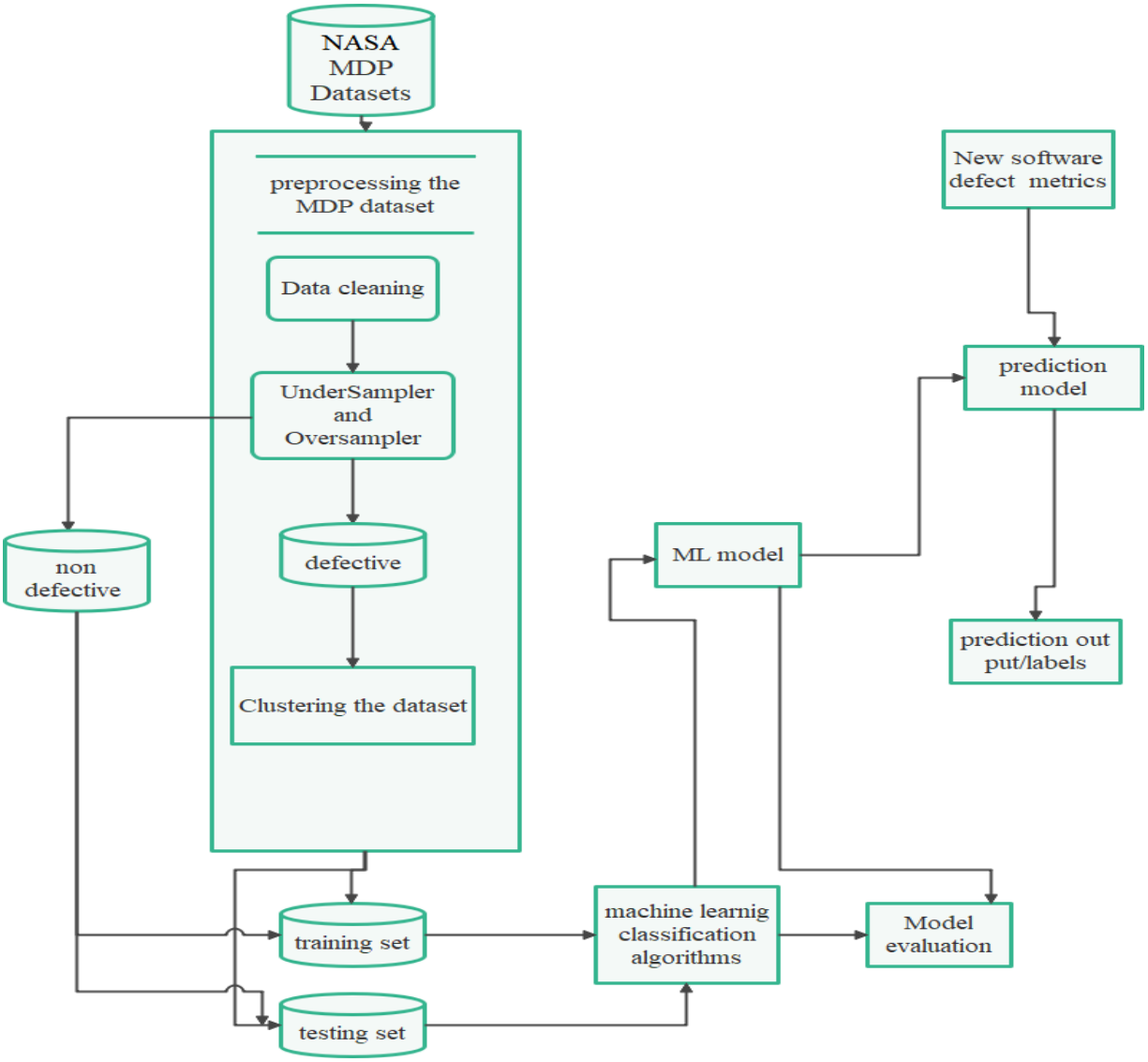


Figure 4.1 Software Defect Severity Prediction Framework

4.2. Proposed software Defect Severity Prediction Preprocessing

The proposed Software Defect Severity level prediction preprocessing component performs cleaning of the dataset by removing irrelevant records from the dataset and handling missing values and then sampling techniques are used for the imbalanced nature of the dataset. Defective and non-defective modules of the datasets are balanced by using a combination of under-sampling and oversampling techniques. After this, the defective modules are clustered using the k-means clustering algorithm and grouped to appropriate class labels for prediction.

And then the clustered defective modules datasets are merged with the non-defective modules datasets.

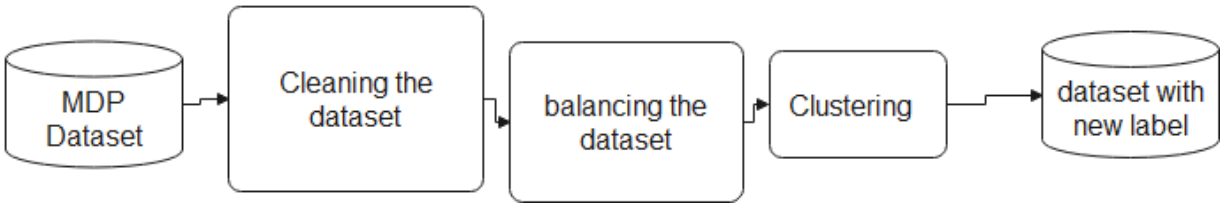


Figure 4.2 Dataset preprocessing

4.2.1. Cleaning the Dataset

The data set may contain some missing values with nan, irrelevant records that may affect the prediction model. This cleaning removes and cleans all those mentioned records

---

Pseudocode: 4.1 cleaning the dataset

Input: Records in the dataset

Output: Cleaned dataset

Begin:

1. Read records in the dataset
2. While (!end of records in dataset)
   If the data contains nan(not a number) value then
   Drop the record with nan.
   If the data contains null values
   Drop the record with null value
3. Return clean dataset

4.2.2. Proposed Dataset Balancing Method

A combination of under-sampling and oversampling methods are proposed in this study to balance the defective and non-defective classes of the dataset. This method randomly reduces non-defective 'N' (Majority class) from the dataset reduces the class to balance with the

defective ‘Y’ (Minority class). also randomly choose defective classes ‘Y’(minority class) and duplicates to balance with the non-defective ‘N’ classes which are majority class. This method is used to improve performance by reducing bias towards both classes. As shown in Figure 4.3 the algorithm takes the number of defective and non-defective classes and randomly reduces non-defective classes which are majority classes and increase the number of minority class which is defective classes to balance the classes.

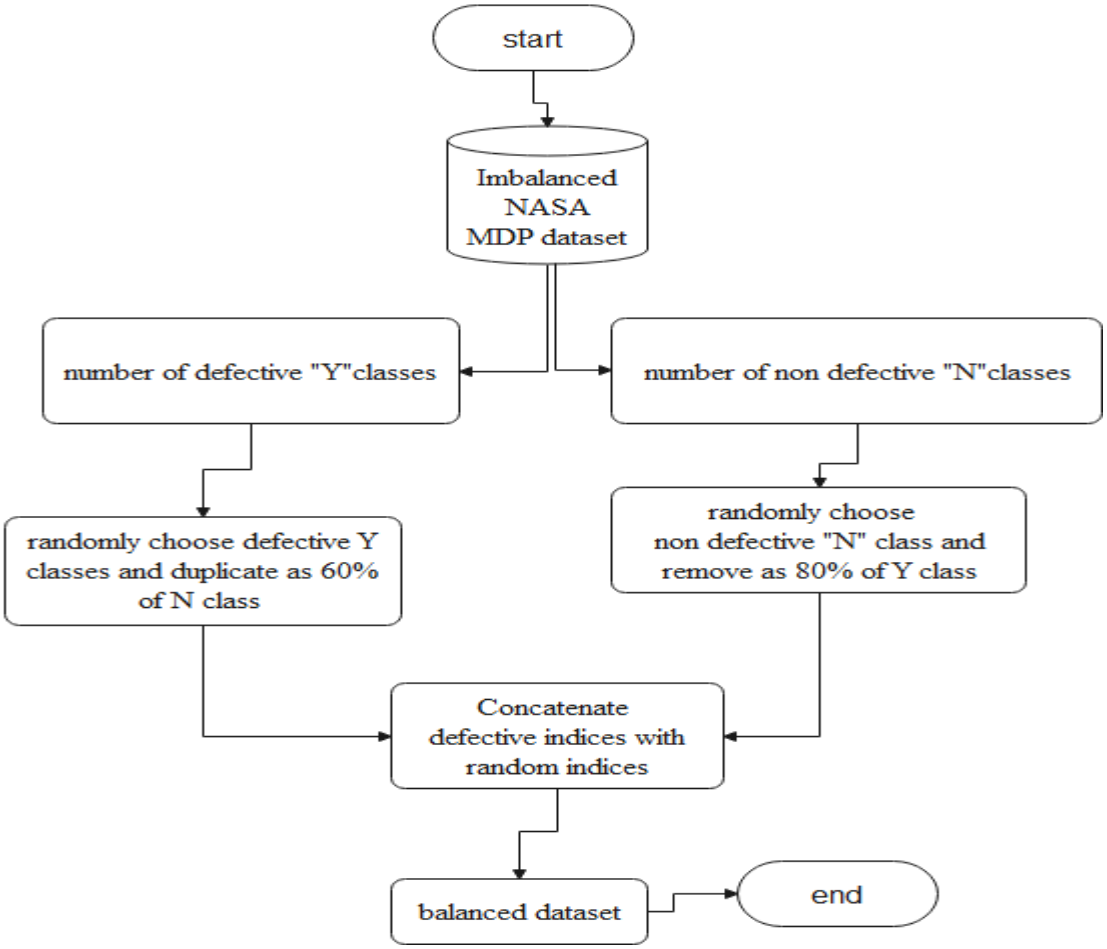


Figure 4.3 flow diagram for dataset balancing

4.2.3. Proposed Machine learning Clustering Method

The study proposed the k-means clustering algorithm to obtain appropriate labels of severity for prediction. The algorithm divides the data into k groups of clusters. It groups items into clusters by analyzing the similarity between the items. As shown in Figure 4.4. In this study after the dataset is balanced, a dataset with only defective classes is clustered into severity

levels. The algorithm works first by identifying the number of k clusters. In this study, elbow methods are used for finding a number of clusters. Then select the centroids and calculate the distance between the instance and the centroid. The data points are grouped based on the distance calculated.

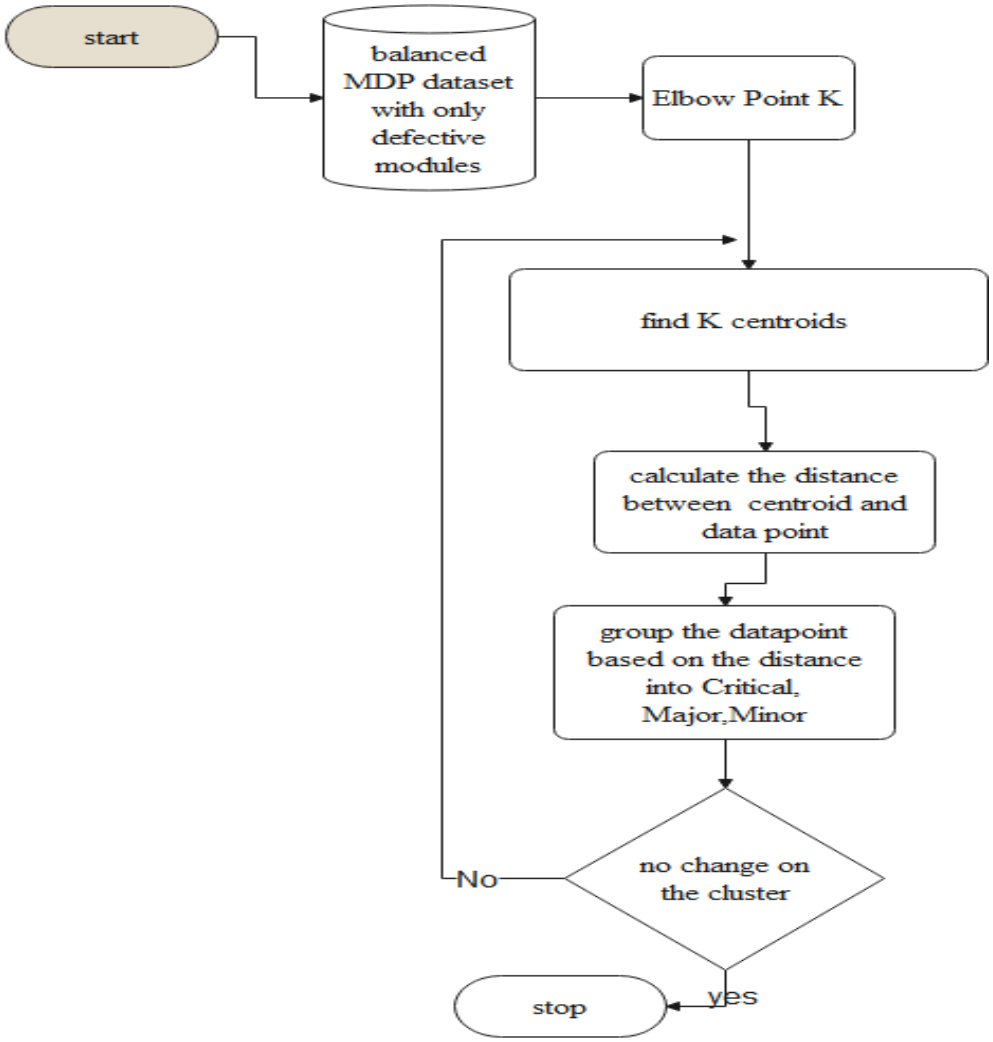


Figure 4.4 K-means clustering flow diagram for NASA MDP dataset

4.3. Machine Learning Models Building

This component of the proposed framework of Software Defect Severity prediction applies a machine learning classifier on clustered defective modules and non-defective modules of the dataset. In this study machine learning algorithms DT, RF, SVM, KNN is used to build the classification model from which could be selected for the prediction of Software Defect

Severity level. Instances with metrics (features) and labels are extracted from the dataset. The dataset is divided into a training set and test set to evaluate the model. The modeling process finds patterns in the training set of the dataset that maps instances with their metrics (features) to the target class by using the learning algorithm. New software defect metrics can be used for predicting severity level by using the trained prediction model.

The study proposed a random forest classifier, which is a meta estimator that fits multiple decision tree classifiers on different sub-samples of the dataset. Builds multiple decision trees and merges them by using the bagging method.

The study proposed a decision tree classifier, which creates a model based decision rules inferred from the data features to predict a value of the target variable, use a branching structure to map the output of the decision.

The study proposed a support vector machine classifier, the One-vs-Rest (OVR) strategy of support vector machine is proposed in this study to handle the multiclass problem. It used to separate one group from another of the classes.

The study proposed a k-nearest neighbor classifier, which assigns the number of k-neighbors and then calculates the distance between the k-neighbors.

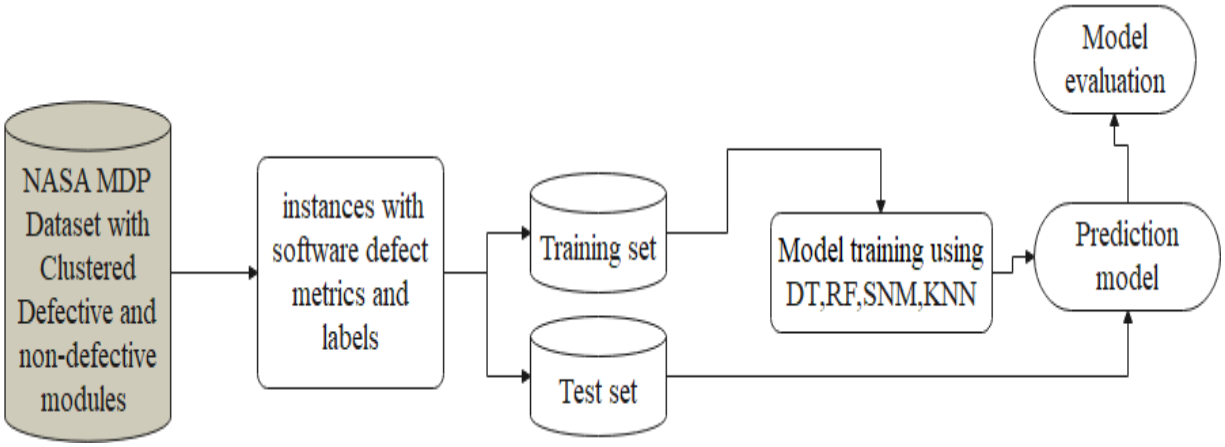


Figure 4.5 Model Building Flow diagram

4.4. Models Evaluation

Model evaluation is an important aspect that helps us to choose the best algorithm for our data. It helps us to know how well our algorithm will perform in the future. This study proposed performance evaluation metrics such as accuracy, precision, recall, F-measure, and confusion matrix, as described in chapter three for evaluating the learning ability of the model trained on the dataset for defect severity level prediction.

CHAPTER FIVE

5. Implementation and Experimentation

This chapter discusses the implementation of Software Defect Severity Prediction by using the methodology and proposed solutions in the previous chapter. This study implements a different experimental approach using the best machine learning techniques. Performs five experiments by using five datasets and four machine learning algorithm, compare the algorithm, and propose the best one.

5.1. Implementation Environment

To implement Software defect Severity Prediction this study used different development tools and packages. Each proposed solution is implemented using a python programming language. Python is preferred because of its simplicity, consistency, and can access to different libraries. Python is used for preprocessing modeling to performance evaluation, list of tools and python packages used are described in Table 5.1.

Table 5.1 Description of Tools and Python Packages Used in the Implimentation

Tools		
Tools	***Version***	***Description***
Anaconda Navigator	1.9.7	Used to launch the developments environment and packages
Jupyter notebook	6.0.0	Allows us to create and share documents that contain data visualization, equations, statistical modeling, data cleaning, and machine learning.
Python	3.7.4	Is one of the easiest and understandable programming language to develop machine learning

Microsoft Excel	2010	Used for data preparation
Python package		
Scikit-learn	0.21.3	A set of python modules called Sklearn used for clustering, training, and testing model
NumPy	1.16.1	Numerical python library that contains a multi-dimensional array, matrix data structure. Used to convert data to NumPy array
Pandas	0.25.1	Allows us to import data from various file formats like CSV, merging, reshaping.
Matplotlib	3.1.1	This study used it for the result, and data, visualization.
Seaborn	0.9.0	This study used to plot graphs in
Imblearn	0.7.0	the Method used to generate a dataset that has an equal ration class, balanced class.

5.2. Deployment Environment

The tools which are discussed above in section 5.1 have been deployed on a personal computer equipped with a processor Intel® Core™ i7-6500UM, CPU 2.50GHz, 2 Core(s), 8 Gigabytes of physical memory, 1 terabyte hard disk storage capacities. The operating system is Windows 10 Pro, 64 bits.

5.3. Dataset Description

For this study five projects are selected namely CM1 which is a project from NASA spacecraft instrument and written in C language, PC1 flight software from earth-orbiting satellite written

in C, PC3 from flight software for earth-orbiting satellite written in C++, KC1 from storage management for receiving ground data and written in C++, and KC3 from storage management for processing ground data and written in Java, Those datasets are selected based on the number of defective modules they have. The study didn't include a dataset with less than 5% of a defective module and one dataset with a high number of missing values. The rest of the two datasets are not included because important metrics are not included in the dataset. The following table shows the list of attributes(metrics) in the dataset with their description.

Table 5.2 Dataset Metrics

No	***Attribute/metrics***	***Description***
1.	LOC_BLANK	Number of blank lines in a module
2. .	BRANCH COUNT	Number of logical branches
3.	CALL PAIRS	Executable calls between modules
4.	LOC_CODE_AND_COMMENT	The number of lines that contain both code and comment in a module
5.	LOC_COMMENTS	The number of lines which contains comments in a module.
6.	CONDITION_COUNT	Number of iteration in a module
7.	DECISION_COUNT	Number of decisions in a module
8.	CYCLOMATIC_COMPLEXITY	Shows measure for the amount of decision logic in a single software module.Measures the complexity of modules structure.
9.	CYCLOMATIC_DENSITY	It is the measure of the amount of logic per source line.

10.	DECISION_DENSITY	It measures cyclomatic complexity per line of source code.
11.	DESIGN_COMPLEXITY	It is the measure of the interaction between modules in a system
12.	DESIGN_DENSITY	Measures number of path calling by divided by Cyclomatic Complexity
13.	EDGE COUNT	It's the measure of the number of edges in the flowgraph.
14.	ESSENTIAL_COMPLEXITY	It's the measure of the degree to which a given software has unstructured constructs.
15.	ESSENTIAL_DENSITY	Shows the ratio of essential complexity to cyclomatic complexity.
16.	LOC_EXECUTABLE	The number of lines that are executable for a given module(not blank or comment)
17.	PARAMETER_COUNT	The number of parameters in a given module.
18.	GLOBAL_DATA_COMPLEXITY	Count the cyclomatic complexity of modules structure related to a global parameter.
19.	GLOBAL_DATA_DENSITY	The ratio of global data logic to total logic
20.	HALSTEAD_CONTENT	Shows the information content of the program.

21.	HALSTEAD_DIFFICULTY	Measures the difficulty of reading the program which implies how briefly the program was written.
22.	HALSTEAD_EFFORT	The mental effort required to implement a program.
23.	HALSTEAD_ERROR_EST	Calculates errors that are present in a program.
24.	HALSTEAD_LENGTH	Measures usage of all operators and operands that are present in the program
25.	HALSTEAD_LEVEL	Measures level of abstraction provided by the programming language.
26.	HALSTEAD_PROG_TIME	Shows time needed to translate the algorithm into implementation in a specific programming language.
27.	HALSTEAD_VOLUME	Shows the size of information used to identify the program
28.	MULTIPLE_CONDITION_COUNT	Measures the number of multiple conditionals in a module.
29.	MODIFIED_CONDITION_COUNT	Measures the number of modified conditionals in a module.
30.	NODE_COUNT	Count the number of nodes in a flowgraph
31.	NUM_OPERANDS	Number of operands
32.	NUM_OPERATORS	Number of operators

33.	NUM_UNIQUE_OPERANDS	Number of unique operand
34.	NUM_UNIQUE_OPERATORS	Number of unique operator
35.	NUMBER_OF_LINES	The number of lines in a module, count every line from open bracket to close bracket.
36.	PERCENT_COMMENTS	Percentage of the code which is a comment
37.	LOC_TOTAL	Count the total number of lines for a given module.

5.4. Preprocessing implementation

Preprocessing methods have been implemented for cleaning the dataset, handling missing values, resampling the dataset for balancing the biased classes, and clustering the dataset to the appropriate group of class labels for prediction.

In this study, python programming language is used to implement software defect severity level prediction. To perform the implementation, we imported library packages and loaded the dataset used in the implementation by using the following python code in figure 5.1.

```
import csv
import pandas as pd
data1=pd.read_csv("CM1.csv")
data2=pd.read_csv("PC1.csv")
data3=pd.read_csv("PC3.csv")
data4=pd.read_csv("KC1.csv")
data5=pd.read_csv("KC3.csv")
```

Figure 5.1 Python code for loading the datasets

StandardScaler() method used for feature scaling which is called standardization, this method scales the values for each numerical features in the dataset.

```
from sklearn.preprocessing import StandardScaler
x= StandardScaler().fit_transform(data)
```

Figure 5.2 Python code for dataset standardization

5.4.1. Implementation for cleaning the dataset

To check and clean the dataset different python methods are used, .info() methods are used to show the rows count and the data types, describe () methods are used to show the main statistics for every numerical column in our dataset, isNull().values.any() methods are used to check if the dataset has nan values and missing values .all()method is used to check if the dataset has the same values through the table. Those methods return True if there is a value or False if there is no value, drop() function used to drop true values that have missing or nan values and return cleaned data.

5.4.2. Implementation for Balancing Dataset

To balance the classes of defective and non-defective classes of selected datasets using the proposed method, the study used imblearn python package that gives access to resampling techniques. RandomUnderSampler sub-module of imblearn package is used to implement the under-sampling techniques and RandomOverSampler to implement oversampling techniques.

RandomUnderSampler () class under-sample majority class which is a non-defective class by randomly picking samples from the dataset, the class takes sampling strategy to specify the ration of under-sampling techniques; in this study, we used 0.8 ratios to minority class to under-sample the dataset, this means the majority class is under-sampled 80% as the number of the minority class. *RandomOverSampler()* over-samples the minority by randomly duplicating the minority class. we used 0.6 ratios to the majority class to oversample the dataset. This means the minority class is oversampled 60% as the majority class. fit_resample () function is called to fit and apply the transformed data to the dataset.

```
import imblearn
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler
from collections import Counter
```

```
over = RandomOverSampler(sampling_strategy=0.6)
under = RandomUnderSampler(sampling_strategy=0.8)
X_over, y_over = over.fit_resample(X, y)
X_combined_sampling, y_combined_sampling = under.fit_resample(X_over, y_over)
```

Figure 5.3 Sample code for undersampling and oversampling implementation

5.4.3. Implementation for k-means clustering

To cluster the dataset into severity groups using the proposed algorithm, we used a python sci-kit learning library package and started by importing the important library package for modeling.

```
import csv
import pandas as pd
from pandas import DataFrame
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
from sklearn.cluster import KMeans
import sklearn.cluster as cluster
import numpy as n
```

Figure 5.4Importing important packages for clustering

After balancing the dataset only defective classes are clustered to severity levels. For severity level assessment only defective software modules are needed.

```
# # loading dataset with only defect yes
data1 = pd.read_csv("KC1Yes.csv")
data2 = pd.read_csv("CM1Yes.csv")
data3 = pd.read_csv("KC3Yes.csv")
data4 = pd.read_csv("PC1Yes.csv")
data5 = pd.read_csv("PC3Yes.csv")
```

Figure 5.5 python code for loading dataset with only defective class

This study used the elbow method to select an appropriate number of clusters, the method is implemented using the KMeans class from sklearn.cluster package library of scikit learn, the class accepts parameter numbers of clusters used to find by iterating through the range. Kmeans.fit() used to fit in to scaled data to find the cluster. Then we select the number of clusters, where WCSS (Within Cluster Sum of Squares) the measure of the variability between data points within each cluster begins to laying off, where it doesn't decrease with every iteration.

```
from sklearn.cluster import KMeans
wcss = []

for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 500, n_init = 10, random_state = 0)
    kmeans.fit(y)
    wcss.append(kmeans.inertia_)

#Plotting the results onto a line graph, allowing us to observe 'The elbow'
plt.style.use("fivethirtyeight")
plt.plot(range(1, 11), wcss)
plt.title('The elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS') #within cluster sum of squares
plt.show()
```

Figure 5.6 Python code implementation for elbow method

K-means clustering implemented using the number of clusters obtained by the elbow method, three clusters are obtained from the method. Kmeans clustering algorithm is implemented using KMeans class used from sklearn.cluster package library of scikit learn. Kmeans.fit_predict() method compute cluster centers ad predicts cluster group for each data. Kmeans.cluster_center shows the last location of the centroids. Kmeans.labels returns clustered group

```
kmeans=cluster.KMeans(n_clusters=3,init="k-means++",max_iter = 500, n_init = 10, random_state = 0)
y_kmeans=kmeans.fit_predict(x)
kmeans.cluster_centers_
data['Severity'] =kmeans.labels_
```

Figure 5.7 K-means clustering python code

5.5. Machine Learning Model Implementation

To build machine learning models using the proposed algorithm, we used a python sci-kit learning library package and followed by the method that is importing the important library package for modeling and metrics for model evaluation.

```
import csv
import pandas as pd
from pandas import DataFrame
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.datasets import make_multilabel_classification
```

Figure 5.8 Importing necessary packages for modeling

After the dataset is clustered into severity groups the new clustered datasets are used for predicting the severity level of a software defect. The dataset with severity label is loaded, and by using the train_test_split() method we split the dataset into 80% training and 20% test set, X_train, y_train to train using the *fit()* method with the appropriate set parameter for each classifier instantiate the class. X_test, y_test to test the model.

```
data1 = pd.read_csv("KC1cluster.csv")
data2 = pd.read_csv("CM1cluster.csv")
data3 = pd.read_csv("KC3cluster.csv")
data4 = pd.read_csv("PC11cluster.csv")
data5 = pd.read_csv("PC3cluster.csv")
```

```
X = data.drop('Severity', axis=1)
y = data['Severity']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

Figure 5.9 python code for splitting dataset to train and test set

To implement the RF classifier, *RandomForestClassifier* () from sklearn ensemble package is used to train the classifier. Grid search methods are used for hyperparameter tuning, *GridSearchCV()* from sklearn model selection used to generate candidates from a grid of parameter values specified with the param, while fitting the best combination between parameters are evaluated and will be retained. This classifier fits several decision tree classifiers as declare in n_*estimators* parameters.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

param = {"n_estimators":[100, 120, 150,1000],
        "max_depth":[3, 5, 10],
      "min_samples_leaf":[3, 5, 10],
         "min_samples_split":[2, 5, 10]
        }
#instanciate random forest
clf = RandomForestClassifier()
forest=GridSearchCV(clf, param)
#fit model to train the model
forest.fit(X_train,y_train)
```

Figure 5.10 Python code for instantiating RF and Fit the model

To implement DT classifier, *DecisionTreeClassifier()* from sklearn tree package is used to train the classifier. *GridSearchCV()* from sklearn model selection used to generate candidates from a grid of parameter values specified with the param, and build a decision tree classifier from the training set.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
param = {"max_depth":range(1,10),
              "min_samples_leaf": range(1, 9),
              "criterion": ["gini", "entropy"]}
#initializing tree classifier
clf= DecisionTreeClassifier()
DT = GridSearchCV(clf, param)
# fit the model to predict
DT.fit(X_train,y_train)
```

Figure 5.11 Python code for instantiating DT and Fit the model

The study used the *LinearSVC()* classifier of the sklearn package to building the SVM model. This classifier instantiate OVR parameter for multi-class classification

```
from sklearn.svm import LinearSVC
#inistanciate the training model
svclassifier =LinearSVC(multi_class='ovr',loss='squared_hinge', dual=True,
                        max_iter=10000,class_weight='balanced', C=0.01,
                        verbose=0)
# fit the model to train
svclassifier.fit(X_train,y_train)
```

Figure 5.12 Python code for instantiating SVM and Fit the model

To implement KNN model we used *KNeighborsClassifier()* from sklearn neighbors package to train the classifier. *GridSearchCV()* from sklearn model selection used to generate candidates from a grid of parameter values specified with the param, Implements learning for each data point based on the nearest neighbors of the data point generated by the grid search method.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
param={"leaf_size":range(3,10),
       "n_neighbors":range(1,10),
          "p":[1,2]
      }
#instanciate the model
clf = KNeighborsClassifier()
clfK=GridSearchCV(clf, param)
#fit to train the model
clfK.fit(X_train, y_train)
```

Figure 5.13 Python code for instantiating and Fit the model

5.6. Model Testing and Evaluation

The performance of the training model is evaluated using a 20% test dataset. The study used a *classification_report ()*, *accuracy_score()*, and *confusion_matrix()* methods from sklearn metrics package are used to evaluate the performance of the built models using predict the

value of the test set. These methods give a result of evaluation methods such as accuracy, precision, recall, and F1-score value of the model under testing

CHAPTER SIX

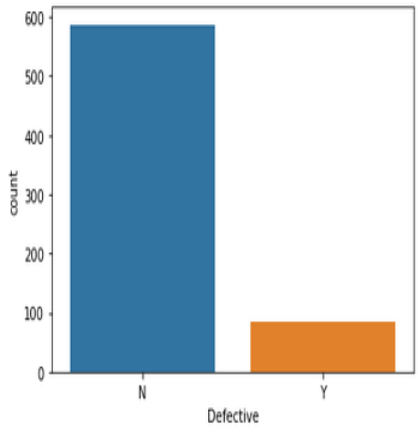
6. Result of the Study and Discussions

This chapter presents the result obtained from the experiment of the proposed Software Defect Severity level prediction using a machine learning algorithm. Here the result of dataset balancing techniques, clustering, and the result of classification models for each dataset is discussed. In the final section, we discuss the result obtained by each experiment in this study.

6.1. Result of dataset balancing techniques

To balance the selected class of the dataset the study used a combination of under-sampling and over-sampling techniques. The techniques are applied to five datasets named CM1, PC1, PC3, KC1, and KC3. The following Figures show the original and balanced dataset using a combination of oversampling and under-sampling techniques. As shown in Figures Y is the defective class and N is the non-defective class of the dataset.

Figure 6.1 shows the CM1 original dataset with 671 modules, from which 587 modules are non-defective and 84 modules are defective. After balancing the dataset as shown in Figure 6.2 the defective class modules are 352 class and non-defective class are 440 data with a total number of 792 data.



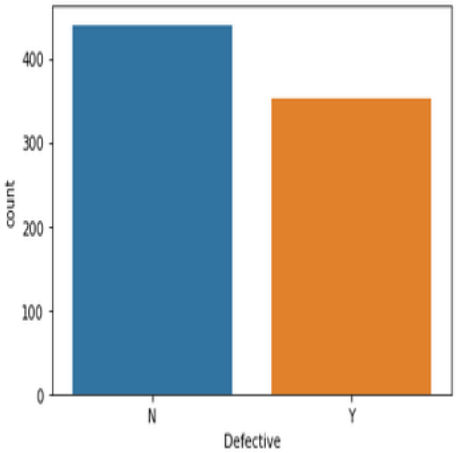
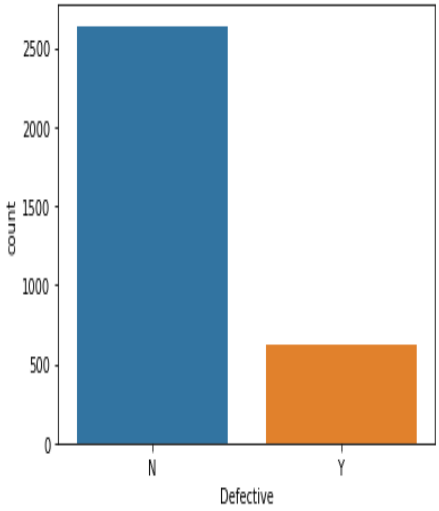


Figure 6.1 Original CM1 dataset. *Figure 6.2 class distribution after balancing CM1*

Figure 6.3 shows the KC1 original dataset with 3257 modules, from which 2638 modules are non-defective and 619 modules are defective. After balancing the dataset as shown in Figure 6.4 the defective class is 1582 class and the non-defective class are 1977 data with a total number of 3559 data.



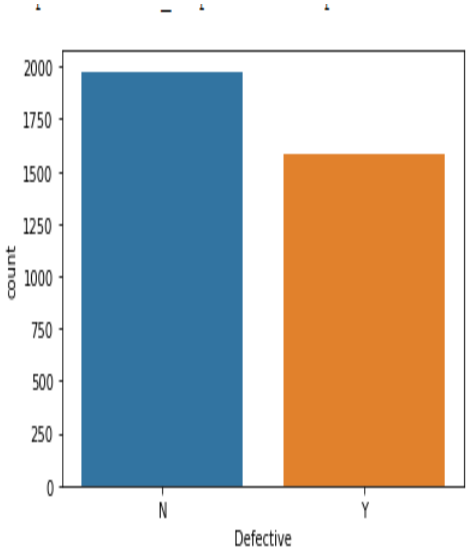
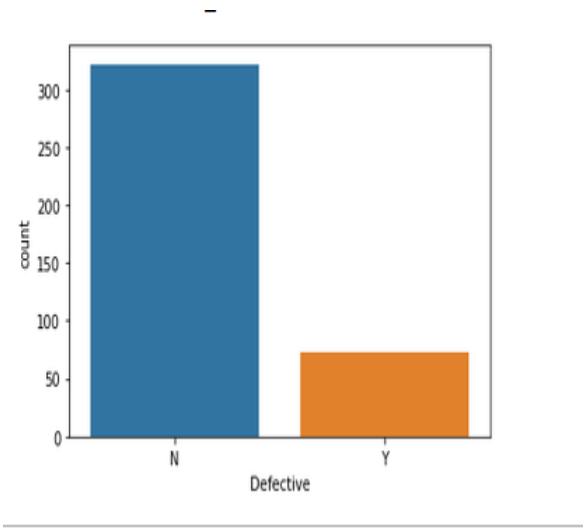


Figure 6.3 original KC1 dataset. *Figure 6.4 class distribution after balancing KC1*

Figure 6.5 shows the KC3 original dataset with 394 modules, from which 322 modules are non-defective and 72 modules are defective. After balancing the dataset as shown in Figure 6.6 defective class are 193 class and non-defective class are 241 data with a total number of 434 data.



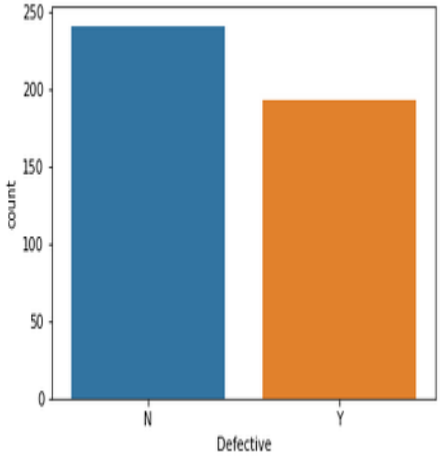
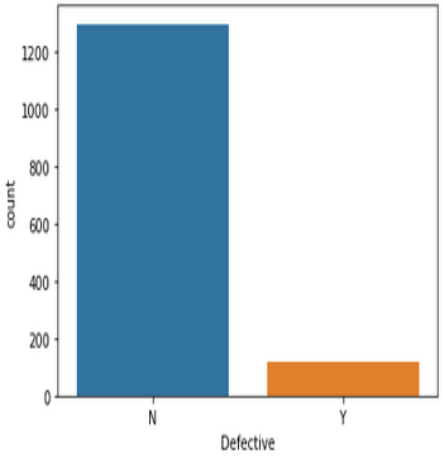


Figure 6.5 original KC3 dataset. *Figure 6.6 class distribution after balancing of KC3*

Figure 6.7 shows the PC1 original dataset with 1414 modules, from which 1298 modules are non-defective and 116 modules are defective. After balancing the dataset as shown in Figure 6.8 defective class is 778 and the non-defective class are 972 data with a total number of 1750 data.



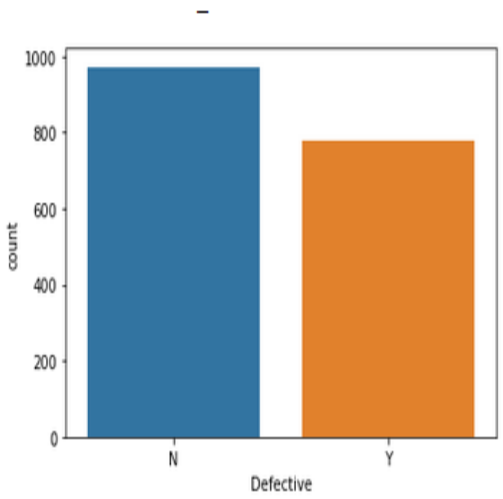
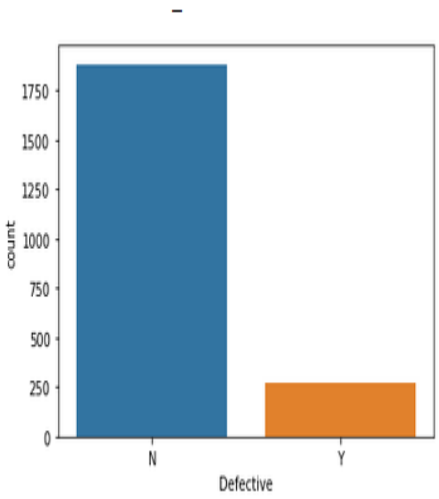


Figure 6.7. Original PC1 dataset. *Figure 6.8 class distribution after balancing PC1*

Figure 6.9 shows the PC3 original dataset with 2152 modules, from which 1884 modules are non-defective and 267 modules are defective. After balancing the dataset as shown in Figure

6.10 the defective class are 1130 class and non-defective class are 1412 data with a total number of 2542 data



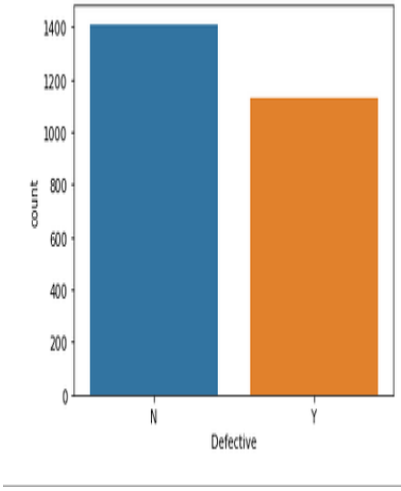


Figure 6.9 Original PC3 dataset *Figure 6.10 class distribution after balancing PC3*

6.2. K-means clustering result

Defected classes from each selected dataset are clustered into critical, major, and minor cluster groups which are appropriate for defect severity prediction. The clustering result for each dataset is presented below.

The appropriate number of cluster is selected using the elbow method, the number is selected where the elbow point started to laying off.

The elbow point for PC3 is shown in Figure 6.11 the number of cluster is 3 where the elbow is started to lay down

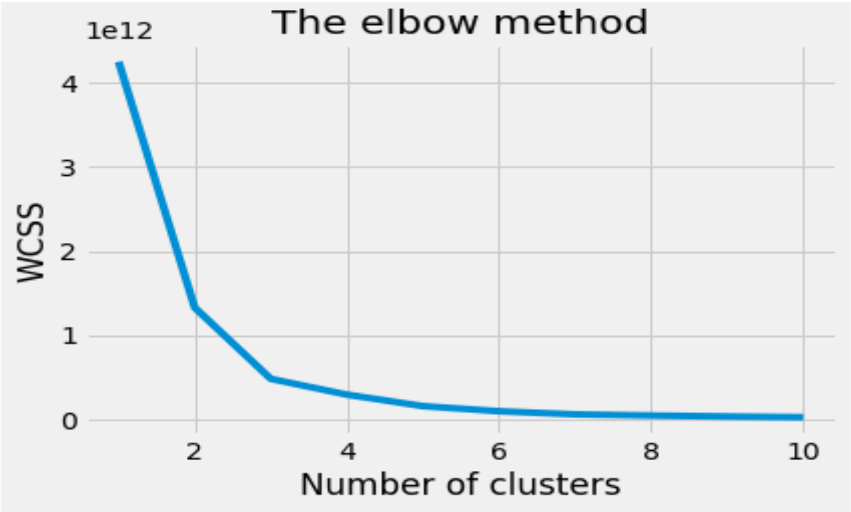


Figure 6.11 Elbow point for PC3

Table 6.1 shows cluster groups of PC3, From the PC3 dataset 1130 defective classes are clustered into 3 groups which result in 243 items on cluster 0, 833 items on cluster 1, 54 data on cluster 2.

Table 6.1 Cluster group for PC3

Cluster group	***Items***	***Severity level***
Cluster 0	243	Major
Cluster 1	833	Minor
Cluster 2	54	Critical

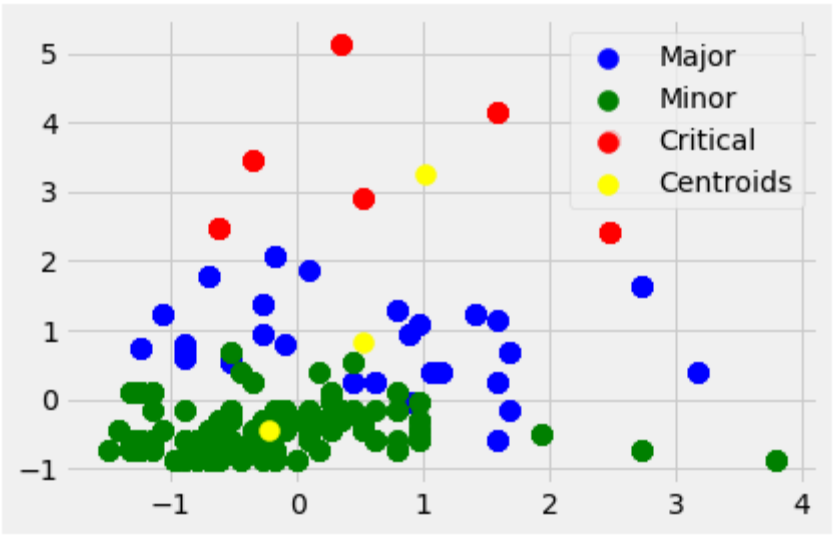


Figure 6.12 Cluster group for PC3

The elbow point for KC1 is shown in Figure 6.13 the elbow result shows 3 as the number of clusters where the point started on laying off.

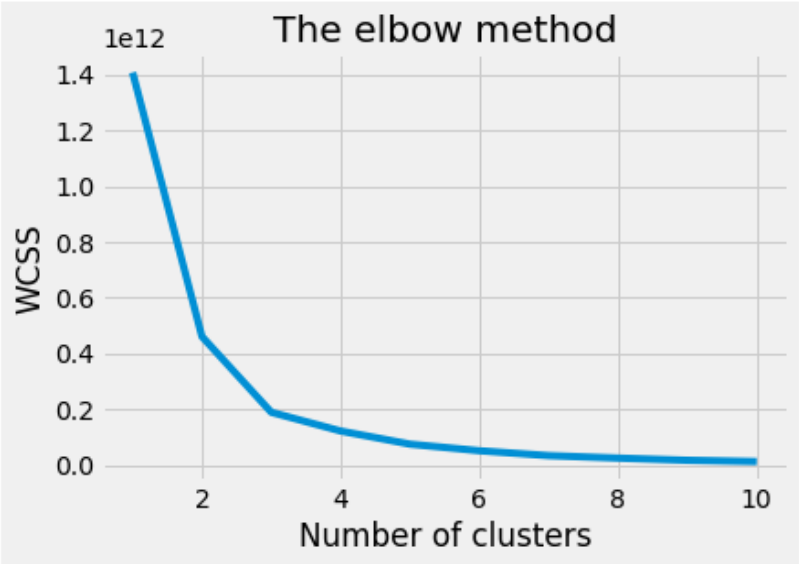


Figure 6.13 Elbow point for KC1

Table 6.2 shows cluster groups of KC1, From the KC1 dataset 1582 defective classes are clustered into 3 groups which result in 130 items on cluster 0, 1040 items on cluster 1, 412 data on cluster 2.

Table 6.2 Cluster group for KC1

Cluster group	***Items***	***Severity level***
Cluster 0	130	Critical
Cluster 1	1040	Minor
Cluster 2	412	Major

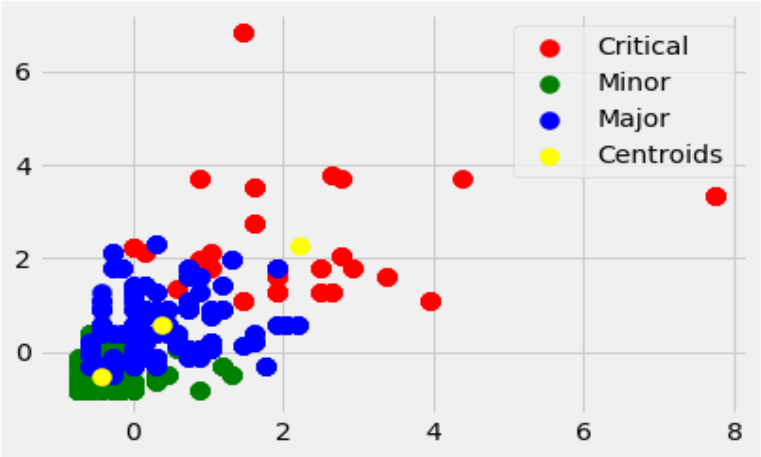


Figure 6.14 Cluster group of KC1

The elbow point for CM1 is shown in Figure 6.15 the elbow result shows 3 as the number of clusters where the point started on laying off.

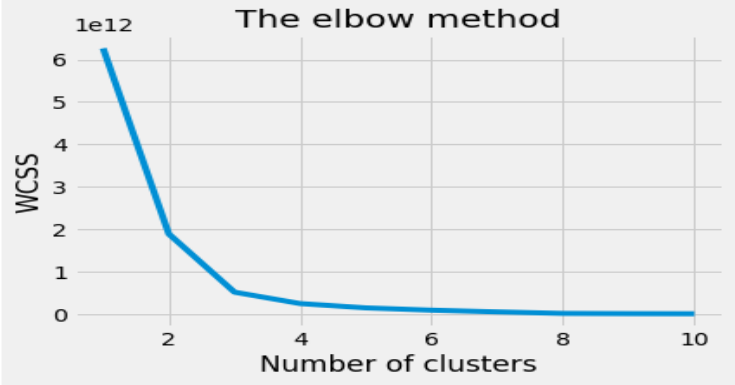


Figure 6.15 Elbow point for CM1

Table 6.3 shows cluster groups of CM1, From the CM1 dataset 352 defective classes are clustered into 3 groups which result in 15 items on cluster 0, 231 items on cluster 106 data on cluster 2.

Table 6.3 Cluster group for CM1

Cluster group	***Items***	***Severity level***
Cluster 0	15	Critical
Cluster 1	231	Minor
Cluster 2	106	Major

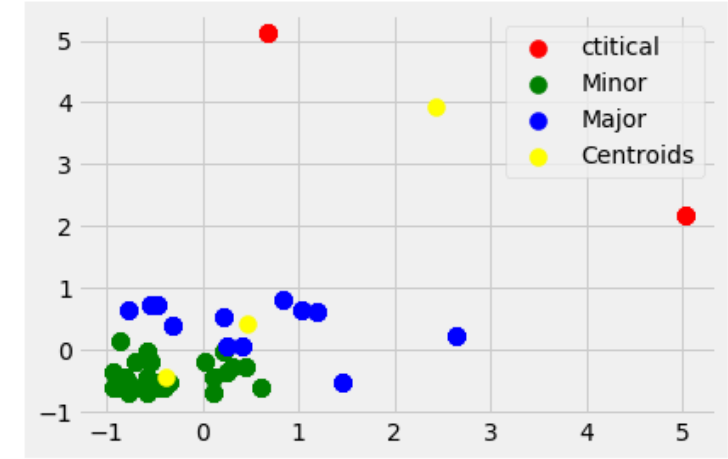


Figure 6.16Cluster group for CM1

The elbow point for PC1 is shown in Figure 6.16 the elbow result shows 3 as the number of clusters where the point started on laying off.

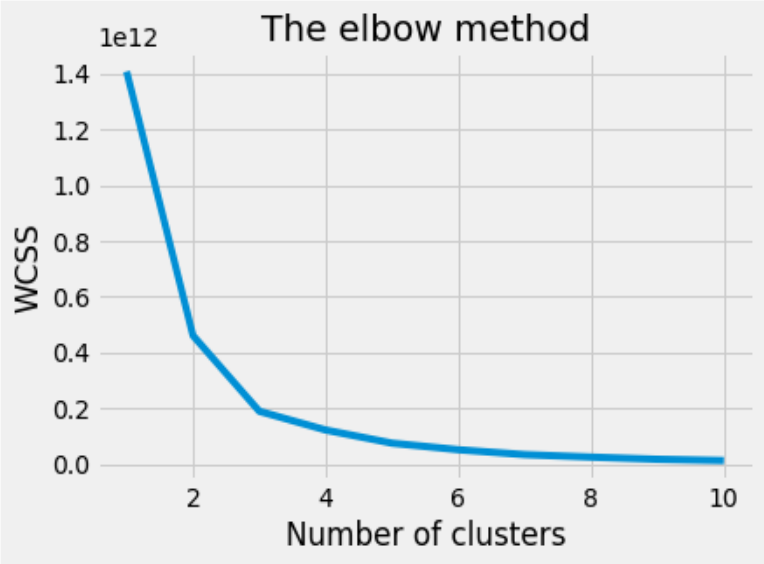


Figure 6.17 Elbow point for PC1

Table 6.4 shows cluster groups of PC1, From the PC1 dataset 778 defective classes are clustered into 3 groups which result in 296 items on cluster 0, 28 items on cluster 454 data on cluster 2.

Table 6.4Cluster group for PC1

Cluster group	***Items***	***Severity level***
Cluster 0	295	Major
Cluster 1	28	Critical
Cluster 2	454	Minor

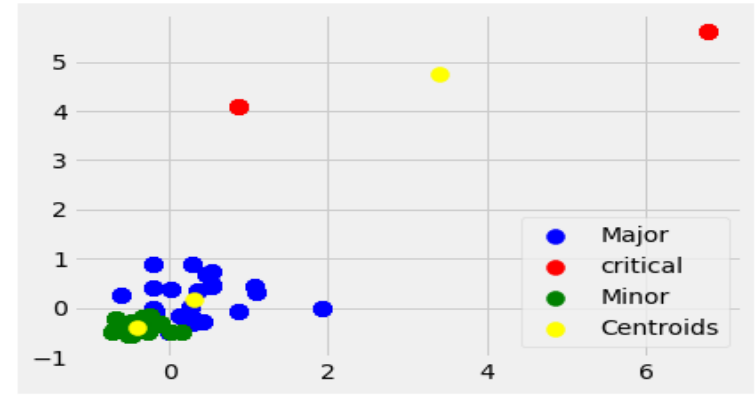


Figure 6.18 Cluster group for PC1

The elbow point for KC3 is shown in Figure 6.16 the elbow result shows 3 as the number of clusters where the point started on laying off.

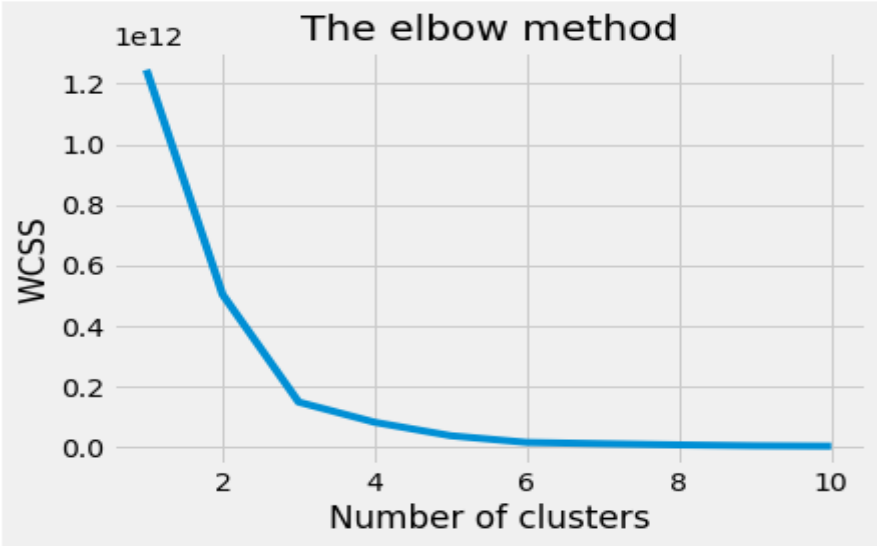


Figure 6.19 Elbow point for KC3

Table 6.5 shows cluster groups of KC3, from the KC3 dataset 193 defective classes are clustered into 3 groups which result in 94 items on cluster 0, 78 items on cluster 1, 21 data on cluster 2.

Table 6.5 Cluster group for KC3

Cluster group	***Items***	***Severity level***
Cluster 0	94	Minor
Cluster 1	78	Major
Cluster 2	21	Critical

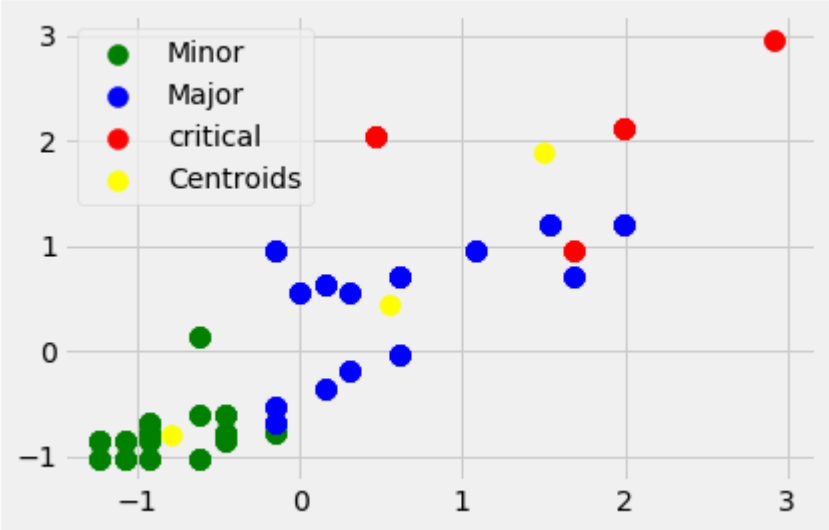


Figure 6.20 Cluster group for KC3

6.3. Models Evaluation Result

For the experiment, we used 5 clustered datasets CM1, KC1, KC3, PC1, and PC3 which result in four models for each dataset. The performance of the trained models is tested using 20% test set. The result of each trained model testing accuracy, precision, recall, and F1-score are presented for, DT, RF, KNN, and SVM respectively are shown in the tables below.

The result in **Table 6.6** shows the performance score of the DT algorithm for each dataset, the lower accuracy recorded is 83.8 % on KC1 and the higher accuracy recorded is 94.3 % on CM1, the F1 score is between 84% and 94%.

Table 6.6 Performance score for DT

Datasets	***Accuracy***	***Precision***	***Recall***	***F1-score***
CM1	94.3	95	94	94
KC1	83.8	84	84	84
KC3	90.8	93	91	91
PC1	93.7	93	93	93
PC3	90.3	92	90	90

The result in **Table 6.7** shows the performance score of the RF algorithm for each dataset, the lower accuracy obtained is 90.1% on KC1 and the highest accuracy is 97.4% on PC1, the F1 score is between 90% and 97%.

Table 6.7 performance Scores for RF

Datasets	***Accuracy***	***Precision***	***Recall***	***F1-score***
CM1	96.5	98	97	97
KC1	90.1	90	90	90
KC3	94.2	95	94	94
PC1	97.4	98	97	97
PC3	95.8	96	96	96

The result in **Table 6.8** shows the performance score of the KNN algorithm for each dataset, the lower accuracy obtained is 77.1% on KC3 and the highest accuracy is 89.3% PC1, the F1 score is between 77% and 89%.

Table 6.8 performance score for KNN

Datasets	***Accuracy***	***Precision***	***Recall***	***F1-score***
CM1	86.1	90	86	86
KC1	87.3	88	87	87
KC3	77.1	84	77	77
PC1	89.3	91	89	89
PC3	86.2	89	86	86

The result in **Table 6.9** shows the performance score of the SVM algorithm for each dataset, the lower accuracy obtained is 55.2% on KC3 and the highest accuracy is 73.4% on PC1, the F1 score is between 55% and 67.4%.

Table 6.9 Performance score for SVM

Datasets	***Accuracy***	***Precision***	***Recall***	***F1-score***
CM1	63.5	71	64	66
KC1	57.1	65	57	58
KC3	55.2	56	55	55
PC1	73.4	77	73	74
PC3	64.8	62	65	63

The bar chart in Figure 6.16 shows the visuals representation of the accuracy of each model based on the datasets in the above four tables. The blue represents the accuracy of the RF models, the brown represents the classification accuracy of DT models, the green represents the classification accuracy of KNN models and the purple represents the classification accuracy of SVM models. The x-axis is represented with the selected datasets and the y-axis is based on the accuracy result, which shows the accuracy of the RF models greater than that of the DT, KNN, and SVM model on each dataset.

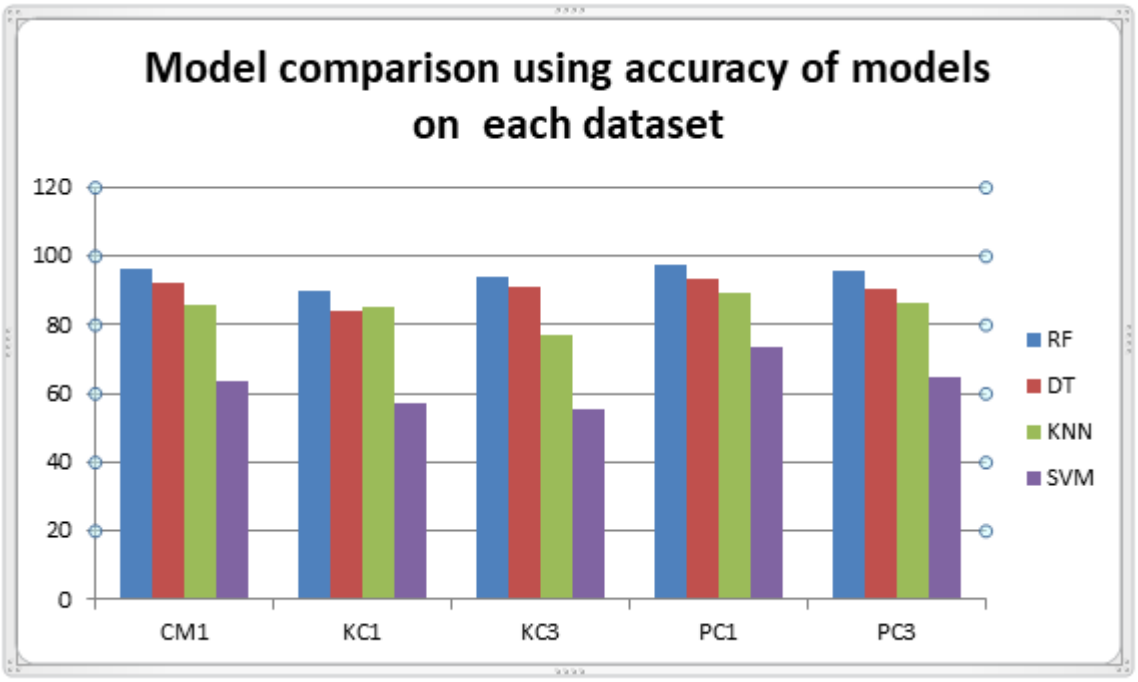


Figure 6.21 Model comparison using accuracy

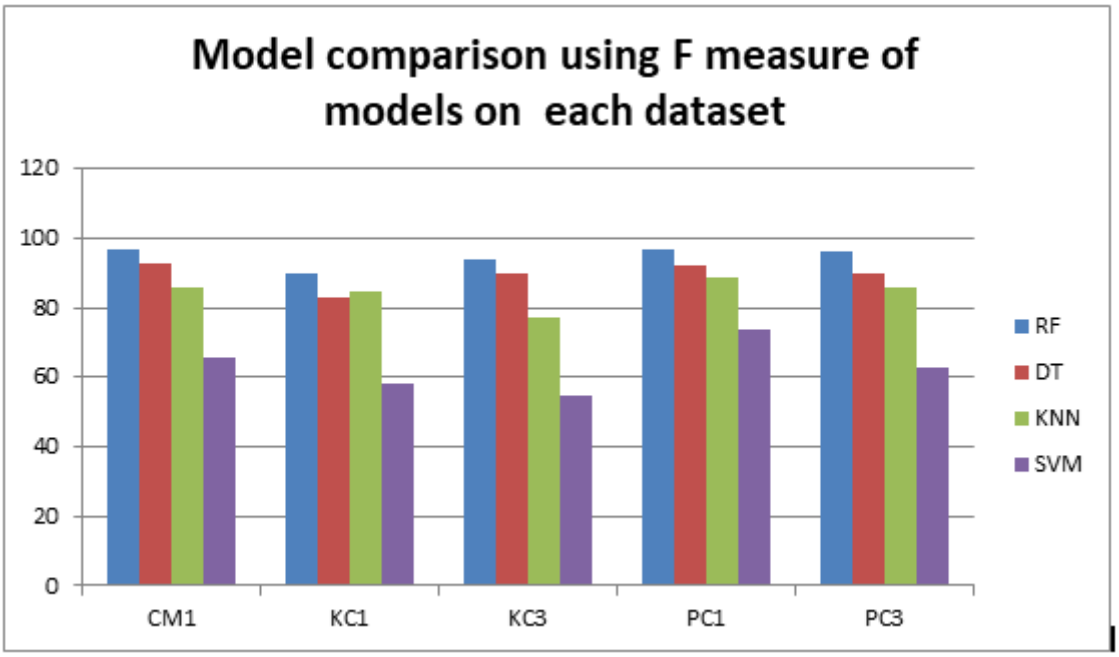
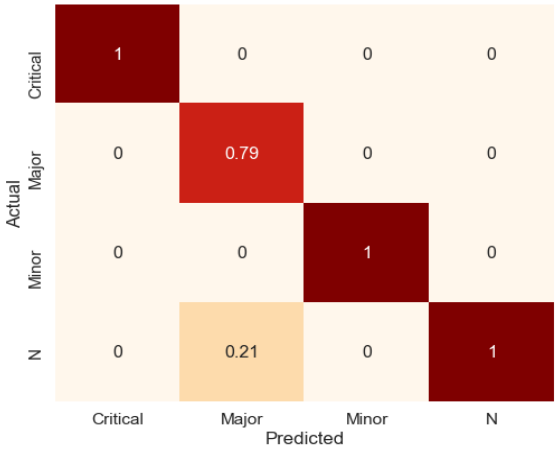


Figure 6.22 Model comparison using F-measure

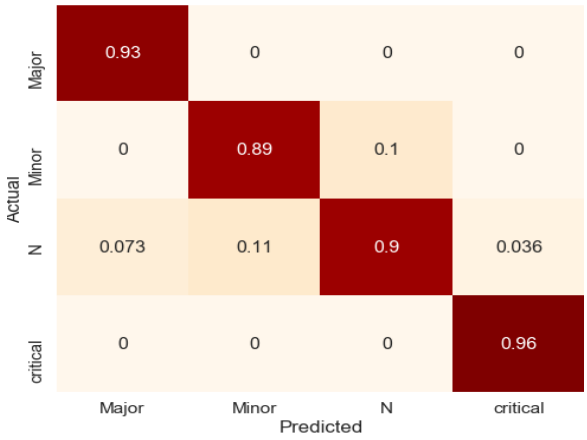
We used F-measure values to compare the four classifiers with recall and precision. Figure 6.21 shows the F1 measure values for the used algorithms in the five selected datasets. The x-axis is represented with the selected datasets and the y-axis is based on the F measure result. As shown in the figure, the RF model has the highest f1 score in all datasets followed by DT, KNN, and then SVM classifier.

The normalized confusion matrix result for the RF model for each dataset is shown in the figure below.

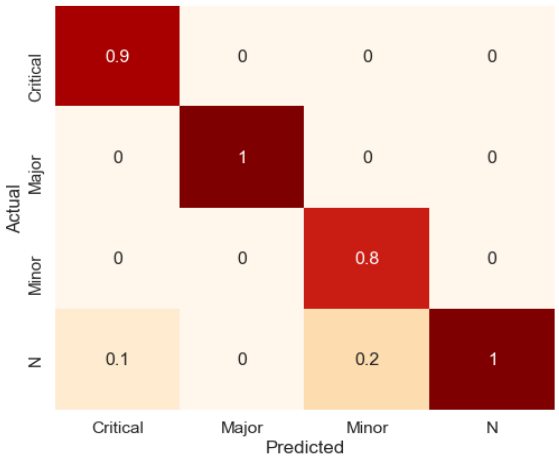
CM1



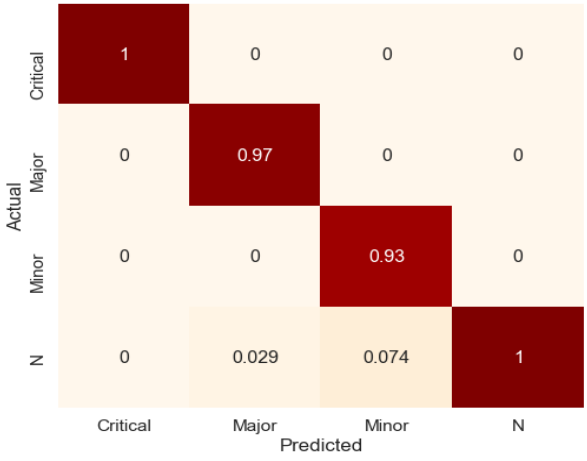
KC1



KC3



PC1



PC3

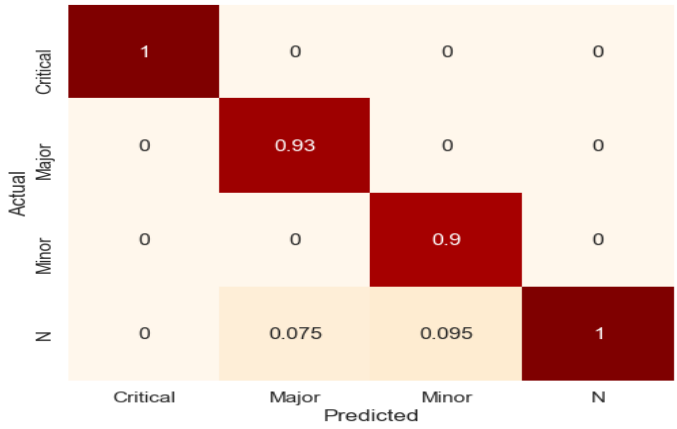
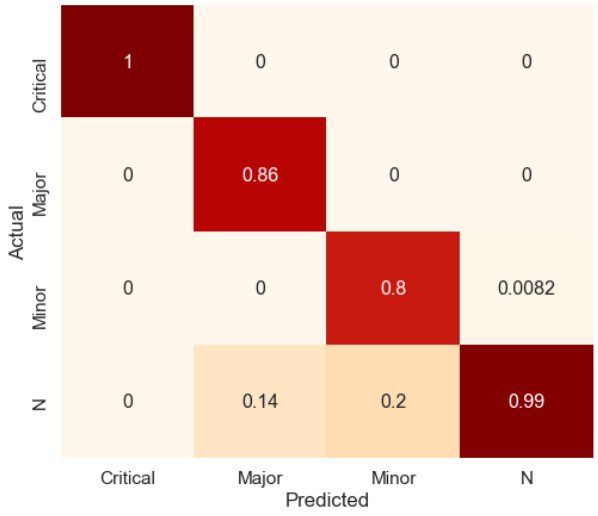


Figure 6.23 Confusion Matrix for RF models for each selected datasets

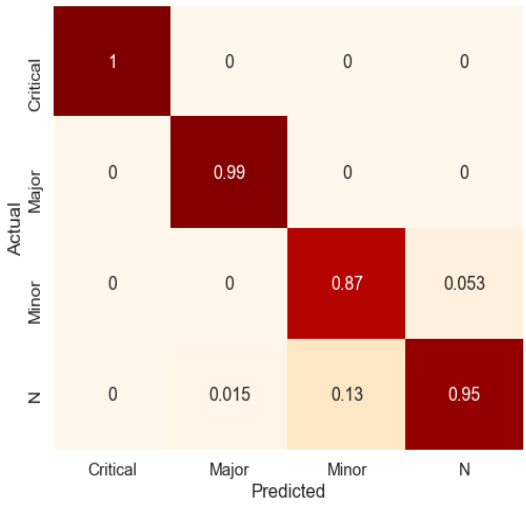
Figure 6.16 shows the confusion matrix for RF models. RF on CM1 classifies 100% of Critical, 79% of Major, 100% of Minor, and 100% of N class correctly; misclassification between Major and N is 21%. On KC1 93% of Major, 89% of Minor, 90 % of N, and 96% of critical class classified correctly, but 7.3% of Major class misclassified as N class, misclassification between Minor and N class is 11%. On KC3 90% of critical, 100% of Major,80% of Minor, and 100% of critical classified correctly, but 20% of Minor misclassified as N, misclassification between critical and N class is 10%. On PC1 100% of Critical, 97% of Major, 93% of Minor, 100% of N classes are classified correctly, but 3% Major class misclassified as N and 7.5% Minor class misclassified as N. On PC3 100% of Critical, 93% Major, 90% Minor and 100% of N class classified correctly, but 9.5% of Minor misclassified as N and 7.5% of Major class misclassified as N.

The normalized confusion matrix result for the DT model for each dataset is shown in the figure below.

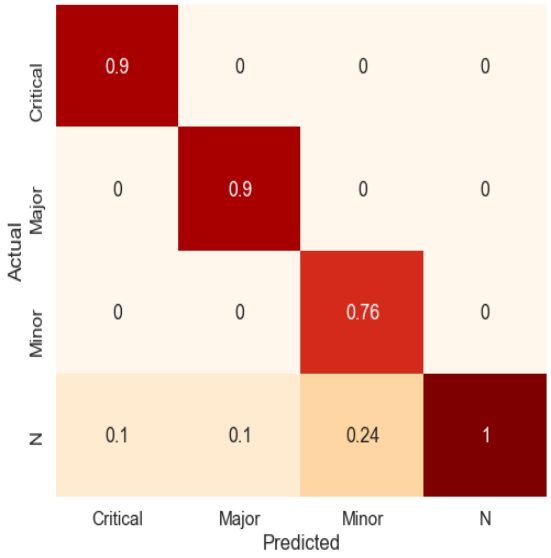
PC3



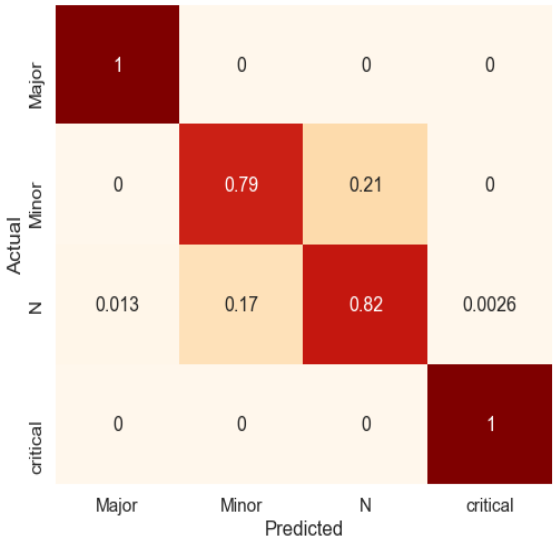
PC1



KC3



KC1



CM1

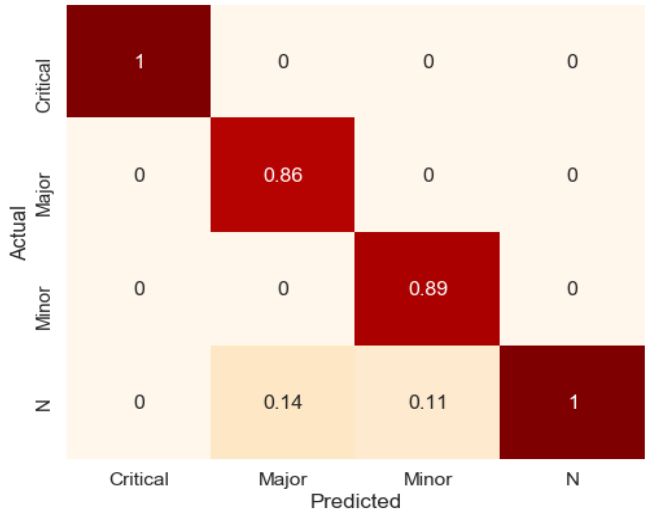
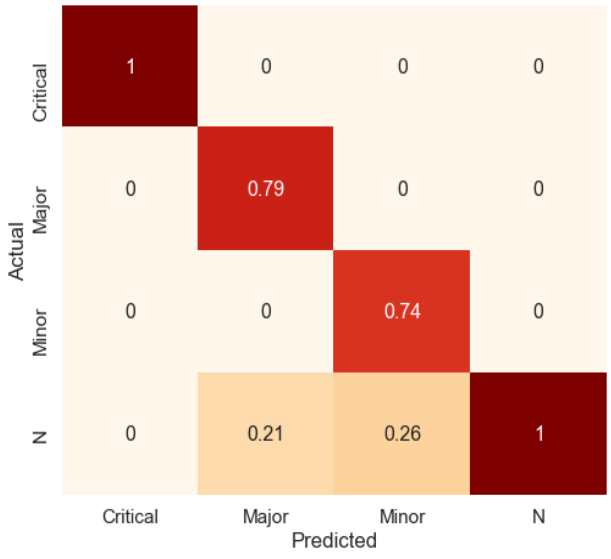


Figure 6.24 Confusion Matrix for Decision tree models for each selected datasets

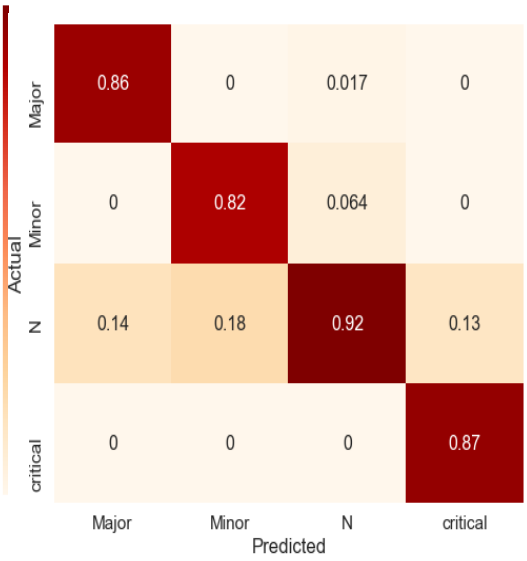
Figure 6.17 shows the confusion matrix for DT models. On PC3 100% of Critical, 86% Major, 80% of Minor, and 99% N classes classified correctly, but 14% of Major misclassified as N, 20% of Minor misclassified as N and 0.82% of N misclassified as Minor. On PC1 100% Critical, 99% of Major, 87% of Minor, and 95% of N classes are classified correctly. But 1.5% of Major classes are misclassified as N and 13% of Minor misclassified as N. on KC3 90% of Critical, 90% of Major, 76% of Minor and 100% of N classified correctly, but 10% of critical class misclassified as N,10% of Major class misclassified as N and 24% of Minor misclassified as N class. On KC1 100% of Major, 79% of Minor, 82% of N, and 100% of Critical classes are classified correctly, but 17% of Minor class misclassified as N, 21% of N class misclassified as Minor. On CM1 100% of Critical, 86% Major, 89% of Minor, 100 of N classes are classified correctly, but 11% of Minor class misclassified as N and 14% of Minor class misclassified as N.

The normalized confusion matrix result for the KNN model for each dataset is shown in the figure below.

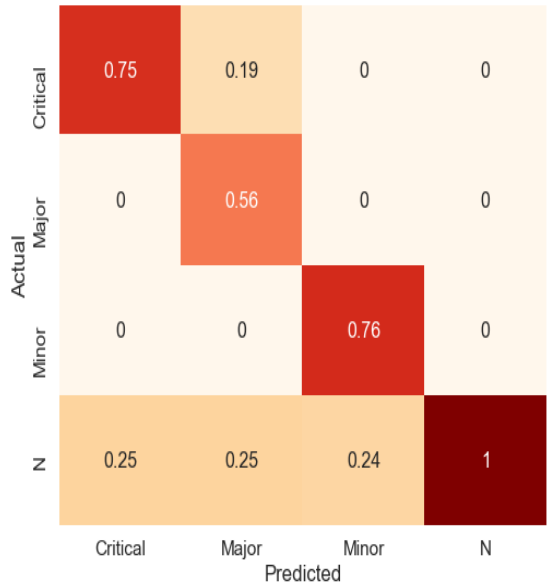
CM1



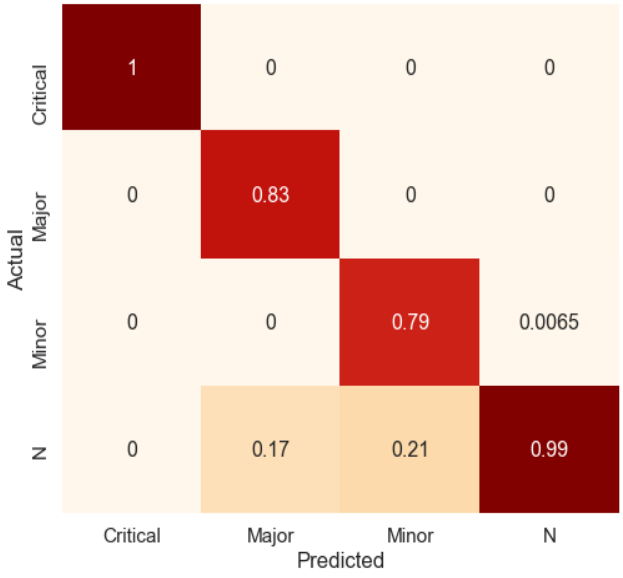
KC1



KC3



PC1



PC3

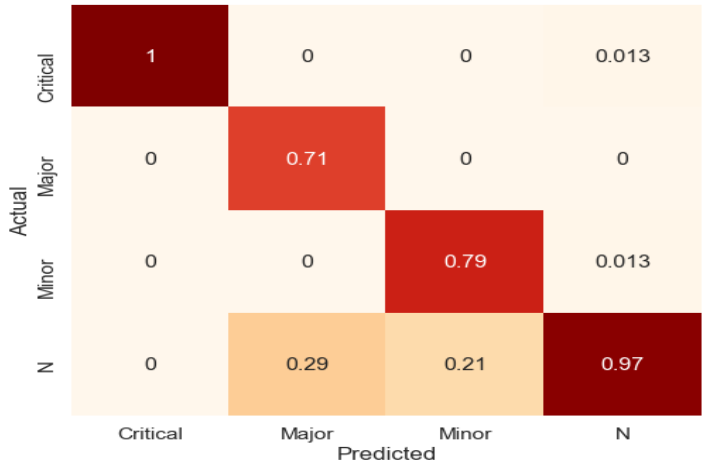
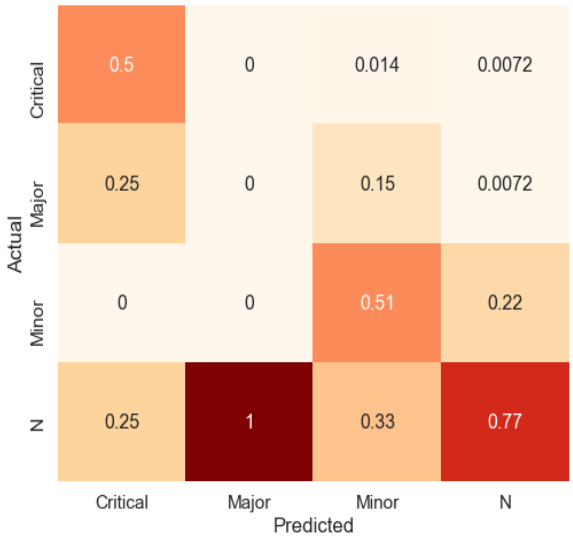


Figure 6.25 Confusion Matrix for K-nearest neighbor models for each selected datasets

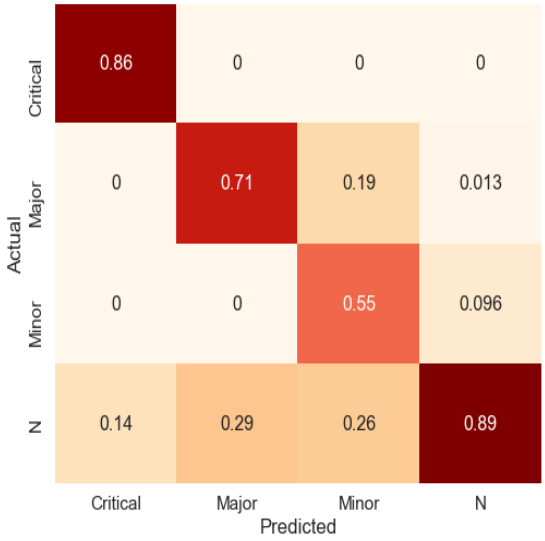
Figure 6.18 shows the confusion matrix for KNN models. On CM1 100% of Critical, 79% of Major, 74% of Minor, and 100% of N classes are classified correctly, but 21% of Major class misclassified as N, and 26% of Minor class misclassified as N. On KC1 86% of Major, 82% of Minor, 92% of N, 87% of Critical classes are classified correctly, but 14 % of Major classes are misclassified as N, 18% of Minor classes are misclassified as N, 6.4% N class misclassified as Minor, 1.7 % N class misclassified as Major, 13% critical classes misclassified as N and 6.1% of critical classes misclassified as Major. On KC3 75% of Critical, 56% of Major, 76% of Minor,100% of N are classified correctly, but 25% of Critical classes misclassified as N, 19% of Major classes misclassified as Critical, 25% of Major class misclassified as N and 24% Minor class misclassified as N. On PC1 100% of critical, 83% of Major, 79% of Minor, 99% of N are classified correctly, but 17% of Major class misclassified as N and 21% of Minor class misclassified as N. On PC3 100% critical, 71% major, 79% of Minor, and 97% of N classes are classified correctly, but 29% of Major class misclassified as N, 21% of Minor class misclassified as N, and 1.3% of N classes are misclassified as Critical and Minor.

The normalized confusion matrix result for SVM Model for each dataset is shown in the figure below.

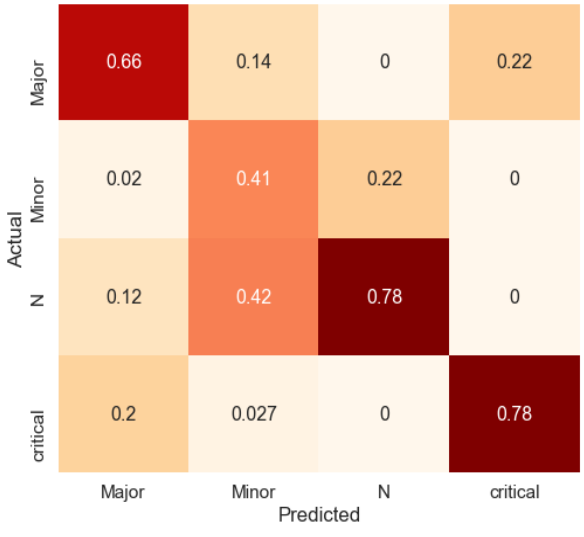
PC3



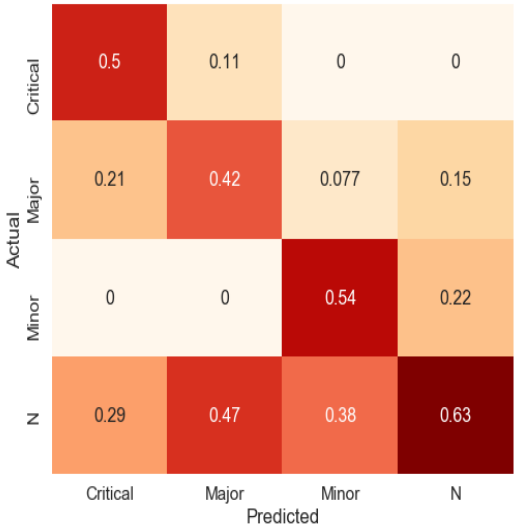
PC1



KC1



KC3



CM1

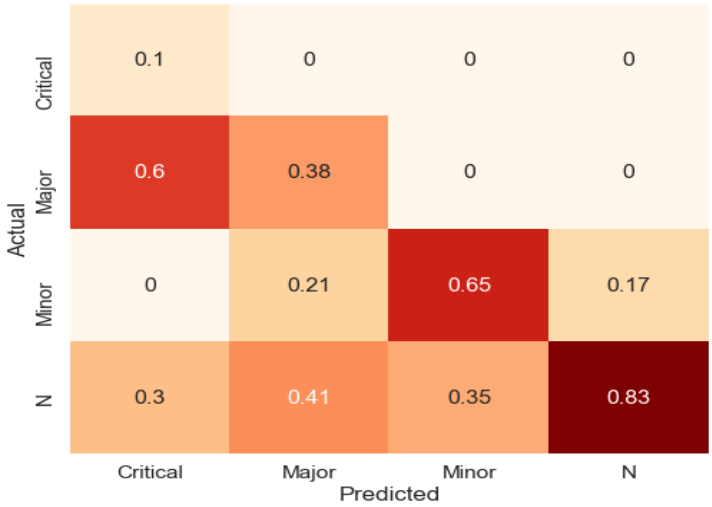


Figure 6.26 Confusion Matrix for SVM models for each selected datasets

Figure 6.19 shows the confusion matrix for SVM models. On PC3 50% of Critical, 0% of Major, 51% of Minor, and 77% of N are classified correctly, but 25% of Critical class is misclassified as N and 25% of Critical class is misclassified as Major, 100% of Major classes are misclassified as N, 33% of Minor classes are misclassified as N class, 15% of Minor class misclassified as Major, 0.14% of Minor class misclassified as Critical class, 22% class of N classes are misclassified as Minor, 0.72% of N class misclassified as Major and 0.72% N classes are misclassified as critical. On PC1 86% of Critical, 71% Major, 55% Minor, and 89% N classes are classified correctly, but 14% of Critical classes are misclassified as N, 29% of Major classes are misclassified as N, 19% of Minor class is misclassified as Major, 26% of Minor classes are misclassified as N and 9.6% of N class are misclassified as Minor. On KC1 66% of Major, 41% of Minor, 78% of N, and 78% of Critical classes are classified correctly, but 2 % of Major classes misclassified as Minor, 12% of Major classified as N, 20% of Major classes are misclassified as Critical, 14% of Minor classes are misclassified as Major, 42% of Minor class misclassified as N, 22% of N class misclassified as Minor, and 22% of Critical class misclassified as Major. On KC3 50% of Critical, 42% of Major, 54% of Minor, and 63% of N classes are classified correctly, but 29% of Critical class misclassified as N, 21% of Critical misclassified as Major, 47% of Major class misclassified as N, 11% of Major class misclassified as Critical, 7.7% of Minor class misclassified as Major, 38% of Minor class

misclassified as N, 22% of N class misclassified as Minor and 15% of N misclassified as Major. On CM1 10% of Critical class, 38% of Major, 65% of Minor, and 90% of N class classified correctly, but 60% of critical class misclassified as Major, 30% Critical class misclassified as N, 41% of Major class misclassified as N, 21% of Major class misclassified as Minor, 35% of minor class misclassified as N, and 17% of N class misclassified as Minor.

The normalized confusion matrix in Figures 6.16, 6.17, 6.18, 6.19 respectively shows the classification result of each model. The actual class and the predicted class are the prediction labels by the models which are Critical, Major, Minor, and N. For each dataset RF model result shows better performance than DT, SVM, and KNN models.

6.4. Discussion

In this paper, we addressed defect severity level prediction for software modules. Some published studies addressed severity level prediction for bug reports. Our findings show that the severity level of defective software can be predicted using software defect metrics measurement of a software module.

The dataset we used for the study contains software defect metrics attributes with the class defective and non-defective. Since the objective of the study is to predict the severity level of defective software modules but in the existing dataset, we observed that the number of the non-defective class is much greater than the defective class of the dataset. To tackle this problem we used under-sampling and oversampling methods, we used a 60% ratio of oversampling and 80% of under-sapling to balance the classes, we decreased the ratio of oversampling which is duplicating the number of minority classes to 60% because as we have little examples in our minority class duplicating synthetic data can disturb the class and will result in over-fitting of the model so that we increased the ratio of under-sampling which is reducing of the majority classes as 80% of the minority class. So the result shows combining both methods allows balancing the bias in both classes.

The existing dataset has no severity information to label the defect data k means clustering algorithm is used. The algorithm segment defect data based on the values of the software metrics. The most difficult task in k means clustering is finding the number of k cluster

numbers; in this study, we used the elbow method using WCSS which measures distance variance between data points within a cluster. A cluster with a small WCSS shows lower variability between the points within a cluster, the cluster numbers for each dataset are identified where WSCC started to decrease which resulted in 3 clusters for each dataset. We observed that the variation in the density of the data in the dataset resulted in a different number of examples on each cluster also as shown in **Figure 6.27** for the KC1 dataset the resulted clusters are much overlapping because the nature of the data has close points and outliers.

When there are different size clusters in the data finding and identifying clusters is very challenging, as the objective is to label the dataset with severity information the resulted cluster labels are different in numbers of data to identify and assign severity information to each group of clusters. So we observed what kind of data each cluster has based on the features defect metrics. , large LOC metrics shows the number of lines within a module, large LOC with more than 20 lines will result in high defect as its difficult to maintain, also McCabe complexity metrics measure the complexity of a module's structure, modules with the highest complexity are highly defective and cannot be maintained easily, Higher cyclomatic complexities correlate with greater testing and maintenance requirements, Complexities above 20 also shows highly defect prone modules. The other Halstead metrics shows the difficulty in understanding and writing a program, the highest difficulty shows defect prone modules.

Based on the observation for the PC3 dataset as shown in **Table 6.1** the clustered group cluster 2 contains modules with large LOC metrics, highest values of McCabe complexity, and highest values of Hallstead difficulty. Cluster 1 contains software modules with the lowest values of the metrics and cluster 0 contains values between the two values, based on this insight from the dataset, cluster 2 group are named as critical, cluster 1 is named minor and cluster 0 is named major. For the KC1 dataset as shown in **Table 6.2** the clustered group cluster 0 contains modules with large LOC metrics, highest values of McCabe complexity, and highest values of Hallstead difficulty. Cluster 1 contains software modules with the lowest values of the metrics and cluster 2 contains values between the two values, cluster 0 groups are named as critical, cluster 1 is named minor and cluster 2 is named major. For the CM1 dataset as shown in **Table 6.3** the clustered group cluster 0 contains modules with large LOC

metrics, highest values of McCabe complexity, and highest values of Hallstead difficulty. Cluster 1 contains software modules with the lowest numbers of the metrics and cluster 2 contains values between the two values, cluster 0 groups are named as critical, cluster 1 is named minor and cluster 2 is named major. For the PC1 dataset as shown in **Table 6.4** the clustered group cluster 1 contains modules with large LOC metrics, highest values of McCabe complexity, and highest values of Hallstead difficulty. Cluster 2 contains software modules with the lowest values of the metrics and cluster 0 contains values between the two values, cluster 1 group are named as critical, cluster 2 is named minor and cluster 1 is named major. For the KC3 dataset as shown in **Table 6.5** The clustered group cluster 2 contains modules with large LOC metrics, highest values of McCabe complexity, and highest values of Hallstead difficulty. Cluster 0 contains software modules with the lowest values of the metrics and cluster 1 contains values between the two values, cluster 2 group are named as critical, cluster 0 is named minor and cluster 1 is named major.

So cluster group with large LOC, highest complexity and difficulty number are named as Critical as they are highly defective which shows high severe, cluster groups with the lowest value of the above metrics are named Minor shows low defect, the rest of clusters between this group are named Major which shows medium severe. Our findings show that software metrics can be used to identify severity levels of a given software module. Their value gives information on the most and least defective modules. Software module severity levels can be determined by using the software defect metrics.

This study experimented with four machine learning algorithms RF, DT, KNN, and SVM on five selected datasets CM1, KC1, KC3, PC1, and PC3. CM1 contains 792 total data with 37 metrics, KC1 contains 3559 total data with 21 metrics, KC3 contains 434 total data with 39 metrics, PC1 contains 1750 total data with 37 metrics, PC3 contains 2542 total data with 37 metrics.80% for training and 20% for testing. As shown in **Table 6.6**, **Table 6.7**, **Table 6.8,** and **Table 6.9** for the CM1 dataset the highest accuracy obtained is 96.5% by the RF model, and the lowest accuracy obtained is 63.5 % by the SVM model. For the KC1 dataset, the highest accuracy obtained is 90.1% by the RF model and the lowest accuracy is 57.1% by SVM. For KC3 the highest accuracy obtained is 94.5% by the RF model and the lowest accuracy obtained is 55.2% by the SVM model. For PC1 the highest accuracy obtained is

97.4% by the RF model and the lowest accuracy obtained is 73.4 % by the SVM model. For PC3 the highest accuracy obtained is 95.8% by the RF model and the lowest accuracy obtained is 64.8% by the SVM model. The maximum accuracy obtained by DT is 94.3 on the CM1 dataset and the lowest accuracy obtained is 83.8 on KC1. The maximum accuracy obtained by KNN is 89.3 % on the PC1 datasets and the lowest accuracy obtained is 77.1% on KC3. The maximum accuracy obtained by SVM is 73.4% on PC1 and the lowest accuracy obtained is 55.2% on KC3. The result shows that RF performed well and achieved the highest accuracy on each dataset, resulted in maximum accuracy of 97.4% on PC1 and minimum accuracy of 90.1% on KC1 datasets. It has also obtained the highest values in Recall, Precision, and F-measure, the highest f measure is obtained by the model is 97% on PC1 and 90% on KC1 datasets. DT algorithm also performed well but the performance RF was higher than DT on each dataset. RF model can handle large dimensions in our case we have more than 35 metrics on each dataset. The algorithm works by identifying important features so large dimensions cannot affect the performance of the algorithm. it has also a method for automatically balancing the dataset where there is a bias towards one class, that's why the algorithm outperforms in our case The classification performance of SVM is worst compared to other classifiers, as the dataset contains large software metrics SVM is not suitable when there are a large number of attributes because of the very high training complexity of SVM and also as the resulted cluster label is not equal there is a class imbalance between the cluster labels, when there are some overlapping classes, which means not well-separated SVM cannot perform well. While separating hyperplane the algorithm will be biased towards the minority class, this will decrease the performance of the algorithm.

CHAPTER SEVEN

7. Conclusion, Recommendation, and Future Work

In this chapter, the proposed Software Defect Severity level Prediction using machine learning and the finding of this research is concluded. It also described the recommendation and future research directions.

7.1. Conclusion

This research proposed to develop a solution for Software Defect Severity prediction using machine learning techniques. The study attempted to develop, implement, and compares machine learning methods for Software Defect Severity Prediction.

It was important to understand and define software defects, software metrics, and software categories to successfully execute the study, as discussed in chapter two. Also, methods used to implement and design for software defect severity level prediction; these methods include resampling to balance the classes of the dataset, clustering for grouping the defective classes to the appropriate severity level, and model training using RF, DT, SVM, and KNN and model testing and finally comparing model using evaluation metrics.

In this study, we used five NASA's MDP datasets namely CM1, KC1, KC3, PC1, and PC3. We resampled each dataset using a combination of oversampling and under-sampling techniques to balance the dataset. Each balanced defective class of dataset is clustered into severity level groups Critical, Major, and Minor. Those clusters are identified by using software metrics of a given software module on the dataset.

Based on these labels the model developed using RF, DT, SVM, and KNN. The experiment is performed on 5 selected datasets resulted in 4 models for each dataset. RF model resulted in better test accuracy on each dataset with a minimum test accuracy of 90.1% on KC1 and maximum test accuracy on 97.4% on PC1 datasets and also the f measure is between 97% on PC1 and 90% on KC1 as compared to DT, SVM, and KNN classifiers.

In general, the results of the study were very hopeful. It was possible to reduce biased classes of the existing dataset using sampling methods. It was possible to label the dataset with severity information for software modules using clustering machine learning techniques, software metrics can be used in identifying the severity level of a software module. RF models performed well on each dataset compared to the other models so using the RF model is recommendable in defect severity prediction.

7.2. Recommendation and Future Work

Since the prediction models have shown some capability of predicting the severity level of defective modules. As a result, the researcher recommends software developing companies to develop defect management models that assist the software quality assurance team to track defects that are detected during the testing phase.

The study proposed a solution using clustering and classification machine learning algorithm to build a model. It's better to see the difference in results using deep learning algorithms to build the models.

This study is limited only to the severity level of a defect, for a more inclusive prediction model, future research can focus on developing a model for some other categories such as priority and probability of a defect. Additionally, the study proposed software defect severity level prediction for software modules on the offline dataset is not addressed any automated system for online prediction software defects. Therefore the study can be conducted on real-time software defects.

Special Acknowledgement: This research was funded by Adama Science and Technology University under the grant number **ASTU/SM-R/093/19**.

Adama, Ethiopia.

References

[1] "How Many Software Developers are There in the World?" [Online]. Available: https://evansdata.com/press/viewRelease.php?pressID=278. [Accessed: 30-Nov-2020].

[2] L. Bergmane, J. Grabis, and E. Žeiris, "A Case Study: Software Defect Root Causes," *Inf. Technol. Manag. Sci.*, vol. 20, no. 1, pp. 54–57, 2018, doi: 10.1515/itms-2017-0009.

[3] S. and G. N. T. V, "Defect Management Startegies in Software Development," 2009.

[4] I. Arora, V. Tetarwal, and A. Saha, "Open issues in software defect prediction," *Procedia Comput. Sci.*, vol. 46, no. Icict 2014, pp. 906–912, 2015, doi: 10.1016/j.procs.2015.02.161.

[5] C. Jones, "Software Metrics," *Computer (Long. Beach. Calif).*, vol. 27, no. 9, pp. 98–100, 1994, doi: 10.1109/2.312055.

[6] J. Abbineni and O. Thalluri, "Software Defect Detection Using Machine Learning Techniques," *Proc. 2nd Int. Conf. Trends Electron. Informatics, ICOEI 2018*, no. Icoei, pp. 471–475, 2018, doi: 10.1109/ICOEI.2018.8553830.

[7] S. Reddivari and J. Raman, "Software quality prediction: An investigation based on machine learning," *Proc. - 2019 IEEE 20th Int. Conf. Inf. Reuse Integr. Data Sci. IRI 2019*, pp. 115–122, 2019, doi: 10.1109/IRI.2019.00030.

[8] A. Hammouri, M. Hammad, M. Alnabhan, and F. Alsarayrah, "Software Bug Prediction using machine learning approach," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 2, pp. 78–83, 2018, doi: 10.14569/IJACSA.2018.090212.

[9] ISDW Group and others, "1044-2009-IEEE Standard Classification for Software Anomalies," *IEEE, New York*, 2010.

[10] "Top Software Testing Failures of 2019 | Kualitee." [Online]. Available: https://www.kualitee.com/software-testing/top-software-testing-failures-of-2019/.

[Accessed: 02-Dec-2020].

[11] "What Is a Software Defect? - The Practical Guide to Defect Prevention." [Online]. Available: https://learning.oreilly.com/library/view/the-practical-guide/9780735622531/9780735622531_ch01lev1sec1.html. [Accessed: 01-Dec-2020].

[12] D. DeVolder, S. Ghazanshahi, and J. Zadeh, *Software testing and quality assurance*, vol. 1. 2008.

[13] I. Evans and R. Black, "FOUNDATIONS OF SOFTWARE TESTING."

[14] H. Wang, "Software Defects Classification Prediction Based On Mining Software Repository," 2014.

[15] "ISTQB Glossary." [Online]. Available: https://glossary.istqb.org/en/term/defect-3. [Accessed: 01-Dec-2020].

[16] E. E. Ogheneovo, "Software Dysfunction: Why Do Software Fail?," *J. Comput. Commun.*, vol. 02, no. 06, pp. 25–35, 2014, doi: 10.4236/jcc.2014.26004.

[17] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, B. K. Ray, and D. S. Moebus, "Orthogonal Defect Classification—A Concept for In-Process Measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992, doi: 10.1109/32.177364.

[18] S. Wagner, "Defect classification and defect types revisited," *DEFECTS'08 2008 Int. Symp. Softw. Test. Anal. - Proc. 2008 Work. Defects Large Softw. Syst. 2008, DEFECTS'08*, pp. 39–40, 2008, doi: 10.1145/1390817.1390829.

[19] "Defect Severity In Software Testing |Professionalqa.com." [Online]. Available: https://www.professionalqa.com/defect-severity-in-software-testing. [Accessed: 24-Feb-2020].

[20] IEEE -1061, "Software Quality Metrics Methodology," 1998.

[21] P. B. Nirpal, "A Brief Overview Of Software Testing Metrics," *Int. J. Adv. Res. Comput. Commun. Eng.*, vol. 3, no. 12, pp. 4655–4659, 2013.

[22] “Software Engineering Software Metrics - javatpoint.” .

[23] A. A. Aljadaa, “Software Metrics Birzeit University Faculty of Engineering and Information Technology Master of,” no. February, 2015, doi: 10.13140/2.1.4026.1923.

[24] “Code Metrics: Lines of Code (LoC).” [Online]. Available: https://gunnarpeipman.com/code-metrics-lines-of-code-loc/. [Accessed: 24-Feb-2020].

[25] H. Zhang, “An investigation of the relationships between lines of code and defects,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 274–283, 2009, doi: 10.1109/ICSM.2009.5306304.

[26] J. Mccabe, “A complexity Measure: Cyclomatic Complexity,” no. 4, pp. 308–320, 1976.

[27] “Mccabe’s Cyclomatic Complexity: Calculate with Flow Graph (Example).” [Online]. Available: https://www.guru99.com/cyclomatic-complexity.html. [Accessed: 11-Jan-2021].

[28] T. J. McCabe, “Structured Testing: a Software Testing Methodology Using the Cyclomatic Complexity Metric, 1982.,” *Natl. Bur. Stand. Spec. Publ.*, 1982.

[29] J. R, L. Florence, and A. Arya, “A Review on Software Defect Prediction Techniques Using Product Metrics,” *Int. J. Database Theory Appl.*, vol. 10, no. 1, pp. 163–174, 2017, doi: 10.14257/ijdta.2017.10.1.15.

[30] A. Abran, “Halstead’s Metrics: Analysis of Their Designs,” *Softw. Metrics Softw. Metrol.*, no. October 2010, pp. 145–159, 2010, doi: 10.1002/9780470606834.ch7.

[31] T. M. Mitchell, *Machine Learning*, vol. 17. 2011.

[32] Royal Society, *Machine learning : the power and promise of computers that learn by example*, vol. 66, no. January. 2017.

[33] “Machine Learning - Definition and application examples.” [Online]. Available: https://www.spotlightmetal.com/machine-learning--definition-and-application-

examples-a-746226/. [Accessed: 24-Feb-2020].

[34] T. Oladipupo, “Types of Machine Learning Algorithms,” *New Adv. Mach. Learn.*, no. August, 2010, doi: 10.5772/9385.

[35] “Types of Machine Learning Algorithms You Should Know.” [Online]. Available: https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861. [Accessed: 24-Feb-2020].

[36] M. Usama *et al.*, “Unsupervised Machine Learning for Networking: Techniques, Applications and Research Challenges,” *IEEE Access*, vol. 7, no. September 2017, pp. 65579–65615, 2019, doi: 10.1109/ACCESS.2019.2916648.

[37] “Types of Machine Learning Algorithms: Supervised and Unsupervised Learning | Netguru Blog on Machine Learning.” [Online]. Available: https://www.netguru.com/blog/types-of-machine-learning-algorithms-supervised-and-unsupervised-learning. [Accessed: 25-Feb-2020].

[38] P. Harrington, *Machine Learning in Action*, vol. 37, no. 3. 2012.

[39] D. Z. (California State, U. and J. J. P. T. (University, and U. of Illinois, Chicago, “Introduction to Machine Learning and Software Engineering,” no. Ml, 2005, pp. 1–36.

[40] Z. Li, X. Y. Jing, and X. Zhu, “Progress on approaches to software defect prediction,” *IET Softw.*, vol. 12, no. 3, pp. 161–175, 2018, doi: 10.1049/iet-sen.2017.0148.

[41] S. Pattnaik and B. K. Pattanayak, “A survey on machine learning techniques used for software quality prediction,” *Int. J. Reason. Intell. Syst.*, vol. 8, no. 1–2, pp. 3–14, 2016, doi: 10.1504/ijris.2016.080058.

[42] R. M. Arunima Jaiswal, “Software Reliability Prediction Using Machine Learning Techniques,” in *Proceedings of Fifth International Conference on Soft Computing for Problem Solving, Advances in Intelligent Systems*, 2016, vol. 436, pp. 105–116, doi: 10.1007/978-981-10-0448-3.

[43] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, “Systematic literature review of machine learning based software development effort estimation models,” *Inf. Softw. Technol.*, vol. 54, no. 1, pp. 41–59, 2012, doi: 10.1016/j.infsof.2011.09.002.

[44] E. N. Regolin, G. A. De Souza, A. R. T. Pozo, and S. R. Vergilio, “Exploring machine learning techniques for software size estimation,” *Proc. - Int. Conf. Chil. Comput. Sci. Soc. SCCC*, vol. 2003-Janua, pp. 130–136, 2003, doi: 10.1109/SCCC.2003.1245453.

[45] S. Maggo and C. Gupta, “A Machine Learning based Efficient Software Reusability Prediction Model for Java Based Object Oriented Software,” *Int. J. Inf. Technol. Comput. Sci.*, vol. 6, no. 2, pp. 1–13, 2014, doi: 10.5815/ijitcs.2014.02.01.

[46] S. Puranik, P. Deshpande, and K. Chandrasekaran, “A Novel Machine Learning Approach for Bug Prediction,” *Procedia Comput. Sci.*, vol. 93, no. September, pp. 924–930, 2016, doi: 10.1016/j.procs.2016.07.271.

[47] S. K. Pandey, R. B. Mishra, and A. K. Triphathi, “Software Bug Prediction Prototype Using Bayesian Network Classifier: A Comprehensive Model,” *Procedia Comput. Sci.*, vol. 132, no. Iccids, pp. 1412–1421, 2018, doi: 10.1016/j.procs.2018.05.071.

[48] Universitas Gadjah Mada. Program Pascasarjana, Institute of Electrical and Electronics Engineers, and I. International Annual Engineering Seminar (8th : 2018 : Yogyakarta, “Proceedings, the 12th SEATUC Symposium : 12-13 March 2018, the Graduate School of Universitas Gadjah Mada, Yogyakarta, Indonesia.,” 2018.

[49] A. Alsaeedi and M. Z. Khan, “Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study,” *J. Softw. Eng. Appl.*, vol. 12, no. 05, pp. 85–100, 2019, doi: 10.4236/jsea.2019.125007.

[50] T. Menzies and A. Marcus, “Automated severity assessment of software defect reports,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 346–355, 2008, doi: 10.1109/ICSM.2008.4658083.

[51] K. K. Chaturvedi and V. B. Singh, “Determining Bug severity using machine learning techniques,” *2012 CSI 6th Int. Conf. Softw. Eng. CONSEG 2012*, 2012, doi:

10.1109/CONSEG.2012.6349519.

[52] R. Malhotra, N. Kapoor, R. Jain, and S. Biyani, “Severity Assessment of Software Defect Reports using Text Classification,” *Int. J. Comput. Appl.*, vol. 83, no. 11, pp. 13–16, 2013, doi: 10.5120/14492-2622.

[53] R. Jindal, R. Malhotra, and A. Jain, “Prediction of defect severity by mining software project reports,” *Int. J. Syst. Assur. Eng. Manag.*, vol. 8, no. 2, pp. 334–351, 2017, doi: 10.1007/s13198-016-0438-y.

[54] G. Sharma, S. Sharma, and S. Gujral, “A Novel Way of Assessing Software Bug Severity Using Dictionary of Critical Terms,” *Procedia Comput. Sci.*, vol. 70, pp. 632–639, 2015, doi: 10.1016/j.procs.2015.10.059.

[55] A. Baarah, A. Aloqaily, Z. Salah, M. Zamzeer, and M. Sallam, “Machine learning approaches for predicting the severity level of software bug reports in closed source projects,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 8, pp. 285–294, 2019, doi: 10.14569/ijacsa.2019.0100836.

[56] “NASA IV &V Facility. Metric Data Program available on/GitHub - NASADefectDataset: NASA Defect Datasets.” [Online]. Available: https://github.com/klainfo/NASADefectDataset. [Accessed: 14-Nov-2020].

[57] O. Arbelaitz, I. Gurrutxaga, J. Muguerza, and J. M. Pérez, “Applying resampling methods for imbalanced datasets to not so imbalanced datasets,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8109 LNAI, pp. 111–120, 2013, doi: 10.1007/978-3-642-40643-0_12.

[58] R. Mohammed, J. Rawashdeh, and M. Abdullah, “Machine Learning with Oversampling and Undersampling Techniques : Overview Study and Experimental Results,” pp. 243–248, 2020, doi: 10.1109/ICICS49469.2020.239556.

[59] “Oversampling and Undersampling. A technique for Imbalanced… | by Kurtis Pykes | Towards Data Science.” [Online]. Available: https://towardsdatascience.com/oversampling-and-undersampling-5e2bbaf56dcf.

[Accessed: 22-Nov-2020].

[60] "Unsupervised Machine Learning: What is, Algorithms, Example." [Online]. Available: https://www.guru99.com/unsupervised-machine-learning.html. [Accessed: 25-Feb-2020].

[61] "The 5 Clustering Algorithms Data Scientists Need to Know." [Online]. Available: https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68. [Accessed: 26-Feb-2020].

[62] "K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks." [Online]. Available: https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a. [Accessed: 28-Feb-2020].

[63] J. Pérez-Ortega, N. Nely Almanza-Ortega, A. Vega-Villalobos, R. Pazos-Rangel, C. Zavala-Díaz, and A. Martínez-Rebollar, " The K -Means Algorithm Evolution ," *Introd. to Data Sci. Mach. Learn.*, 2020, doi: 10.5772/intechopen.85447.

[64] S. Aleem, L. F. Capretz, and F. Ahmed, "Benchmarking Machine Learning Techniques for Software Defect Detection," *Int. J. Softw. Eng. Appl.*, vol. 6, no. 3, pp. 11–23, 2015, doi: 10.5121/ijsea.2015.6302.

[65] J. Nam, *Survey on Software Defect Prediction*. 2014.

[66] J. Hurwitz and D. Kirsch, *Approaches to machine learning*, vol. 35, no. 5. 1984.

[67] M. Awad and R. Khanna, "Support Vector Machines for Classification," *Effic. Learn. Mach. Theor. Concepts, Appl. Eng. Syst. Des.*, no. January, pp. 1–248, 2015, doi: 10.1007/978-1-4302-5990-9.

[68] Z. Zhang, "Introduction to machine learning: K-nearest neighbors," *Ann. Transl. Med.*, vol. 4, no. 11, 2016, doi: 10.21037/atm.2016.03.37.

[69] A. Zheng, *Evaluating Machine Learning Algorithms*. 2015.

[70] “Confusion Matrix in Machine Learning - GeeksforGeeks.” [Online]. Available: https://www.geeksforgeeks.org/confusion-matrix-machine-learning/. [Accessed: 28-Feb-2020].

Appendix A: Sample Source Code

Appendix A.1: A sample code for sampling

```
import imblearn

from imblearn.under_sampling import RandomUnderSampler,RandomOverSampler

over = RandomOverSampler(sampling_strategy=0.6)

under = RandomUnderSampler(sampling_strategy=0.8)

X_over, y_over = over.fit_resample(X, y)

X_combined_sampling, y_combined_sampling = under.fit_resample(X_over, y_over)
```

Appendix A.2: A sample code for K-means clustering

```
From sklearn.cluster import KMeans

#finding elbow point

Wcss=[]

For i in range(1,11):

        kmeans=KMeans(n_clusters=i,  init='k-means++',  max_iter=500,  n_init=10,
        random_state=0)

      kmeans.fit(x)

  wcss.append(kmeans.inertia_)

#Plotting the results onto a line graph, allowing us to observe 'The elbow'

plt.plot(range(1, 11), wcss)

plt.title('the elbow method')
```

```
plt.xl(‘Number of clusters')

plt.ylabel('WCSS') #within cluster sum of square

plt.show()

kmeans=cluster.KMeans(n_clusters=3,init="k-means++",max_iter = 1000, n_init = 10, random_state = 0)

y_kmeans=kmeans.fit_predict(x)

data['Severity'] =kmeans.labels_

index= {0 : 'Critical', 1 : 'Minor', 2:'Major'}

data ['Severity'] = data .apply(lambda x: index.get(x['Severity'], 'Unknown'), axis=1)

data .head()
```

Appendix A.3 A Sample Code for Building and Evaluating Model

```
#RF

from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import GridSearchCV

param = {"n_estimators":[100, 120, 150,1000],

       "max_depth":[3, 5, 10],

     "min_samples_leaf":[3, 5, 10],

       "min_samples_split":[2, 5, 10]

       }

#instanciate random forest

clf = RandomForestClassifier()
```

```
#grid search
forest=GridSearchCV(clf, param)
#fit model to train the model
forest.fit(X_train,y_train)
y_pred=forest.predict(X_test)
forest.score(X_test,y_test) # test accuracy
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test,y_pred,normalize='pred')# normalized confusion matrix
from sklearn.metrics import classification_report
cr = classification_report(y_test,y_pred)# classification report
# DT
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
param = {"max_depth":range(1,10),
        "min_samples_leaf": range(1, 9),
        "criterion": ["gini", "entropy"]}
#initializing tree classifier
clf= DecisionTreeClassifier()
#grid search
DT = GridSearchCV(clf, param)
```

```
# fit the model to predict

DT.fit(X_train,y_train)

y_pred = DT.predict(X_test)

DT.score(X_test, y_test)

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test,y_pred,normalize='pred')# normalize confusion matrix

from sklearn.metrics import classification_report

cr = classification_report(y_test,y_pred)# classification report

#KNN

from sklearn.neighbors import KNeighborsClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import GridSearchCV

#instanciate the model

param={"leaf_size":range(1,10),

      "n_neighbors":range(1,10),

        "p":[1,2]

     }

clf = KNeighborsClassifier()

#gridseearch

clfK=GridSearchCV(clf, param)
```

```
#fit to train the model

clfK.fit(X_train, y_train)

y_pred = clfK.predict(X_test)

clfK.score(X_test,y_test) #test accuracy

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test,y_pred,normalize='pred')# normalize confusion matrix

from sklearn.metrics import classification_report

cr = classification_report(y_test,y_pred)# classification report

#SVM

from sklearn.svm import LinearSVC

from sklearn.multiclass import OneVsRestClassifier

#inistanciate the training model

svclassifier =LinearSVC(multi_class='ovr', max_iter=1000)

# fit the model to train

svclassifier.fit(X_train, y_train)

y_pred = svclassifier.predict(X_test)

svclassifier.score(X_test,y_test)

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test,y_pred,normalize='pred')# normalize confusion matrix
```