ADDIS ABABA UNIVERSITY
ADDIS ABABA INSTITUTE OF TECHNOLOGY
SCHOOL OF GRADUATE STUDIES
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Enhancing Just-in-Time Defect Prediction Using Change Request-based Metrics

A thesis submitted to the School of Graduate Studies of Addis Ababa University in partial fulfilment for the requirements for the Degree of Masters of Science in Computer Engineering

By

Hailemelekot Demtse

Advisor

Dr. Surafel Lemma

February 2021

Addis Ababa, Ethiopia

Statement of Authorship

I hereby declare that this research work is my own original work and all sources are fully acknowledged. At the same time, I affirm that this research work has never been used for other academic degrees.

Signature: _____

Date: _____

Certificate

This is to certify that this thesis work which is submitted by Hailemelkot Demtse is carried under my supervision and guidance and fulfilling the nature and standard required for the partial fulfilment of requirements of masters of science degree in Computer Engineering. The work embodied in this thesis has not been submitted elsewhere for degree.

_____

Dr. Surafel Lemma

School of Electrical and Computer Engineering

Addis Ababa Institute of Technology

Addis Ababa University

Date: _____

Addis Ababa University

Addis Ababa Institute of Technology

School of Electrical and Computer Engineering

The undersigned have examined the thesis titled:

Enhancing Just-in-Time Defect Prediction Using Change Request-based Metrics

Presented by Hailemelekot Demtse Tessema, a candidate for the degree of Master of Science and hereby certify that it is worthy of acceptance.

Approved By Board of Examiners

Dr. Yalemzewd Negash	_____
Dean, School of Electrical and Computer Engineering	Signature
Dr. Fitsum Asaminew	_____
Internal Examiner	Signature
Mr. Kinde Mekuria	_____
External Examiner	Signature
Dr. Surafel Lemma	_____
Advisor	Signature

February 2021

Addis Ababa, Ethiopia

Abstract

Identifying defective software components as early as their commit helps to reduce significant software development and maintenance costs. In recent years, several studies propose to use just-in-time (JIT) defect prediction techniques to identify changes that could introduce defects at check-in time. To predict defect introducing changes, JIT defect prediction approaches use change metrics collected from software repositories. These change metrics, however, capture code and code change related information. Information related to the change requests (e.g., clarity of change request and difficulty to implement the change) that could determine the change's proneness to introducing new defects are not studied. In this study, we propose to augment the publicly available change metrics dataset with six change request-based metrics collected from issue tracking systems. To build the prediction model, we used five machine learning algorithms: AdaBoost, XGBoost, Deep Neural Network, Random Forest and Logistic Regression. The proposed approach is evaluated using a dataset collected from four open source software systems, i.e., Eclipse platform, Eclipse JDT, Bugzilla and Mozilla. The results show that the augmented dataset improves the performance of JIT defect prediction in 19 out of 20 cases. F1-score of JIT defect prediction in the four systems is improved by an average of 4.8%, 3.4%, 1.7%, 1.1% and 1.1% while using AdaBoost, XGBoost, Deep Neural Network, Random Forest and Logistic Regression, respectively. Finally, among the five algorithms used for building the machine learning models, AdaBoost is found to be better algorithm for enhancing the performance of JIT defect prediction. To see which of the features contributed to the improvement of JIT defect prediction, we computed feature importance using the best performing algorithm, AdaBoost. The result shows that *number of comments (NC), Severity* and *number of developers assigned (NDA)* are among the top important features from the entire augmented dataset.

Keywords—Just-In-Time software defect prediction, Software Defect Prediction, Software Bugs, Issue Tracking Systems, Software Metrics

Acknowledgments

In the first, place I want to forward my sincere thanks to God and Saint Mary (“Eme Berhan”) for standing at my side regardless of different challenges and ups and downs. Next, I want to thank my advisor Dr. Surafel Lemma for his consistent guidance and supervision. I also thank Mr. Melaeke Serawit for his unwavering help while doing the second phase of data collection by devoting his time. I want also to forward my sincere thanks to my mother Abeba M., my father Demtse T., and my brother Hayleyesus D. for the priceless contribution they made to me in all aspects of my life.

At this incidence, I would like to forward my heartfelt gratitude to those who stood on the side of me and gave me their support during my early stage of education. Genet Mulugeta and Eleny Mulugeta, may God repay you and be with you in all aspects of your life. In addition, I would like to express my sincere gratitude to those of my elementary and high school teachers who encouraged me relentlessly to thrive in my education and gave me any academic support whenever I required it. In this regard, Meskerem H, Mentamir T, Getahun T, Belaynesh H, Endalkacew A and Tigist D. deserve this.

May God repay all of you for all the good things you did for me.

Contents

List of Tables

List of Figures

Abbreviations

Acc.	Accuracy
AdaBoost	Adaptive Boosting
ADAM	Adaptive Moment Estimation
AUC	Area Under the ROC Curve
BTS	Bug Tracking Systems
CPU	Central Processing Unit
CSV	Comma Separated Value
CVS	Code Version System
DNN	Deep Neural Network
Eclipse Plat.	Eclipse Platform
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
IDE	Integrated Development Environment
ITS	Issue Tracking System
JIT	*Just-In-Time*
LR	Logistic Regression
LOC	Number of Line of Code
NC	Number of Comments
N DevAInHis	Number of Developers Assigned During the History of the Bug
NLC	Number of Line of Code
Pre.	Precision
RAM	Random Access Memory
Rec.	Recall
ReLU	Rectified Linear Unit
RF	Random Forest
ROC	Receiver Operator Characteristics

RQ	Research Question
SDP	Software Defect Prediction
SMOTE	Synthetic Minority Oversampling Technique
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate
XGBoost	Extensive Gradient Boosting

Chapter 1

Introduction

Identifying software defects at early stage of development will reduce the effort and resources deployed to correct the defects after the software is released. One of the efficacious mechanisms to achieve this goal is to utilize software defect prediction. Defect prediction helps to identify defect prone components of a software system at different levels of granularity.

Just-in-time (JIT) defect prediction approaches identify probable defective changes at check-in time [1]. These approaches help developers to minimize the time they spend in identifying sources of new defects. To build a model and predict, several JIT-based approaches [2][3] usually use change metrics, in particular the one prepared by Kamei et al. [1]. These change metrics are collected from software repositories. The information captured in software repositories, however, is limited to code and change history. Change requests on the other hand have other related relevant information (e.g., clarity of change request, difficulty to implement the change) that could be used to determine the change's proneness to introduce new defects. We conjuncture that augmenting the change metrics with such information about the change requests will enhance the performance of JIT defect prediction approaches.

Information related to a change request is usually captured in issue tracking systems. Issue tracking systems are used to request, discuss, prioritize and track changes. They serve as the main platform for communication among developers themselves and users of the software system. In this research, we propose to augment the existing change metrics collected from software repositories with metrics collected from issue tracking systems. In particular, we identified six metrics from issue tracking systems to augment the change-based metrics dataset defined by Kamei et al. [1]. The identified metrics are *severity*, *priory*, *time taken to complete the change*, *number of comments*, *depth of discussion* and *number of developers assigned*.

To assess the impact of the new metrics, we conducted an experiment using four open source software systems: Eclipse platform, Eclipse JDT, Bugzilla and Mozilla. To build the prediction model, we employed five commonly used machine learning algorithms: AdaBoost, XGBoost, Deep Neural Network, Random Forest and Logistic Regression. The augmented dataset is used to build a JIT defect prediction model and is compared with the baseline model built using the dataset prepared

by Kamei et al. [1]. The results show that the augmented dataset improves the performance of JIT defect prediction in 19 out of 20 cases. F1-score of JIT defect prediction in the four systems is improved by an average of 4.8%, 3.4%, 1.7%, 1.1% and 1.1% while using AdaBoost, XGBoost, Deep Neural Network, Random Forest and Logistic Regression, respectively. Besides, we also compare the contribution level of each feature for both the existing change metrics and the new change request-based metrics by computing the feature importance using AdaBoost, XGBosst and Random Forest algorithms.

1.1 Motivation

In 2021, it is expected that 36 billion devices will be connected through the internet and 54% of our global community will be online using the aforementioned devices [4]. This vivid fact demonstrates how software systems become influential and important for the day to day life of human beings. The aforementioned fact not only dictates this but also shows the undesirable consequences of software defects once they are introduced into these complex software systems. In this regard, software development is an arduous task that requires a significant amount of effort and economy [4].

It is clear that software defects are the results of unexpected negative consequences which occur when a particular product is not able to satisfy the expectation and requirements of its users [5]. Regardless of their cause, software defects might have very severe consequences. They cost capital, time, energy even the life of human beings [5][6][7].

Nowadays, regardless of the software development companies' experience, the failure rate of software systems is still at a rising slope. In 2018, the global cost of software development was more than 4 trillion dollars. In the same year, the cost of poor quality software in the USA only was about 2.84 trillion dollars [4].

As it is said before, the impact of software defects is multidimensional, complicated and vast. They can cause loss of human life, they cost a huge amount of economy and they affect the reputation of companies. The following examples demonstrate the tragic consequences of software defects:

1. The unit conversion system bug in a software system makes NASA lose a climate orbiter rocket that costs 125,000,000 dollars [4][7].
2. A numeric overflow error crashed the Ariane rocket which costs 8,000,000,000 dollars [4].
3. Russia faces the largest non-nuclear explosion which is resulted from software defects [4].

4. Software defect (numerical round-off error) costs the government of USA the life of its soldiers and its military infrastructure because of the failure of its patriot missel during the first gulf war [8][7].

In order to avoid or minimize the unfavorable consequences of software defects, deploying quality assurance mechanisms and processes have a vital role and paramount importance. Such practices are vital to ensure the quality of the software systems—making the software systems reliable and free from different types of errors. However, software quality assurance mechanisms and procedures are not easy to implement and use as they require a huge amount of resources (skilled manpower, economy, and time). Due to this fact, virtually it is impossible to create defect-free software systems except for very high mission and life-critical systems. Therefore, it is possible to say that bugs are inevitable threats in a software system and once they are introduced it is highly costly to find/locate them.

In addition, when errors are identified after the software is released, it costs a lot of time, effort, and finance [9][10]. Due to this fact, it is not an easy task to provide adequate and efficacious solutions for defects which are occurred after the software is released.

Therefore, aside from the usual software testing and other quality assurance activities, it is essential to devise efficient mechanism that tells about the probable defective components of a software system at commit time.

1.2 Statement of the Problem

In order to minimize the negative consequences of software defects, enormous research works have been conducted on software quality assurance and software defect prediction. These research works include predicting bugs, localizing bugs [11], etc. by examining the data found in different software repositories like Git-hub and bug tracking systems like Bugzilla. Despite these plethoras of research works, no research work is reported till now that tries to examine change request-based information like the time to fix the bug, severity of the bug, the priority of the bug, the number of idea exchange in the form of discussion, and the actual size of discussion in the form of word count for resolving the reported bug. By considering the above issues, this research work aimed to empirically examine the impact of change request-based metrics on *just-in-time* software defect prediction. After taking the above facts under consideration, the entire research is based on the following rationales to compute

our change request-based metrics from software bug tracking systems:

- If the fixing time is long, the bug is either new/peculiar or complex—so, efforts made to fix such bugs might induce another bug.
- Higher priority bugs are critical to the system and require tight time to fix. Any resolution effort for such defects might lead to another bug.
- The severity of the bug might have the capability to tell about future defective commits.
- If the number of idea exchanges in the form of comments is large, it indicates that the bug requires due attention. This might lead to the conclusion that the bug is either new or it is complex which requires arduous effort. A resolution effort made for such bugs might lead to another bug.
- Aside from the number of idea exchanges, which is explained above, the depth of discussion is determined by the total number of words used for expressing bug-related issues assumed to portray the nature of the bug. Therefore, committing code to resolve such issues might induce another bug.

Based on the aforementioned assumptions, this research work tries to examine the impact of the above-proposed parameters on just-in-time software defect prediction. To achieve our objective we aim to conduct empirical experiments after augmenting the new metrics with the publicly available data-set which is prepared for *just-in-time* software defect prediction by Kamei et al [1].

Therefore, at completion, this research work tries to answer the following research questions.

1. To what extent the change request-based metrics improve the prediction performance of JIT defect prediction scheme?
2. With regard to JIT defect prediction scheme, what are the important features in the existing change metrics and the new change request-based metrics?

1.3 Objectives

1.3.1 General Objective

The general objective of this research is to examine the impact of change request-based metrics towards *just-in-time* software defect prediction.

1.3.2 Specific Objectives

- To extract issue tracking-based information to define new metrics and augment the previously created *just-in-time* defect prediction dataset.
- To evaluate the impacts of the newly defined change request-based metrics on JIT defect prediction.
- To examine the importance of each feature in JIT defect prediction approach.
- To compare performance of five machine learning algorithms in JIT-based defect prediction system.

1.4 Methodology

1.4.1 Literature Review

Since bug prediction is a hot research issue in software engineering and software quality assurance disciplines, a wide and in-depth survey of literature will be conducted to examine the previously done research works and to have a due understanding of future research direction. The literature survey will examine different aspects of previously conducted experiments like the methodology used, the algorithm applied, the performance achieved and the gaps left for prospective research activities.

1.4.2 Data Collection

The data for JIT defect prediction are collected from software repositories and issue tracking systems. The software repositories are used to collect change metrics. The change metrics used in this research

are defined by Kamei et al. [1], for use in JIT software defect prediction. Kamei et al. [1] identified fourteen change metrics categorized into five dimensions, i.e., diffusion, size, purpose, history and experience. Table 4.2 presents a brief summary of the change metrics.

The issue tracking systems are used to collect the six change request-based metrics identified in this study. The six change request metrics are computed following the description in section 4.1. To identify the change request from ITS, we used the change request id in the defect inducing commit messages. Developers usually insert change request id in the message of a commit corresponding to a change request. The defect inducing commit is identified by matching the commit timestamp in the change based-metrics record with the commit timestamp in the corresponding software repository. The extracted six change request-based metrics are then augmented to the change metrics, creating a total of 20 metrics.

1.4.3 Design and Implementation

As a review of the literature suggests, aside from the existing metrics which are prepared by Kamei et al. [1], additional prediction metrics shall be examined to figure out their impact on different performance evaluation metrics of *just-in-time* software defect prediction scheme. Hence, those parameters are extracted from the Bugzilla bug tracking repository using web scrapping method. For scrapping the required information for the dataset python programming language is used. For the purpose of building, training and testing the machine learning models the cosponsoring python libraries and other third-party libraries are used.

1.4.4 Results and Evaluation

In order to evaluate the performance of our approach, we used commonly used evaluation metrics, i.e., precision, recall, F-measure, accuracy and false-positive rate, which are derived from a standard confusion matrix.

1.5 Scope

This thesis work is limited to analyzing the impact of the newly introduced change request-based metrics which are presented in section 4.1.

In general, the scope of this thesis can be summarized by the following core points:

- Preparing the dataset for the newly proposed defect prediction parameters to augment the existing dataset.
- Examining whether the newly identified defect prediction metrics have an impact on *just-in-time (JIT)* software prediction.
- Examining the level of impact of the newly introduced change request-based metrics on the previously mentioned defect prediction scheme.
- Examining source code based parameters for *just-in-time* defect prediction is not in the scope of this research work.

1.6 Significance of the Study

Software defects are potentially challenging threats for the prevailing technology-dependent contemporary era. As the unwanted consequences of software defects are multidimensional and critical by nature, it is necessary to give due attention, careful examination, allocate huge amount of time and enormous budget to minimize their presence in software systems especially in large software systems. Due to the aforementioned challenges, it is difficult, if not impossible, to conduct full-fledged software quality assurance mechanisms to either eliminate or minimize the negative consequences of bugs in software systems. As a result, deploying different software defect prediction techniques will be helpful not only to minimize the number of defects in a software system but also it saves a reasonable amount of resources that might be deployed in pre-release and post-release time of software systems. Therefore, this study put adequate effort to enhance the state of the art *just-in-time* software defect prediction scheme, by using a new change request-based metrics which are extracted from the issue tracking systems. It is our conjecture that this research will be beneficial for both the academics and the software industry by providing the following key benefits.

- The research plays important role to identify and examine the impacts of the new change request-based defect prediction metrics which are collected from the issue tracking system. This will be beneficial to design and implement a more robust and accurate defect prediction system that solves the aforementioned real-world problem.

- It provides a research output for scholars who are interested in *just-in-time (JIT)* and other related defect prediction schemes. In this regard, this work can be used as one significant piece of literature for other related researches in the future.
- If the approaches used in this research are used for real-world applications, it will be beneficial to reduce the disastrous consequences of defects in software systems.
- Provide a new way of research direction towards enhancing the performance of *just-in-time* software defect prediction schemes.

1.7 Contributions

A plethora of research works has been conducted about bug prediction in the last five decades. Each research works use different approaches to predict software bugs. The first research work regarding bug prediction was done in 1971 by Akiyama [12]. This research work uses the size of code (NLC) as an independent variable for predicting bugs. After the aforementioned work, an enormous number of researches has been done using different approaches and methods. Among these research works some of them done at coarse-grained level which involve prediction at file, package, or class level [13][14]. Others use fine-grained approaches [15][16] and other research works use change level metrics for predicting future bugs [17][18][19]. In addition to the above approaches, a plethora of research works even have been done using other bug prediction matrices such as code metrics, complexity metrics [20], object-oriented metrics [16][21][22], and code smells [23][24][25].

As it is mentioned above, even if the previously done research works are enormous in terms of quantity and the subject matter they try to address, none of them, actually to the best of our knowledge, tried to exploit the information found in the issue tracking systems and asses their impact on *just-in-time* defect prediction schemes. By taking the above-mentioned fact into account, this research work targeted to investigate the impact of issue tracking related information by exploiting the data which is found in issue tracking systems (bug tracking systems). Therefore, the main contributions of this research work are as follows:

- Creating a dataset for the newly proposed issue tracking based bug prediction parameters/-metrics for further research.
- Figuring out the impact of these parameters on the existing open source-based software systems.

- Giving a new direction for the subsequent research works to further analyze the data which are found in bug tracking systems.

1.8 Organization of the Thesis

The thesis consists of seven chapters: The first chapter deals with the motivation, the general and specific objectives, the scope, and the contribution of the thesis. The next chapter presents the details of theoretical backgrounds of different machine learning algorithms, code version systems and bug tracking systems. The third chapter makes a detailed investigation of previously done related works of literature. The fourth chapter explains the proposed approach. The fifth chapter presents the experiment setups of the entire thesis which includes: the algorithms hyperparameter configuration, the evaluation metrics used and the validation techniques. The sixth chapter gives a detailed explanation of the results obtained from the conducted sets of experiments. The final chapter concludes the thesis and indicates future research directions.

Chapter 2

Theoretical Background

This chapter is dedicated to provide the theoretical backgrounds of some core components of this research work. It delves into the details of each machine learning algorithm. These includes, their working principle, their mathematical background and the core components that made up these machine learning algorithms. In addition, this section is dedicated to provide a brief introductions about software bug repositories (BTS or ITS) and code version systems (CVS).

2.1 Software and Bug Repositories

2.1.1 Bug Tracking Systems

Issue tracking systems are repositories to keep track of bugs. In other words, they are huge databases to keep bug reports and to assign appropriate developer/s to a typical bug report. Among the common bug tracking systems Bugzilla, Jirra and the bug repository of POSTGRE SQL can be mentioned as an example.

The report accepting format and the data stored in the issue tracking systems might have a difference in different bug repositories however since the focus of this thesis is Bugzilla but tracking system, our focus will be completely on it.

Bugzilla accepts two kinds of reports. The first one is reported bugs and the second one is enhancement requests for software systems. In open source development communities it will be used for suggesting changes, assigning tasks, discussing enhancement solutions, to prioritize tasks, to follow-up dependencies and scheduling etc.

In addition, even if the customization of the system in different open-source projects shows slight differences, it gives a lot of information about the bug. By exploiting the information which is stored in the Bugzilla database it is possible to get information about the priority and the severity of the bug, the time required to fix the defect, the eccentric nature or the difficulty of the bug etc.

Reported bugs to Bugzilla issue tracking systems have different life cycles from the time they are

reported to the time they are completely fixed. The stratified life cycles of reported bugs and enhancement requests in the Bugzilla issue tracking system is briefly discussed below:

- UNCONFIRMED—when a newly reported either bug or enhancement requests come they are automatically put under this category. Till the quality assurance personnel gives confirmation, it stays under this category.
- NEW—After the bugs are confirmed by appropriate staff, the unconfirmed bugs will be moved to this category. When the bugs are at this stage, they might be fixed by the staff who look at them or they may be transferred to another developer.
- RESOLVED—when a bug is fixed, it will be moved to the resolved category and its resolution will state will be marked by FIXED.
- DUPLICATE—when a reported bug is reported again, it will fall under this category.
- WORKSFORME—when all efforts at reproducing in current build is unsuccessful, the bug will for under this category.

Figure 2.1 (which is adapted from http://www.bugzilla.com) shows the detailed lifecycles of bugs in Bugzilla defect tracking systems.

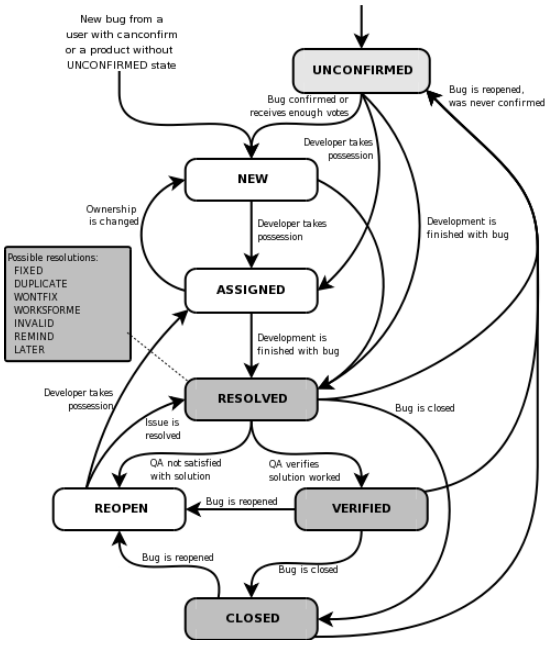

Figure 2.1: Bug life cycle in Bugzilla issue tracking system

2.1.2 Code Version Systems (CVS)

Code version systems are large databases that store different versions of textual files by storing the differences between successive revisions of program source code with associated metadata (the id of the committer, the date, the textual block of the user comment, etc.).

In open-source projects, they are used to store every file regarding the software system under development. By exploiting the information in the CVS systems, the entire history of the file revision can be retrieved. Among the common code version systems, Github is the one which is used by several open-source software projects.

2.2 Different Machine Learning Algorithms

2.2.1 Deep Learning Algorithm

Artificial neural networks are one of the well-known machine learning methods that simulate the learning principles of the brain of organisms [26]. Both of them possess neurons (which serve as a computation unit) and a connection that joins the neurons together. Artificial neural network calculates the function of the inputs by propagating the calculated values from the input neurons to the output neurons by using the weights as intermediary scaling factors.

The one trait that makes neural networks robust comes from the aggregation of the basic unit of classical machine learning units, like least square regression, together in such a way that they learn the weights of several units in order to minimize the prediction error [26]. One thing that requires a due consideration is if deep learners and neural networks are not looked at entirely as one unit, the learning algorithm of individual units or subunits is not different from the classical machine learning algorithms. When the size of the computational units joined together increases, the learning power of neural network systems is also increases and this makes them very powerful computational systems that are capable of learning very complex functions. As shown in Figure 2.2, neural networks consists of the following basic parts:

- Input layer neurons—that are directly interacted with the external input of the neural network.
- Hidden layer neurons—the internal computational units of the neural network systems. Neural networks may have 0 or more hidden layers however in order to be classified as a deep neural

networks they must have more than one hidden layer.

- Output layer Neurons—which provide the computational results of the neural network system to the external world.

The neurons at one layer will be connected to another layer by a certain interconnection mechanism which is dependent on the architecture of the neural network. For instance, in the case of a fully connected deep neural network structure, each layer n neuron is expected to be connected to all n+1 layer neurons and at the same time to all neurons at n-1 layers. However, it shall be noted that input layer neurons are not connected to n-1 layers as they are the first layer themselves, and output layers are not connected to another n+1 layer as they are the end layers.

In neural networks, the weights that are assigned for each inter connection of neurons will determine the output of that particular neuron. The propagation of the computed results of the n^{th} layer neuron to $\{n+1\}^{th}$ layer neuron is determined by the value of the weights. The entire training process of neural networks involves modifying the weights of the neural networks by comparing the actual sample with the computed result.

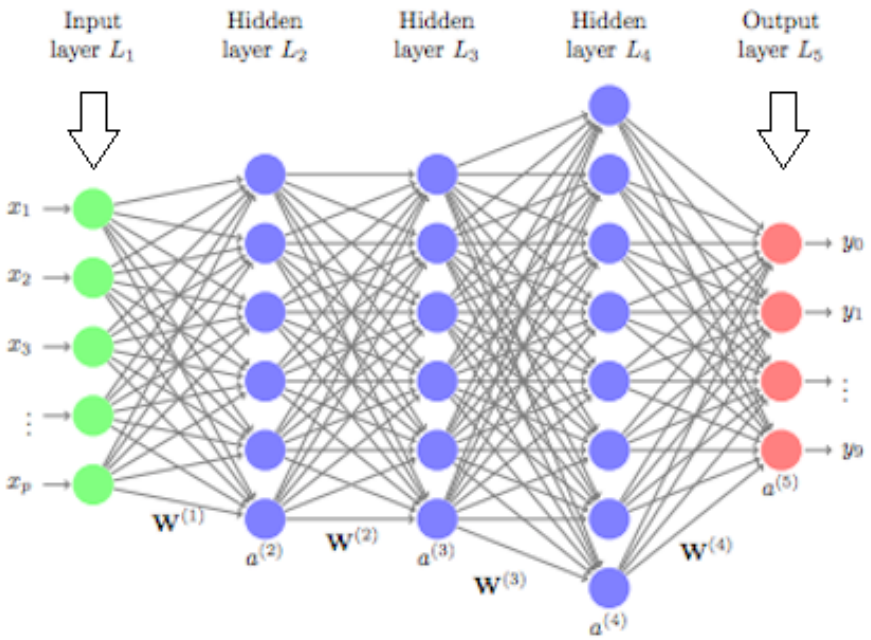

Figure 2.2: A typical fully connected DNN structure

In order to be useful for solving problems, neural networks shall be trained and tested. For this purpose, after the design process is completed, appropriate data shall be feed to them for training and testing. In order to train neural networks, sample data that consists of sample input and output

data will be provided. In addition, the desired accuracy level that the system required to achieve is also provided.

After the required data are feed to them, during the training stage the neural network system adjusts the weight of each interconnection by computing the mean square error of the computed output and the actual output. This process is continued several times iteratively till the system reaches the desired accuracy level. The flowchart shown in the Figure 2.3 clearly demonstrates the basic stage of the neural network system development from design to deployment.

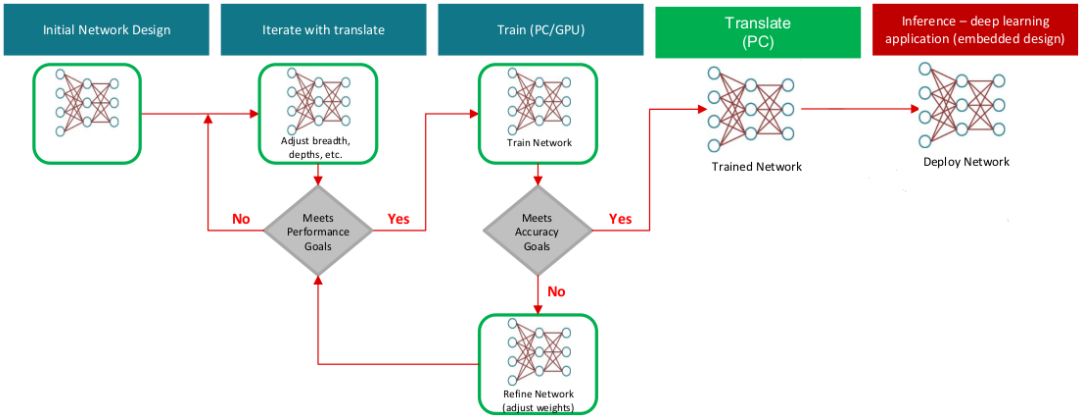

Figure 2.3: DNN working principle flowchart

The following section presents a brief highlight of the mathematical implementation of the neural networks systems.

Let the network consists of D input features $\{x_1, x_2, \ldots, x_D\}$

M hidden unit activations

$$a_j = \sum_{i=1}^{d} w_{ji}^{(1)} x_i + w_{j0}^{1} \quad where \quad j = 1, \ldots, M \tag{2.1}$$

Hidden unit activation functions

$$z_j = h(a_j) \tag{2.2}$$

K output activations

$$a_k = \sum_{i=1}^{d} w_{ki}^{(1)} x_i + w_{k0}^{1} \quad where \quad k = 1, \ldots, K \tag{2.3}$$

Output activation functions

$$y_k = \sigma(a_k) \tag{2.4}$$

$$y_k(x, w) = \sigma(\sum_{j=1}^{M} w_{kj}^{(2)} h(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}) + w_{k0}^{(2)}) \tag{2.5}$$

Calculation of loss and regularization

$$y = f(x, w) \tag{2.6}$$

$$E = \frac{1}{N} \sum_{i=1}^{N} E_i(f(x^{(i)}, w), t_i) \tag{2.7}$$

In order to update weights w form a labeled training dataset, DNN algorithm applies the gradient descent method which involves two steps—evaluate the derivatives of the loss($\nabla E(w)$ with respect to weights $w_1, w_2, \ldots w_T$ and use the derivative vectors to modify the value of weights at each iteration.

$$w^{(n+1)} = w^{(n)} - \eta \nabla E(w^{(n)}) \quad where \quad \nabla E(w) = [\frac{\partial E}{\partial w_0}, \quad \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}]' \tag{2.8}$$

In the above equation, η is the learning rate of the neural network.

2.2.2 AdaBoost Algorithm

The concept of boosting in machine learning is the mechanism of creating a robust and accurate prediction rule by combining many weak and inaccurate rules [27]. Boosting can also be viewed as the way of enhancing the selection quality and target of a machine learning system in areas where the system is not performing well [28]. It is a highly efficacious and prevalent approach in machine learning [29].

Adaptive Boost (AdaBoost) algorithm is one kind of algorithm that uses the general concept of boosting and ensemble method. It involves training and deploying trees in series. Since this algorithm uses the concept of boosting, it uses a set of weak classifiers that are connected in series. Each weak classifier in a series tries to improve the misclassified samples by the previous weak classifiers. Each

tree is trained to concentrate its attention on the weakness of only the previous tree [30]. While doing the aforementioned tasks, the entire system becomes better and better or the classification power of the system is boosted adaptively and step by step. In this sense, adaptive boosting involves coming up with a strong classifier by fusing a set of weak classifiers [31]. The collection of weak learners are assigned with a collection of weights over training data which are adaptively updated after each training cycle. The process of updating weights is done by evaluating the previous cycle classification results—the weights of the samples which are classified incorrectly by the weak learners get more weight (their weights are increased) while the weights of the samples which are classified properly by the weak learners get less weight (the weights are decreased). The working principle of AdaBosst algorithm is shown in the Figure 2.4.

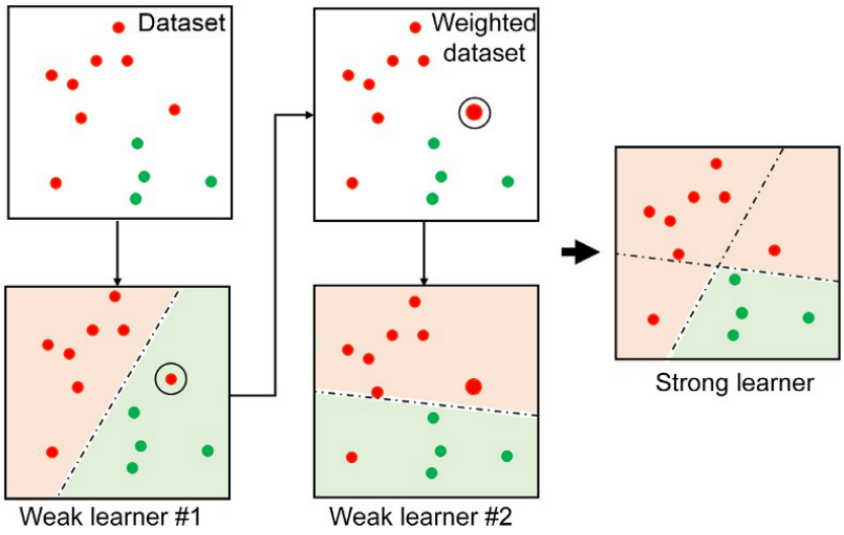

Figure 2.4: The working principle of AdaBoost algorithm [30]

A short explanation about the mathematical model that realize the concept of AdaBoost algorithm is presented below.

Let the weight distribution of on training i on round t is denoted by $D_t(i)$

The main function of the weak learners to find a weak hypothesis $h_t :\rightarrow \{$ -1, 1$\}$

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \epsilon X, y_i \epsilon Y = \{$ -1, 1$\}$

Initialize:

$$D_1(i) = \frac{1}{m} \quad for \quad i = 1, ..., m \tag{2.9}$$

(2.10)

for $t = 1, \ldots, T$:

Train weak learners using using distribution D_t

Get weak hypothesis (h_t)

$$h_t : X \rightarrow \{-1, 1\} \tag{2.11}$$

With error (E_t)

$$E_t = Pr_{i \sim D_t} \left[h_t \left(x_i \right) \neq y_i \right] \tag{2.12}$$

Choose

$$\alpha_t = \frac{1}{2} ln \frac{1 - E_t}{E_t} \tag{2.13}$$

Update:

$$D_{t+1} \left(i \right) = \frac{D_t \left(i \right)}{z_t} \times \left\{ \begin{array}{ll} e^{-\alpha t}, & \text{if } h_t \left(x_i \right) = y_i \\ e^{\alpha t}, & \text{if } h_t \left(x_i \right) \neq y_i \end{array} \right\} \tag{2.14}$$

$$= \frac{D_t \left(i \right) e^{-\alpha y_t h_t (x_i)}}{Z_t} \tag{2.15}$$

Where Z_t is a normalization factor

The output of the final hypothesis is given by:

$$H \left(x \right) = sign \left(\sum_{i=1}^{T} \alpha_t h_t \left(x \right) \right) \tag{2.16}$$

2.2.3 XGBoost Algorithm

XGBoost is the enhanced form of gradient boosting algorithm that incorporates several systems and algorithms optimization which makes it a scalable algorithm and, in addition to this, it has a very fast computation speed that runs 10 times faster than other similar algorithms [29]. One trait that

make this algorithm faster than another boosting algorithm is that the weak learners are added in a multithreaded and parallel manner which is essential to use beneficently all the CPU cores of the computing system in a parallel manner [32].

The mathematical implementation of the XGBoost algorithm is presented as follows:

Let $F = \{f_1, f_2, f_3, f_4, \ldots, f_n\}$ be set of base learners

The final prediction output be $\hat{y} = \sum_{t=1}^{m} f_t(x_i)$

$D = \{x_1, x_2, x_3, \ldots, x_m\}$ be data points

$L^{<t>} = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{<t-1>} + f_t(x_i)) + \Omega(f_t)$

If we see the famous Taylor series expansion $f(a+h) = f(a) + f'(a)h + \ldots + \frac{f^{(n)}(a)h^n}{h!}$

By taking the analogues components of the above two equations, we get:

$a = \hat{y}_i^{<t-1>}$

$h = f_t(x_i)$

$f(a) = l(y_i, \hat{y}_i^{<t-1>})$

$$L(t) = \sum_{i=1}^{n} l(y_i, \hat{y}^{<t-1>}) + \left(\frac{\partial l(y_i, \hat{y}_i^{<t-1>})}{\partial \hat{y}_i^{<t-1>}}\right) f_t(x_i) + \left(\frac{\partial^2 l(y_i, \hat{y}_i^{<t-1>})}{\partial \hat{y}_i^{<t-1>2}}\right) f_t(x_i)^2 + \ldots \tag{2.17}$$

But, we can see that $l(y_i, \hat{y}_i^{<t-1>})$ is a constant

Therefore,

$$L^{<t>} = \sum_{i=1}^{n} (c + g_i(f_t(x_i) + h_i f_t(x_i)) + \Omega(f_t) \tag{2.18}$$

Let f_t has K leaf nodes, i_j be the set of instances belonging to node j. w_j be the prediction for node j.

$$\Omega(f_t) = \gamma K + \frac{1}{2}\lambda \sum_{j=1}^{k} W_j^2 \tag{2.19}$$

$$L^{<t>} = \sum_{j=1}^{k} [\sum_{(i \epsilon I_j} g_i) w_j + \frac{1}{2} (\sum_{(i \epsilon I_j} h_i + \lambda) w_j^2] + \delta k \tag{2.20}$$

Then, for each leaf j,

$$\frac{dL^{<t>}}{dw_j^*} = 0 \tag{2.21}$$

$$0 = \sum_{I \epsilon I_j} g_i + \frac{1}{2}(\sum_{i \epsilon I_j} h_i + \lambda) \times 2w_j^* \tag{2.22}$$

This gives,

$$w_j^* = \frac{-\sum_{i \epsilon I_j} g_i}{\sum_{i \epsilon I_j} g_i + \lambda} \tag{2.23}$$

substituting the weight in the above equation gives the equation of the best loss.

$$L^{<t>} = -\frac{1}{2}\sum_{j=1}^{k} \frac{(\sum_{i \epsilon I_j} g_i)^2}{\sum_{i \epsilon I_j} h_i + \lambda} + \gamma k \tag{2.24}$$

If we think of on node (I) slitted into two leafs node of left (I_l) and one right node (I_r), the loss at parent becomes,

$$-\frac{1}{2}\frac{(\sum_{i \epsilon I} g_i)^2}{\sum_{i \epsilon I} h_i + \lambda} + \gamma \times 1 \tag{2.25}$$

and the loss at the left node becomes,

$$-\frac{1}{2}\frac{(\sum_{i \epsilon I_l} g_i)^2}{\sum_{i \epsilon I_l} h_i + \lambda} + \gamma \times 1 \tag{2.26}$$

and the loss of at the right node becomes,

$$-\frac{1}{2}\frac{(\sum_{i \epsilon I_r} g_i)^2}{\sum_{i \epsilon I_r} h_i + \lambda} + \gamma \times 1 \tag{2.27}$$

$$L_{split} = -\frac{1}{2}\frac{(\sum_{i \epsilon I} g_i)^2}{\sum_{i \epsilon I} h_i + \lambda} + \gamma - [-\frac{1}{2}\frac{(\sum_{i \epsilon I_l} g_i)^2}{\sum_{i \epsilon I_l} h_i + \lambda} + \gamma + -\frac{1}{2}\frac{(\sum_{i \epsilon I_r} g_i)^2}{\sum_{i \epsilon I_r} h_i + \lambda} + \gamma] \tag{2.28}$$

Finally, the loss becomes

$$L_{split} = \frac{1}{2}[\frac{(\sum_{i\epsilon I_l} g_i)^2}{\sum_{i\epsilon I_l} h_i + \lambda} + \frac{(\sum_{i\epsilon I_r} g_i)^2}{\sum_{i\epsilon I_r} h_i + \lambda} - \frac{(\sum_{i\epsilon I} g_i)^2}{\sum_{i\epsilon I} h_i + \lambda}] - \gamma \tag{2.29}$$

2.2.4 Random Forest Algorithm

Random Forest is algorithm that uses the concept of ensemble method for solving problems of classification, regression and feature selection. This algorithm deploys the concept of bagging to train several decision trees in parallel for carrying out the previously mentioned basic tasks. Bagging involves two basic processes which are done sequentially—bootstrapping and aggregation. During the stage of bootstrapping, several decision trees are trained using different subsets of the training dataset that consists of different subsets of features which are available in the training dataset. This makes the individual decision trees in a random forest are unique. Then the aggregation process will follow which aggregates the individual decision trees for the final decision. The final decision of the random forest algorithm will be decided either by majority voting which involves taking the result which is selected by the majority individual decision trees (for the case of classification problems) as a final result or through averaging which takes the mean value of the individual decision trees (for the case of regression problems) output as a final decision [30]. The basic working principles of the RF algorithm is shown in the Figure 2.5. The subsequent section is dedicated to give a brief highlight about the mathematical implementation of the Random Forest algorithm.

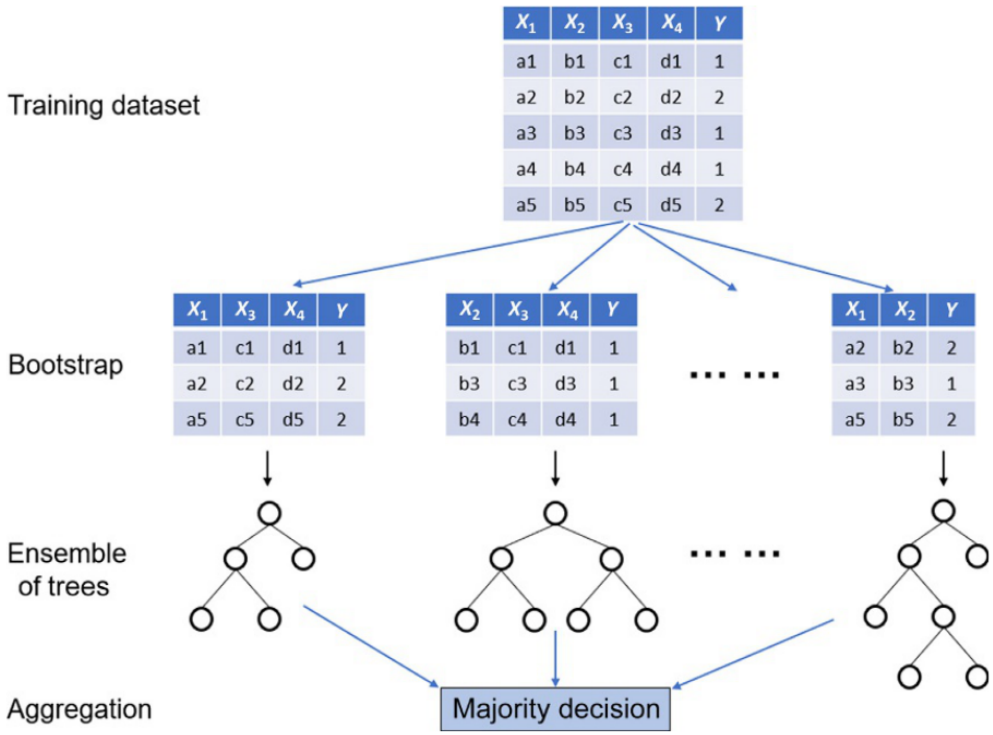

Figure 2.5: The working principle of Random Forest algorithm [30]

Random forest classifier is an ensemble algorithm which consists of tree structured classifiers $\{h(x, \Theta_k, k = 1, 2, 3, \ldots)\}$ where $\{\Theta_k\}$ independent random vectors which have identical distribution. Each tree casts a single vote to the most popular class in x. After running k times, the algorithm gives a sequence of classification results which are given by $\{h_1(x), h_2(x), \ldots, h_k(x)\}$

The final classification results is determined by the majority vote which is computed by using the decision function which is given by equation 2.30.

$$H(x) = arg(max_y \sum_{i=1}^{n} I(h_i(X) = Y) \tag{2.30}$$

Where $H(X)$ stands combination of classification mode, h_i single decision tree mode, Y output variable and I indicator function. In random forest classifier algorithm, there is one parameter which is used to measure the extent in which the average number of votes at X, Y for the correct class exceeds the wrong class. This parameter is referred to us margin function, which is given by the equation 2.31 and a larger margin function is indicator of the prediction is more accurate and the classification result is more confident.

$$mg(X, Y) = aV_k I(h_k(X) = Y) - max_{j \neq y} aV_k I(h_k(X) = j) \tag{2.31}$$

The generalization error of the random forest algorithm is given by the equation 2.32 as follows:

$$PE* = P_{x,y}(mg(X,Y) < 0) \tag{2.32}$$

When the number of decision trees is adequately large, $h_k = h(X, \Theta)$. Experimentally proven results show that random forest classifier algorithm does not over-fit though it produces a limited value of generalization error.
All sequences $\Theta_1, \Theta_2, \ldots, PE^*$ converges

$$P_{X,Y}(P_\Theta(h_k(x,\Theta) = y) = y) - max_{j \neq y} P_\Theta(h(x,\Theta) = j) < 0) \tag{2.33}$$

Another upper bound of the generalization error is given by the equation 2.34

$$PE^* \leq \hat{\rho}\frac{1 - s^2}{s^2} \tag{2.34}$$

Where s stands the strength of the classifier and $\hat{\rho}$ measures the correlation. From equation 2.34, we can understand that the generalization error of the random forest algorithm depends on two parameters—the strength of individual trees and the correlation between the individual trees in the forest.

Chapter 3

Literature Review

3.1 Introduction

Bug prediction is a mechanism of identifying defective components of a software system before it is released. This involves three basic components—selecting the appropriate software metrics for conducting the prediction process, identifying the parameters that are going to be predicted, and deciding what kind of model is used for prediction [33]. It is beneficial to improve the quality, to ensure stability, and to grant robustness of the software systems [10]. For the last few decades, a myriad amount of researches has been done to come up with a better performing defect prediction model. These include: code smell and static code metrics [24], investigating different aspects code smell [25][23], using object-oriented metrics [22], using change metrics [34][2][1] etc. By taking the aforementioned facts into consideration, the coming sections will discuss about different aspects software defect prediction approaches and the metrics used to predict software defects. Hence, the next section presents the details of different types of defect predication metrics.

3.2 Software Metrics

Software metrics are a measure of a portion of code or piece of behaviors of computer applications life cycle which are mapped to a certain numerical value [35]. In other words, software metrics are some sort of measurements that capture different aspects of the software development life cycle and maps to a certain numerical value and this number tells about different aspects of the software development. For instance, the efficacy of the design (code smell-based metrics), the complexity of the software (complexity metrics and size metrics), the frequency of the change in software systems (change metrics), etc.

Generally, software prediction metrics are classified into two major classes [36]. These are code metrics which are directly extracted from existing source code and process metrics which are collected from archived historical data that are found in software repositories [36] (version control systems like

Github and issue tracking system like Bugzilla).

In the last five decades, an enormous amount of research works has been done to formulate and define better software metrics that can be applied to software defect prediction. Among this research works, the first size metrics (which is based on LOC) was formulated by Akiyama [12][36][37] which utilizes the number of lines of code as an independent variable to predict software bugs. This scholar formulated his size based metrics as shown in equation 3.1 [12][38].

$$D = 4.86 + 0.018 \times LOC. \tag{3.1}$$

Where :

D is the number of defects in a code

LOC is the number of lines in a code

However, when the complexity of the emerging software system increased, Akiyama's formula becomes too simple to be applicable to these complex software systems [36][38]. As a result, different defect prediction metrics have been formulated by different scholars. For instance, Halstad comes up with more robust and complex metrics that rely on the number of operators and operands in a code [36]. Subsequently, other software defect prediction metrics are formulated by different scholars. A brief summary of some of the commonly known metrics is discussed below.

1. Static Metrics—these metrics capture different aspects of a program source code [34] which include: the line of code (the total line of code of the program), number of statements, the portion of branch statement like if conditions and switch statement, the number of classes and interfaces, the average number of methods per class, the average number of statements per method and the like.

2. Change Metrics—the change metrics capture the evolution process of the source code or the file [34] and tracks the extent of changes in version control systems [36]. During the development and maintenance stage of software development, a software system undergoes a different stage of changes. The change metrics capture different aspects of these changes which include the average and the maximum number of line of code added[34], the number of refactoring done to a file [36][34], the age of the file, weighted age, the number of authors, the maximum and average changes done to the file, the number of post-release bugs [34], the number of changesets (the number of files committed together) [36] etc.

3. Popularity Metrics—these metrics are developed by Bacchelli [39] by extracting information from the email archive of developers. The rationale behind these metrics is defective program artifacts require more idea exchanges than clean ones [36][39]. Under the basis of the aforementioned rationale, they developed popularity metrics that consist of the number of emails, the number of characters in email, the number of email trades, the number of mails in a thread, and the number of authors [39]. After empirically tested, popularity metrics are proved to be able to boost both the performance and the explanative power of the previously existing defect prediction systems [36][39].

4. Ownership and Authorship—these metrics investigate to what extent the ownership affects the quality of software systems. The empirical result of this research work indicated that a higher level of ownership leads to fewer bugs [36][40].

5. Relative Code Churn—these metrics measure the number of changes in a software system which includes like the number of lines added, the number of lines deleted, the number of files churned (the number of files churned/changed in a certain module divided by the number of files in that particular module) etc [36].

6. ObjectOriented Metrics—these metrics capture parameters which are related to the object oriented design paradigm. These include metrics like the number of methods defined in a class, the depth of inheritance tree, number of children, the coupling between objects, lack of cohesion on methods, the total number of non-empty and non-comment lines of code, etc. [21].

7. Micro-interaction—these metrics investigate whether defects are introduced by developers' interaction with the system. For instance, elongated editing of the code leads to defect [36].

8. Code Smell Metrics—code smells are either symptom of poor design consideration or poor implementation choices during the design and development of a software system [41]. The presence of code smells in a certain module of a software system is one indicator of the change proneness of that particular module [42]. Therefore, exploiting such information will be helpful to identify the probable future defects in a software system. Catoino et al. [42] list several code smell-related metrics which include access to foreign data, changing classes, coupling dispersion, changing methods, number of accessor methods, number of public attributes, the weight of a class, etc.

3.3 Software Defect Prediction Approaches

Since software defect prediction is an essential tool for deploying resources efficaciously and ensure the reliability of software systems, an enormous amount of research works has been done for the last five decades. These research works are versatile in terms of the approach they follow, the algorithm they applied, the features/metrics they selected as independent variables to make the prediction etc. Some of the research works use static code metrics as independent variables, the others used the static and dynamic features and the other uses the semantic features of the software code to predict the possible future software defects. Aside from the feature selection scheme, the granularity of the prediction level and the algorithm used are also different.

3.3.1 Semantic-Based Approaches

Even if the pragmatic experiments showed that static code metrics gave encouraging results towards software bug prediction, they are not cable enough to capture all aspects of defects in a software system [2][43]. Some defective codes might have the same code metrics manifestation with their clean counterparts in terms of complexity, size, etc. If we see the following codes as an example, the first one is a clean code and the second one is a buggy code that enters into an infinite loop. However, if we try to express them in terms of static metrics they are exactly alike.

```
x=0;
while (x<20){
   x++;
}
```

The above code is a clean code that iterates 20 times and exits from the loop however the below code is a buggy code that iterates indefinitely.

```
x=0;
while (x<20){
   x--;
}
```

If the above codes are examined in terms of static metrics-based evaluation, no one can find any difference. Therefore, the above practical example dictates that there are defects that cannot be captured by using the traditional metrics-based approach.

To give a solution for the cases like the above, Albahli [2] develops an ensemble type bug prediction model which extracting the semantic feature of the code and tackle bugs that cannot be caugheted by static metrics. He builds a classifier that ensembles Random Forest, XGBoost, and Multilayer perception models as a single classifier model. On the other hand, Fan et al [43] build a deep neural network-based classifier that takes into account both the static metrics and the semantic feature of the code. Liang et al. [44] do a similar semantic-based bug prediction model using the LSTM method. Even if the final modeling mechanism they followed is unique, all the above mentioned three research works extract the semantic features of the software program code using natural language processing techniques.

3.3.2 Object-Oriented Metrics Based Approach

Singh and Verma [22] conducted empirical research to investigate to what extent the object-oriented metrics are capable of predicting bugs in open source software systems using Naive Bayes and J48 machine learning algorithms. Based on their empirical results they come to the conclusion that the aforementioned metrics are possible candidates' software defect prediction. Malhotra et al. exploited the same metrics to compare 14 machine learning algorithms and the result of this experiment dictates that object-oriented metrics are are valuable candidates for bug prediction schemes as they are highly correlated with the dependent variable and have the least correlation with each other.

3.3.3 Change Metrics Based Approach

Change metrics based bug prediction is a mechanism of identifying defective code elements in a software system by looking at the details of the changes which took place over a certain period of time [45]. It is programming language independent and does not give attention to the content of the source file. It is also advantageous since it operates at a fine granularity level (which is change), the system automatically gives a clear assignment for a developer to fix a bug that comes with the change immediately at check-in time (commit time), predictions are made early at a time where the design decisions are fresh [1].

Kamei et al. [1] built a logistic regression algorithm based *just-in-time* defect prediction which exploits the change-related metrics as independent variables of the prediction model. This research work relies on the dataset which is prepared by the same authors from 7 open source and 5 commercial software systems. The model uses 14 change metrics based prediction parameters. The result of the

experiment demonstrates that this approach outperforms the baseline random predictor about by 90%. On the top of this research work, Qiao and Wang [3] built a deep learning algorithm based *just-in-time* defect prediction model that considers efforts required to inspect the defective code elements. This approach does not consider bug prediction as a classification problem, it considers it as a ranking problem. To achieve this, among the 14 dimensions of Kamei et al.'s dataset [1], it uses 10 of them as a predictor independent variable keeping the two (LA, LD) for the purpose of computing the effort required to make that particular change. The other two metrics (ND, REXP) are discarded since they are highly correlated with each other and other parameters as well. After doing this, they calculate the benefit-cost ratio (BCR) by taking the ratio of the defect predictor model output to the effort required for inspection which is measured by the total lines of code added and deleted during the change (code churn). Equation 3.2 shows the mathematical formulation of the BCR calculation [3].

$$BCR = \frac{\text{probability of defect}}{\text{effort}} = \frac{p(d)}{la + ld} \tag{3.2}$$

Where :

la: the total number of lines added

ld: the total number of lines deleted

Sometimes change metrics might not be effective to predict software bugs because one change might lead to a change of different parts of a code [19]. This intern leads to another issue. That is, if a certain bug is caught and fixed, it is also necessary to fix all associated sections of the code too. To demonstrate this with an example, if the method of a Java abstract class is found erroneously defined, this error is also expected to be propagated to all classes which implemented this erroneous method. Hence, a fixing action that corrects the error in the first abstract class is also expected to be done for other classes. Simple change metrics are not able to resolve this issue. To find a solution for such problems Herzig et al. [19] introduced a new change-based metrics that takes the aforementioned problem into account—change genealogy metrics. The aforementioned researchers examine the impact of one set of changes to the other changes which are applied later and formulated special software change-based metrics called change genealogy metrics.

3.3.4 Code Smell Based Approach

Code smells are the suboptimal implementation of codes [42] and design flaws [23] in the software development process. These design faults and implementation defects have a negative impact on the

change process of a certain component of a software system which is affected by them. Ubayawardana and Karunaratna [23] build a code smell based defect prediction model in order to fill the gap left by other metrics. The main motivation for coming to this idea is that some metrics are not complete by themselves. For instance, if LOC metrics are taken into consideration, the large number of lines of code is considered to be buggy but there is no evidence that grants codes with a smaller line of code are defect-free. By taking this fact into consideration, the above mentioned scholars built a code smell based predictor model and compared it with the process metrics. After examining their empirical results, they conclude that code smells metrics will boost the performance of the software defect prediction if they are used with other types of metrics. After this, Catolino et al. [42] exploited the information regarding the intensity of code smell and come up with additional parameters of bug prediction which is used with other previously known parameters—product, process, and developer based metrics. After conducting empirical experiments for testing the impact of these parameters, the results demonstrate that the augmented parameters improved the performance of the baseline model.

3.3.5 Developer Centred Approach

Aside from the previously mentioned bug prediction metrics, the other aspect that takes the attention of researchers is the different aspects of the developer himself. This includes experiences, sentiments the nature of projects he participated in and the like. Di Nucci et al. [14] for instance, develop a class level bug prediction approach that investigates the impact of developers scattering towards bug prediction. The rationale of this perspective is that if a developer is assigned in different projects, it has a negative impact on attention division that leads the developer to commit defective code. To examine the validity of the previously mentioned rationale the researches look the scattering effect from two perspective—structural (which examine the type of files modified by developers) and semantic (textual similarity of the changed codes). The latter one is included because there is a possibility of placing a class in a package which is not belonged to it due to different reasons like poor design. This model is tested in different open-source software systems and demonstrates encouraging results for well-modularized systems. What it can be understood from this result is that this metrics will be helpful if it is adapted for a well modularized real-world projects.

Chapter 4

Proposed Approach

Before delving into the details of our proposed approach, we will discuss the six change request based metrics which are extracted from bug tracking systems. Section 4.1 is dedicated to discuss about each change request metrics and the rationale behind their definition. The next subsequent sections will present our proposed approach in detail.

To assess the impact of change request-based metrics on *just-in-time* defect prediction, we used the approach followed in other similar studies [1][3]. Our proposed approach involves data collection, preprocessing, model building, and evaluation. Each of the aforementioned components of the system is discussed in detail from section 4.2 to 4.5. Figure 4.1 shows a block diagram that presents the overall architecture of the system.

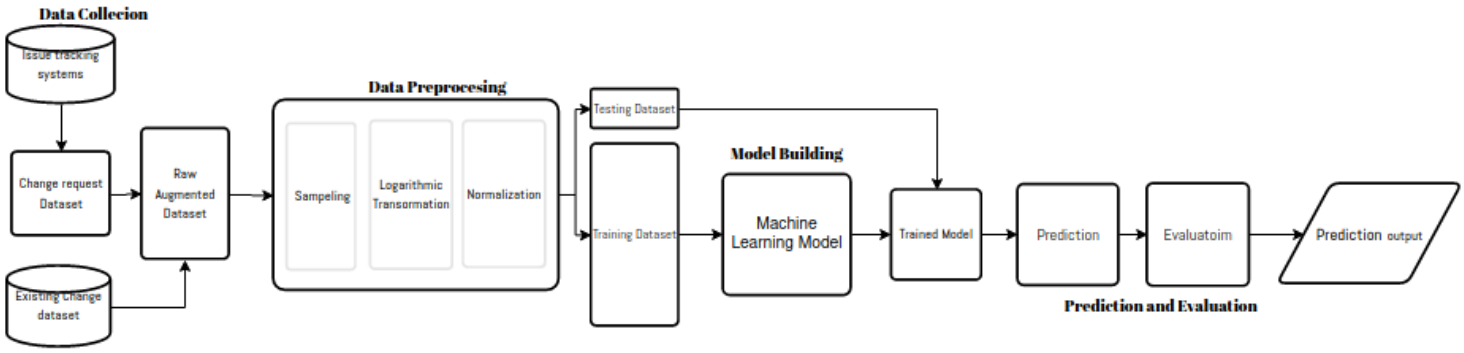

Figure 4.1: System Architecture

4.1 Change Request-based Metrics

To predict defects *just-in-time* bug prediction schemes, previous studies used change based metrics which are extracted from version control systems. Change metrics are advantageous over the other types of metrics mainly because they are programming language independent, work at fine granularity level and are capable of giving a clear assignment for the developer to fix the bug at check-in time [1]. They, however, fail to take into account related relevant information that come with the change request. The main aim of this research is to explicitly take into consideration the information

extracted from change requests. The rationale being change requests contain information related to the clarity, difficulty and eccentricity of the change request, which could determine the change's proneness to defect. Hence, we conjuncture that such information will give a better edge to prediction algorithms in distinguishing probable defective changes. To this end, we have examined issue tracking systems and identified six change request-based metrics that could enhance the performance of just-in-time defect prediction. Below we discuss the identified six metrics along with the rationale.

Time to fix (TTM). This is the time span between the submission of the change request and its resolution. Change requests that take a longer time to implement are usually complex and major changes. Such complex and major changes are likely to introduce defective changes. This metrics is extracted from issue tracking system by taking the difference of the time stamps in which a particular bug is reported and fixed. Equation 4.1 shows the mathematical formula applied for extracting these metrics from issue tracking systems.

$$TTM = TR - TF \tag{4.1}$$

where:

TTM: Time to fix

TR: The time stamp in which the bug is reported

TF: The time stamp in which the bug is fixed

Number of developers assigned (NDA). From the reporting time of a change request to its final resolution one or more developers are assigned to implement the change. Relatively simple change requests are usually given to one or two developers while more complex and major change requests are given to a relatively large number of developers. Hence, the number of developers involved in a change request could be used to explain the nature of the change request and the probability that change would introduce a new defect. This metrics is extracted from issue tracking systems by counting the total number of developers assigned during the history of the bug i.e., from the time the bug is reported until it is fixed.

Priority of the bug. Issue tracking systems provide a means to prioritize change requests. The priorities given to the change requests determine the importance and level of attention the requests would get; and, hence, impact the probability that change would introduce a new defect.

Severity of the bug. Similar to priority, issue tracking systems allow to categorize bugs at different levels of severity based on their impact on the system. Bugs with higher severity level (e.g., blocking)

require tight time and due effort to resolve them. Committing corrective changes to fix such bugs might introduce other defects in the source code.

Number of Comments (NC). As a change request is posted, users of the software system and developers exchange comments to clarify and get the right solution to the requests. Usually a large number of comments indicates that the change request is either eccentric or complex; and, hence, impact on the probability a new defect would be introduced as the change is committed.

This metric is extracted by counting the number of comments made by developers and users in issue tracking systems for a particular change request.

Depth of Discussion (DD). This metrics measures the number of words used during the discussion about a change request. The rationale behind this metrics is that if a large number of words is used while discussing a certain change request, the request could be hard to explain due to eccentricity of the request and multiple dependencies. As a result, committing corrective change for resolving such changes might increase the probability of inducing another defect. This metrics is extracted from issue tracking system by parsing the comments made by developers and users about a particular bug. The parsed words are counted to represent the depth of discussion of that particular bug in numerical form. The summary of the change request metrics is presented in table 4.1. Below we

Table 4.1: Summary of the Identified Change Request-based Metrics

Metrics	Description	Rationale
TTM	Time to fix	Longer time to fix a bug or do the last modification indicates that the code is likely to be prone to defects
Priority	Priority of a Bug	Bugs with higher priority are usually handled with a constrained time, which might lead to commit defective changes
Severity	severity of a Bug	Bugs with higher severity are usually handled with a constrained time, which might lead to commit defective changes
NDA	Number of developers assigned during the history of a bug	A large number of developers assigned for a certain bug indicate the eccentricity or complexity of the bug. This could lead to commits of defective changes
NC	Number of Comments	If stakeholders of a bug do a lengthy discussion about the bug, it is either because the bug is complex or lacks clarity. Committing a change to correct such bugs might usually lead to another defect.
DD	Depth of discussion	If the number of words used during a discussion about a bug is large, it might be an indicator about the peculiarity or complexity of the bug. Committing to resolve such bugs might lead to another defect.

detail each component of the architecture.

4.2 Data Collection

The dataset used in this research work consists of two different datasets that are augmented together to create one large dataset. The first dataset is taken from a part of publicly available dataset which is

created by Kamei et al. [1] for *just-in-time* defect prediction. The other dataset which is augmented with the previously mentioned dataset is collected from the Bugzilla issue tracking repositories using.

Table 4.2: Summary of the Existing Change Metrics [1]

Metrics	Description	Rationale
NS	Number of modified subsystem	Change modifying many subsystems are more likely to be defect-prone
ND	Number of modified directories	Change modifying many directories are more likely to be defect-prone
NF	Number of modified Files	Change touching many files are more likely to be defect-prone
Entropy	Distribution of modified code across a file	Change with higher entropy are more likely to be defect-prone, because a a developer will have to recall and track a large number of scattered changes across each file
LA	Line of code added	The more line of code added, the more likely a defect is introduced
LD	Line of code deleted	The more line of code added the higher the chance of a defect
LT	Line of code in a file	The larger a file, the more likely before the change might introduce defect
FIX	Whether or not change is defect fix	Whether or not the made in an earlier implementation therefore it may indicate an area where errors are made likely to occur
NDEV	The number of developers changed files	The larger the NDEV, the more likely defect is introduced, because the file revised by large developers often contain different design thoughts and code codding styles
AGE	The average time interval between the last and the current change	The lower the age, (i.e. the more recent the last change), the more likely a defect will be introduced
NUC	The number of unique changes	The larger the NUC, the more likely a defect is introduced, to the modified because a developer will have to files recall and track many previous changes
EXP	Developer Experience	More experienced developer is less likely to introduce a defect
REXP	Recent Developer Experience	A developer that has often modified the file in months is less likely to introduce a defect, because she will be more familiar with the recent developments in the system
SEXP	Developer experience on a subsystem	Developers that are familiar with the subsystems modified by a change are less likely to introduce a defect

To identify the change request from ITS, we used the change request id in the defect inducing commit messages. Developers usually insert change request id in the message of a commit corresponding to a change request. The defect inducing commit is identified by matching the commit timestamp in the change based-metrics record with the commit timestamp in the corresponding software repository. After identifying the defect inducing commits, the corresponding commit messages (comments in the commits) are mined to extract the change request ids (bugids). Then, by using the change request id as a unique identifier, we extract the change request metrics from issue tracking systems using web-scrapping mechanism.

While applying the web scrapping method to collect the new information from the issue tracking database, the bugid is used as a unique identifier. The scrapping program traverses all the necessary pages of the issue tracking website and fetches the required information. After collecting the required information, at each iteration, it writes one row of data into a CSV file. While scrapping the page, if the information associated with the particular bug data is missed in the issue tracking database, the entire information will be discarded (the associated data of the publicly available dataset is also excluded). The extracted six change request-based metrics are then augmented to the change metrics, creating a total of 20 metrics. The Algorithm used for extracting the data is shown below.[1]

[1]A fragment of python code which implements this algorithm is found in Appendix A.3.

```
procedure extractMetrics(array_bugID)
   for each array_element i in array_bugID:
      if bug i exists in Bugzilla_repository:
         extract all the metrics from the repository
      else
        discard bug i and move to the next
end procedure extractMetrics
```

While the data collection tasks are performed, some data conversion mechanisms are applied in order to make the data be easily processed by the model. Such conversion procedures are applied to those metrics that are not represented numerically by issue tracking systems like the priority of the bug and severity of the bug. For those parameters, each level of severity and priority is directly mapped with a unique integer to ease the processing of the data. After the previously mentioned steps are completed, the two CSV files are augmented to form a single dataset which consists of JIT based metrics and the change request-based metrics.

4.3 Preprocessing

The performance of a machine learning model is highly affected by data preprocessing [10]. In this important stage, different activities are performed to minimize the impacts of noise and to create clean data for training the model. The activities which are involved in this stage of development are mainly targeted to create balanced data (to avoid the skewness), scaling down the data to be confined between a specific range of values and to remove irrelevant data (for instance those CSV rows that contain null values). To exploit the aforementioned benefits of data preprocessing, in this thesis, the following four types of preprocessing activities are conducted before train the model:

4.3.1 Logarithmic Transformation

When the distribution of data in a dataset is highly skewed, it is necessary to perform a logarithmic transformation to make the data less skewed. When a logarithmic transformational is applied on a data it will be helpful to reduce the variation of the data, to make the patterns in the data easily decipherable, and to make the data conform to normality [3]. The log. transformation process can be formulated using equation 4.2.

$$d_l = ln(d_i) \tag{4.2}$$

where :

d_l : the log. transformed version of a particular datum d_i

d_i : any datum in a particular column of the dataset

$ln(d_i)$: the natural logarithm of datum d_i

4.3.2 Normalization

Input data normalization is a process of scaling down of the data to be confined within a specific range [10]. The range of the values of the preprocessed data is dependent on the form of the normalization technique applied to the data. Input data normalization is essential for both enhancing the computation speed of the machine learning algorithms while training and testing stage and to obtain good output from the system [46]. In this thesis, we use Min-Max Normalization, which is being formulated using equation 4.3.

$$d_n = \frac{d_i - d_{min}}{d_{max} - d_{min}} \tag{4.3}$$

where :

d_n : a normalized version of a datum d_i

d_i : any datum in a particular column of the dataset

d_{max} : the maximum value of a particular column of the dataset

d_{min} : the minimum value of a particular column of the dataset

4.3.3 Sampling

When a dataset consists of imbalanced data, for instance, if the dataset consists of 90% of clean and 10% buggy data, the output of the system will not be accurate and gives erroneous results. Undersampling the majority class or oversampling the minority class mitigates the negative impact of class imbalance problem. The class imbalance problem might be handled at different levels—data level, algorithm level or the hybridization of the previously mentioned two approaches [47].

The dataset used in this research work is highly unbalanced, the majority class (bug-free data) are much larger than the minority class (buggy data). In order to attenuate the skewness of the data, we examine both random undersampling and SMOTE [48]. With regard to under sampling, some works of literature pointed out that, though it has some drawback of losing useful data [36], it is one of the most effective methods [49] to achieve the aforementioned objective. However, in our case, since the data is severely unbalanced in most of the systems in our study, the performance of the random undersampled data is poor. As a result, we prefer to oversample the minority class using the commonly used technique called SMOTE.

4.3.4 Removal of Irrelevant Data

In this stage of preprocessing, we discard data which are not complete (eg. tuples that have all the required data in a public dataset but which are missed in the bug repositories) and have null values are excluded from the dataset. This helps to create a clean dataset that can be used efficiently by the respective machine learning models.

4.4 Model Building

After the data preprocessing stage is completed, the next stage is building the model. To build our model we use five different machine learning algorithms. Among these five algorithms, one is deep neural network based classifier, while the other algorithms are tree based ensemble classification algorithms and logistic regression. As one of main objective of this research work is to figure out the impact of the newly introduced change request based software defect prediction metrics, we built separate models for the baseline dataset and the augmented dataset. While building the two separate models, except some mandatory parameters that need to be different, we carefully set the models to have similar configurations.

4.5 Prediction and Evaluation

After the models are built, then the dataset is divided into two parts for training the respective models and testing the models. By using the test data, the performance of the trained model is tested using different evaluation parameters which are extracted from the standard evaluation matrix. These

include: Accuracy, precision, Recall, F1-Score and False positive rate. In order reduce bias in our results and reduce variances in estimated results, we use 10-fold cross validation techniques which are commonly used by numerous machine learning and other related research works.

Chapter 5

Experiments

In order to answer our research questions which are defined in section 1.2, we conducted successive empirical experiments. This section details the characteristics of the dataset used for this research work, the experimental setup and configuration of each experiment, and the evaluation mechanisms applied to measure the performance of our approach.

5.1 Dataset

The datasets used for the experiment are collected from four open source software systems: Bugzilla, Mozilla, Eclipse JDT and Eclipse Platform. The four systems are selected because they have already prepared change metrics based dataset and issue tracking systems. The datasets containing the change metrics are prepared by Kamei et al. [1], for use in *just-in-time* software defect prediction. The datasets contain fourteen change metrics categorized into five dimensions, i.e., diffusion, size, purpose, history and experience. Table 4.2 presents a brief summary of the change metrics. These datasets are used in other similar research works [49][3][2]. In this study, we refer to these datasets as *baseline datasets*.

For our experiment, we prepared datasets containing the six change request-based metrics described in Section 4.1. The change request-based metrics are extracted from the change request reports filed in ITS. To identify the change requests that correspond to the change metrics records, we used the change request id (bugID) found in the defect inducing commit messages stored in the corresponding software repository, in our case the code version systems (CVS). The defect inducing commits are identified by matching the commit timestamps in the CVS with the commit timestamp provided in the change-based metrics dataset. Before matching the commit timestamps, we removed all records with the same timestamp in the change-based metrics dataset. Since, the precision of the timestamp in the change-based metrics dataset is at minutes level, we also discarded commits in the CVS made within a minute and contain different commit messages as they cannot be uniquely mapped to the change-based metrics record. Change-based records with matching commits and no change request id in their commit messages are also excluded from the study as they cannot be used to identify

the corresponding change request-based information from issue tracking systems. Following these filtering rules, the datasets are prepared for the above four software systems. The new datasets with the six metrics are then augmented to the corresponding dataset prepared by Kamei et al. [1]. We refer to the new datasets containing both the six and 14 metrics as *augmented datasets*. Summary of the datasets is shown in Table 5.1.

Table 5.1: Summary of the dataset

System	Collection Period	Original Dataset		Matched data with CVS and ITS	
		# of Changes	% of Defect	# of Changes	% of Defect
Eclipse JDT	05/2001–12/2007	35386	14	11982	15.89
Mozilla	01/2000–12/2006	98275	5	37751	4.91
Eclipse Platform	05/2001–12/2007	64250	14	12815	21.58
Bugzilla	08/1998–12/2006	4620	36	2698	40.32

From the identified six change request-based metrics, severity and priority do not have a numerical value in the issue tracking systems. In order to ease further data processing and computation, we mapped these non-numerical values to numbers. For example, in Bugzilla issue tracking system the severity of a bug is expressed as immediate (i.e., most severe), highest, high, normal, low and lowest (i.e., least severe). These severity values are sequentially mapped from six (i.e., most severe) to one (i.e., least severe). Priority has seven levels: blocking, critical, major, normal, minor, trivial and enhancement. Similar to severity, we mapped the priority levels from seven to one, respectively from the highest priority, i.e., blocking to the lowest priority, i.e., enhancement.

5.2 Experiment Setup

To evaluate the impact of the change request-based metrics, we conducted two sets of experiments. In the first set of experiments, we used the change metrics computed by Kamei et al. [1] to build and evaluate the defect prediction models. We refer to the results of these experiments as the baseline. In the second set of experiments, we used the augmented dataset to build and evaluate the defect prediction models.

In order to evaluate the performance of our proposed approach, we use a ten-fold cross validation scheme. In this approach, the data is randomly divided into ten folds and one is used for validation and the remaining nine are used for training the system [45]. This process will continue ten times by shuffling the $(\frac{1}{10})^{th}$ of the data as testing data from the one which is used for training in the previous stage. Then the average of the ten results will be taken as the final result. Ten-fold cross validation

is popular and the most widely used evaluation mechanism [49] in different software engineering research works.

In order to conduct an empirical evaluation for our proposed approach, we use five machine learning algorithms, i.e., Deep Neural Network, AdaBoost, XGBoost, Random Forest and Logistic Regression.

The brief explanation of the hyperparameter configuration of each of the aforementioned algorithms is presented below.

5.2.1 DNN Model Setup

While selecting the optimum model to conduct experiments we follow a trial and error scheme as there is no specific formula or guideline to determine the number of layers of the DNN model and the number of neurons in each layer. Therefore, by starting from a simple structure we try to examine the learning curve of the system and finally, we select the one that gives us a better result.

After following the aforementioned procedures, the final setup of our DNN model becomes as follows:

- Number of Layers: 5 (3 hidden, 1 input, and 1 output)
- The number of neurons in each layer: the number of neurons in each hidden layer is 28.
- The number of input layer neuron: for the augmented dataset it is 19 or 20 to capture all the metrics that consists of the newly defined 5 or 6 metrics and the existing 14 metrics and for the baseline dataset, it is 14 to capture the existing dataset.
- The number of Output Neuron: 1 (since the system is a binary classifier, it is enough to provide the required output.)
- Kernel initializer: uniform for all layers
- Activation function: rectified linear unit (ReLU) for all layers except the output layer which is sigmoid.
- Optimizer: Adaptive moment estimation (ADAM)
- Loss Function: Binary cross_entropy
- Learning Rate: 0.001
- Batch Size: 64

5.2.2 XGBoost Model Setup

Since our target is to figure out the enhancement effect of our change request-based metrics, we did not give much attention to the details of hyper-parameter optimization for this model. The reason behind this is since both the baseline experiments and experiments done using augmented dataset are done with the similar model configuration, its effect on the delta of the result is too minimal. By taking this into consideration, we prefer to use the default configuration for most of the hyper-parameters. Those parameters which are configured by ourselves are done empirically by observing the value of the respective performance metrics like accuracy and F1-score. The hyper parameter configuration of the XGBoost algorithm is presented as follows: [1]

- max_depth = 200
- n_estimators = 600
- learning_rate = 0.001
- colsample_bytree = 0.3
- alpha = 10

5.2.3 AdaBoost Model Setup

We use the same procedure that we follow while setting the hyper-parameters of the XGBoost algorithm above. Hence, the hyper parameter configurations of the AdaBoost algorithm are: [2]

- n_estimators=600
- learning_rate=1

5.2.4 Random Forest Model setup

By following the same procedures which are used in the previously described algorithms the hyper parameters of the Random Forest algorithm are configured as follows: [3]

[1] A fragment of python code implementation for this algorithm is found in Appendix A.1.2
[2] A fragment of python code implementation for this algorithm is found in Appendix A.1.1
[3] A fragment of python code implementation for this algorithm is found in Appendix A.1.3

- n_estimators=600
- max_depth=200
- random_state=0

5.2.5 Logistic Regression Model setup

For this algorithm, no special hyper parameter configuration is made. We just used the default model setup. The default hyper-parameter settings are presented as follows. [4]

- n_penality=l2
- dual=false
- tol=$1e^{-4}$
- float=1;
- class_weight=none
- random_state=none
- solver_'lbfgs'
- max_iter=100
- multi_class='auto'

5.2.6 Experiment Environment

The hardware and software environments used to conduct the experiments briefly presented below.

The hardware system consists of an Intel(R) Core(TM) i7-4510U CPU 2.00GHz, 8GB RAM laptop running Ubuntu 20.04 LTS (64-bit). The software systems, on the other hand, we use Python 3.7 and and its associated third party libraries for building the models, processing the data and visualizing the results etc.

[4] A fragment of python code implementation for this algorithm is found in Appendix A.1.5

5.3 Evaluation Metrics

In order to evaluate the performance of our approach, we use the common evaluation metrics which are derived from the standard confusion matrix shown in Table 5.2. The confusion matrix lists all possible outcomes of the prediction results. If a defective instance is correctly classified as defective, it is true positive (TP). If a clean instance is incorrectly classified as defective, it is a false positive (FP). Similarly, if a clean instance is correctly classified as clean, it is true negative (TN) and if a defective element is misclassified as clean, it is false negative (FN).

Table 5.2: Confusion Matrix

	True Class	
Classified as	Defect	Non Defect
Defect	TP	FP
Non defect	FN	TN

This research work uses the following system performance evaluation metrics to evaluate the system:

- **Accuracy**—it is also referred to as classification rate [22]. Accuracy measures the ratio of the total number of correctly predicted classes to the total test data. This performance evaluation parameter might lead to an erroneous conclusion if the data has a class imbalance problem. However, since our data eliminates this problem through random oversampling, this performance parameter can be considered as a good performance indicator. The accuracy of a model can be calculated by the following formula from the confusion matrix.

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

- **Precision**—It is the ratio of the total number of predicted defective components of a data as defective to the total number of data which are predicted as defective. The precision of the system can be computed by the following mathematical formula.

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

- **Recall**—it is also referred to as defect detection rate [22]. Recall is the ratio of the total defective elements that are predicted as defective to the total number of defective records in

test data. If the recall value is high, it shows that the total number of undetected defects by the system is low. The value of recall is calculated using the following mathematical equation.

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

- **True Negative Rate**—it is also referred to as specificity. It measures the probability the portion of negative outcomes are predicted as negative. Equation 5.4 shows the mathematical formulation of the TNR.

$$TNR = \frac{TN}{TN + FP} \tag{5.4}$$

- **False Positive Rate**—it is the measure of false alarm rate and mathematically formulated using equation 5.5.

$$FPR = \frac{FP}{FP + TN} \tag{5.5}$$

- **False Negative Rate**—it is also called miss rate and measures the probability that a true positive is misclassified by the system. FNR is calculated using equation 5.6.

$$FNR = \frac{FN}{FN + TP} \tag{5.6}$$

- **F-Measure**—it is the harmonic mean of precision and recall. The quality of the predicting system has a direct relation with this performance evaluation metric. The value of F-Measure is calculated using the following mathematical equation.

$$F - Measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{5.7}$$

- **AUC-ROC**—The Receiver Operator Characteristic (ROC) curve is one type of evaluation parameter for binary classification problems. It is a two dimensional curve that measures the performance of classifier systems. It is helpful to understand the performance of the systems as it provides the way to compare classifiers at varying the sensitivity and specificity trade-offs [50]. The ROC curve can be considered as a probability curve that plots the TPR against FPR at various threshold values. The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve.

Chapter 6

Results and Discussion

6.1 Results

This section presents the details of each experiment result. In order to make the results clear and easy to follow, we present each set of experiments with its associated research question.

RQ1 *To what extent the change request-based metrics improve the prediction performance of JIT defect prediction scheme?*

While using the augmented dataset, F1-score of JIT defect prediction is improved in 19 of the 20 cases. F1-score of JIT defect prediction in the four systems is improved by an average of 4.8%, 3.4%, 1.7%, 1.1% and 1.1% while using AdaBoost, XGBoost, Deep Neural Network, Random Forest and Logistic Regression, respectively. The highest improvement in F1-score, 7.8%, is observed for Eclipse Platform while using XGBoost algorithm. The augmented dataset did not show improvement in F1-score for Bugzilla while using Random Forest. The FPR for BugZilla, however, has an improvement while using the augmented dataset with Random Forest. Compared to the other systems, the improvement observed for BugZilla is relatively low in three of the five machine learning algorithms. The F1-scores of BugZilla for the five machine learning algorithms are lower than 78%. The other systems, however, have F1-score that goes upto 95.6%. The small improvement in BugZilla could be attributed to the quality of the change requests posted in the issue tracking systems and the size of the dataset.

Looking at the accuracy metrics, the augmented dataset gives a better result in 19 of the 20 cases. The average delta in accuracy of the four systems is 4.8%, 3.4%, 1.8%, 1.3% and 0.8% while using AdaBoost, XGBoost, Deep Neural Network, Random Forest and Logistic Regression, respectively. This shows that the change request-based metrics have helped in better identifying both correct and defect introducing change commits. In addition, among the five machine learning algorithms used in the study, the highest average delta improvement in F1-score and accuracy is observed for AdaBoost followed by XGBoost.

Among the five machine learning algorithms used in the study, Random Forest followed by AdaBoost and XGBoost gives the highest accuracy and F1-score while using both augmented and baseline

datasets for majority of the cases. AdaBoost and XGBoost have comparable performance. The lowest accuracy and F1-score values are recorded for Logistic Regression algorithm. This shows that the machine learning algorithm used in the JIT defect prediction also plays important role. The comparative results of the baseline dataset and the augmented dataset for the five machine learning algorithms are shown from Table 6.1 to Table 6.6.

Table 6.1: Results Using AdaBoost Algorithm

System	Dataset	Evaluation metrics					
		Accuracy	FPR	Precision	Recall	F1-Score	Δ F1-Score
Eclipse JDT	Augmented	88.29	6.86	92.42	83.43	87.68	**6.54**
	Baseline	81.76	14.96	83.99	78.49	81.14	
Mozilla	Augmented	81.81	19.56	80.96	83.18	82.06	**4.19**
	Baseline	77.64	23.42	77.07	78.70	77.87	
Eclipse Platform	Augmented	83.98	10.31	88.37	78.27	83.00	**6.90**
	Baseline	76.96	10.31	79.01	73.47	76.10	
Bugzilla	Augmented	74.07	25.47	74.41	73.60	73.92	**1.53**
	Baseline	72.86	25.59	73.67	71.30	72.39	

Table 6.2: Results Using XGBoost Algorithm

System	Dataset	Evaluation metrics					
		Accuracy	FPR	Precision	Recall	F1-Score	Δ F1-Score
Eclipse JDT	Augmented	91.20	1.78	97.94	84.18	90.54	**2.42**
	Baseline	88.71	6.29	93.02	83.72	88.12	
Mozilla	Augmented	95.72	1.95	97.95	93.40	95.62	**3.15**
	Baseline	92.64	5.14	94.62	90.41	92.47	
Eclipse Platform	Augmented	84.20	7.18	91.32	75.59	82.71	**7.86**
	Baseline	76.63	16.31	81.03	69.58	74.85	
Bugzilla	Augmented	72.08	23.17	74.43	67.33	70.67	**0.35**
	Baseline	71.77	23.48	74.04	67.02	70.32	

Table 6.3: Results Using DNN Algorithm

System	Dataset	Evaluation metrics					
		Accuracy	FPR	Precision	Recall	F1-Score	Δ F1-Score
Eclipse JDT	Augmented	71.91	34.24	69.57	78.07	73.51	**1.87**
	Baseline	70.63	33.07	69.40	74.34	71.64	
Mozilla	Augmented	81.35	23.08	78.81	85.78	82.14	**3.46**
	Baseline	77.74	26.67	75.51	82.15	78.68	
Eclipse Platform	Augmented	69.21	26.17	71.20	64.58	67.70	**0.59**
	Baseline	67.53	31.29	68.10	66.36	67.11	
Bugzilla	Augmented	71.49	25.59	73.24	68.57	70.60	**0.87**
	Baseline	71.24	23.85	73.79	66.34	69.73	

Table 6.4: Results Using Random Forest Algorithm

System	Dataset	Evaluation metrics					
		Accuracy	FPR	Precision	Recall	F1-Score	Δ F1-Score
Eclipse JDT	Augmented	91.63	7.69	92.21	90.94	91.57	**2.29**
	Baseline	89.12	12.35	88	90.6	89.28	
Mozilla	Augmented	94.79	7.63	92.73	97.20	94.91	**0.92**
	Baseline	93.81	9.10	91.40	96.72	93.99	
Eclipse Platform	Augmented	87.43	11.70	88.11	86.56	87.32	**1.61**
	Baseline	85.53	15.73	84.67	86.78	85.71	
Bugzilla	Augmented	77.48	20.93	78.39	75.90	77.11	**-0.52**
	Baseline	77.58	22.61	77.54	77.76	77.63	

Table 6.5: Results Using Logistic Regression Algorithm

System	Dataset	Evaluation metrics					
		Accuracy	FPR	Precision	Recall	F1-Score	Δ F1-Score
Eclipse JDT	Augmented	65.24	33.46	65.67	63.93	64.78	**0.68**
	Baseline	64.83	33.15	65.48	62.81	64.10	
Mozilla	Augmented	66.91	26.01	69.70	59.82	64.38	**1.03**
	Baseline	66.24	25.89	69.28	58.36	63.35	
Eclipse Platform	Augmented	63.90	32.60	64.95	60.40	62.59	**1.31**
	Baseline	63.28	31.54	64.83	58.11	61.28	
Bugzilla	Augmented	68.17	29.57	69.03	65.90	67.39	**1.39**
	Baseline	66.83	30.81	67.66	64.47	66.00	

While doing our experiments to answer this research question, we also try to figure out which algorithm is performing better towards enhancing the aforementioned defect prediction scheme (*just-in-time* defect prediction). As a result, we compare the performance of the five algorithms that are used in this research work. Here, our basis for comparing one algorithm with another one is the performance improvement which relates to different aspects of their predictive capacity.

The metrics used for comparison are accuracy, f1-score, TNR, FPR and AUC_ROC values. To compare the algorithms, we used three of the four systems which are used for this research work, i.e. Mozilla, Eclipse JDT and Eclipse Platform. These systems are selected because they have relatively good result.

The results of the experiments show that AdaBoost algorithm gives the best performance in most of the test cases. Among a total of 18 different test cases, in the fourteen cases AdaBoost outperforms the other four algorithms. In the remaining four cases, XGBoost performed better than AdaBoost.

The details of the comparison of the five algorithms are presented in Table 6.6.

Based on the results of the above analysis, it is possible to conclude that among the five algorithms

which are used in this research work, AdaBoost outperforms the other algorithms.

Table 6.6: Comparative analysis of the applied algorithms

Evaluation Metrics	System	AdaBoost (Δ)	XGBoost (Δ)	DNN (Δ)	RF (Δ)	LR (Δ)
Accuracy	Eclipse JDT	**6.53**	2.49	1.28	2.51	0.41
	Mozilla	**4.17**	3.08	3.61	0.98	0.67
	Eclipse Plat.	7.02	**7.57**	1.68	1.90	0.62
F1-Score	Eclipse JDT	**6.54**	2.42	1.87	2.29	0.68
	Mozilla	**4.19**	3.15	3.46	0.92	1.03
	Eclipse Plat.	6.90	**7.86**	0.59	1.61	1.31
TNR	Eclipse JDT	**8.10**	4.51	-1.17	4.66	-0.31
	Mozilla	**3.86**	3.19	3.59	1.47	0.12
	Eclipse Plat.	**9.23**	9.13	5.12	4.03	1.06
FPR	Eclipse JDT	**-8.10**	-4.51	1.17	-4.66	0.31
	Mozilla	**-3.86**	-3.19	-3.59	-1.47	-0.12
	Eclipse Plat.	**-9.23**	-9.13	-5.12	-4.03	-1.06
FNR	Eclipse JDT	**-4.94**	-0.46	-3.73	-0.34	-1.12
	Mozilla	**-4.48**	-2.99	-3.63	-0.48	-1.46
	Eclipse Plat.	-4.80	**-6.01**	1.78	0.22	-2.29
AUC_ROC	Eclipse JDT	**6.53**	2.49	1.28	2.51	0.41
	Mozilla	**4.17**	3.08	3.61	0.98	0.67
	Eclipse Plat.	7.02	**7.57**	1.68	1.90	0.62

Our experiment results show that augmented datasets are capable of improving the baseline results up to 7.57% in accuracy and 7.86% in F1-score. In addition the augmented approach improves the true negative rate up to 9.23%, reduce the false positive rates by 9.23% and false negative rate by 4.94%. Further more, among the five algorithms used in this research work, AdaBoost algorithm out performs the others algorithms in most of the experiments.

RQ2: *With regard to JIT defect prediction, what are the important features in the existing change metrics and the new change request-based metrics?*

In order to answer this research question, we computed the importance of each metric using the best performing algorithms. To evaluate the contribution of each metric for accomplishing the JIT defect prediction, we compute the importance of each metric for all of the systems which are under our investigation. The best performing algorithm in terms of delta improvement i.e., AdaBoost algorithm is used for computing the importance of each metrics. [1] AdaBoost uses a tree based classifier system to compute the importance of each metric that is considered as feature. As shown in the Figure 6.1,

[1] A fragment of python codes implementation for computing the importance of each feature are found in Appendix A.2.1

we can see that three of the six change request-based metrics contributed well to the *just-in-time just-in-time* defect prediction scheme

While AdaBoost algorithm is used, for Bugzilla, *AGE, line of code added (LA), number of modified files (NF), and severity* are among the top important features. *line deleted (LD), FIX, developer experience on subsystem (SEXP), priority and depth of discussion (DD)* are found to be the list important metrics for this particular system. Considering Eclipse JDT, on the other hand, *number of developers assigned (NDA), number of comments (NC) and severity* are the top three important features while *recent developers experience (REXP), developer experience in subsystem (SEXP) and priority* are among the features with the least importance level.

In the case of Mozilla, somehow a similar pattern is observed. *severity, number of modified file (NF)* etc are among the top important features while *time to fix (TTM), priority, The number of developers that changed the modified files (NDEV) and entropy* are among the least important metrics. Regrading Eclipse Platform, *number of developers assigned (NDA), number of comments (NC), severity, AGE and number of developers modified the the commit (NDEV)* are features that have the top level of importance. *developer experience on subsystem (SEXP) and priority*, on the contrary, are features with the least level of importance. In general, the details of each experiment results are presented in the Figure 6.1.[2]

[2]Note: The baseline change-based metrics and the change request-based metrics are explained in detail in section 4.2 and 4.1 respectively.

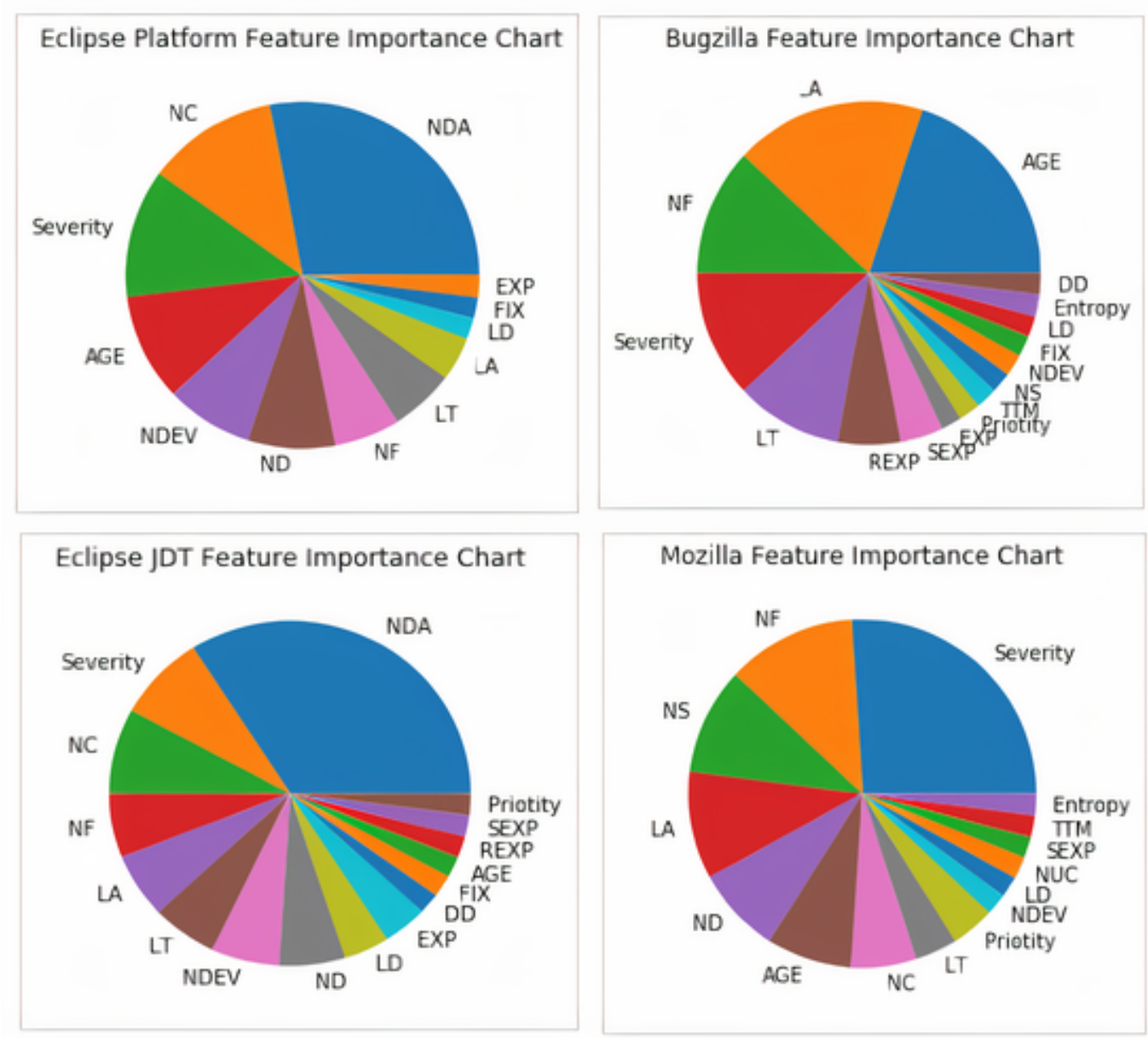

Figure 6.1: Feature importance graph using AdaBoost algorithm

To see which of the change request-based metrics contributed more to the improvement of JIT defect prediction, we ranked the metrics using AdaBoost feature importance algorithm. The results show that severity, number of developers assigned (NDA) and number of comments (NC) are among the top important change request-based metrics in at least two of the four systems. time to fix (TTM), depth of discussion (DD) and Priority , on the other hand, are found to be among the least important features.

6.2 Discussion

All sets of experiments conducted using different algorithms and datasets show the potential of software defect prediction metrics collected from the issue tracking system towards enhancing *just-in-time* bug prediction. By doing empirical experiments, we come up with three basic findings. The first one is, we practically showed that the change-request based metrics have the potential to enhance performance of predictive models for the JIT defect prediction approach. Secondly, we experimentally examine to identify the best performing algorithm among the five algorithms which are used in this research work. The results show that AdaBoost gives the highest delta improvement in JIT defect prediction.

Regarding the potential of enhancing the performance of the JIT defect prediction scheme, the change request-based metrics show encouraging potential to achieve this goal. Among a total of 20 test cases, 19 cases show that the change request-based metrics improves the performance of the prediction in terms of F1-score. The augmented dataset was capable of improving the baseline results up to 7.57% in accuracy, 7.86% in F1-score. Furthermore, this approach even was capable to enhance the true negative rate up to 9.23%. As the empirical results show, our approach also was capable of decreasing the false positive rates by 9.23% and false negative rate by 4.94%. From these results we can come to conclusion that the information captured in the issue tracking systems help to improve *just-in-time* defect prediction.

It is clear that one of our objectives of conducting the experiment is to figure out the better algorithm which is helpful to get insight for further implementation suggestions. In this regard, we compare the results of the five commonly used classification algorithms i.e., AdaBoost, XGBoost, DNN, Random Forest and Logistic Regression in terms of their accuracy, f1-score, TNR, FPR, FNR and AUC_ROC. As the results of the experiments show, among the five algorithms the one which shows the top performance in most of the evaluation metrics.

Considering the other algorithms used in this research work, it is difficult to find out the second best performing algorithm as the results are sporadic and do not show a clear pattern. If the comparison is deemed to be essential, it is necessary to define a clear sifting mechanism and objective. In that case, it might be possible to compare them by using some sort of criteria which are the subset of the aforementioned evaluation metrics.

Our empirical results also show that, in most of the cases, the least preforming algorithm is logistic

regression. From this it is possible to conclude that, the performance of *Just-In-time* defect prediction scheme is not only dependent on the quality of the data used for prediction but also dependent on the nature of the algorithm.

Regarding identifying the metrics which are contributing more to the prediction task, the experiments show that metrics from both the change and change request-based metrics are among those features with higher contribution level (level of importance). As the sets of charts drawn to show the importance of each metrics in a dataset depict, from the change request metrics, *number of comments (NC), severity* and *number of developers assigned (NDA)* are found to be among the highly important features of the aforementioned defect prediction scheme. *time to fix (TTM), depth of discussion (DD)* and *Priority* are found among those features with low importance levels.

Chapter 7

Conclusion and Recommendation

7.1 Conclusion

Just-in-time (JIT) defect prediction approaches use change-based metrics to predict defective commits at check-in time. Change-based metrics usually capture code and code change related information. A change's proneness to introducing new defects, however, could also be related to the nature of the original change request. In this paper, we propose to augment the existing change-based metrics with six change request-based metrics to enhance the performance of JIT defect prediction. The change request-based metrics extracted from the issue tracking systems are *severity, priory, time taken to complete the change, number of comments, the depth of discussion, and number of developers assigned.*

To assess the impact of the change request-based metrics on JIT defect prediction, we conducted an experiment using models built with five commonly used machine learning algorithms: AdaBoost, XGBoost, Deep Neural network, Random Forest and Logistic Regression. The experiment compares the performance of the change metrics-based dataset with the augmented dataset containing both change and change request-based metrics. The metrics are collected from four open-source software systems: Bugzilla, Mozilla, Eclipse JDT and Eclipse Platform.

The results show that the augmented dataset improves the performance of JIT defect prediction in 19 out of 20 cases. F1-score of JIT defect prediction in the four systems is improved by an average of 4.8%, 3.4%, 1.7%, 1.1% and 1.1% while using AdaBoost, XGBoost, Deep Neural Network, Random Forest and Logistic Regression, respectively

From the results of our experiment, we also conclude that change request-based metrics could improve the performance of JIT defect prediction. The change request-based metrics such as *number of comments (NC), severity* and *number of developers assigned (NDA)* are among the top important features towards accomplishing JIT defect prediction. Finally, among the five algorithms used in this research work, AdaBoost performs better towards providing the highest delta than the other algorithms in most of the experiment.

7.2 Future Works

The results of our study show that change request-based metrics help to improve JIT defect prediction. We believe further investigating additional change request-based and related metrics could further help to improve the performance of JIT defect prediction scheme. Taking this into consideration, the following research works are planned to be conducted in the future:

- We propose to fine-tune the data obtained from the issue tracking systems to boost the prediction performance of these metrics.
- We recommend to examine the impacts of issue tracking based metrics by combining them with other related metrics like popularity metrics.
- We suggest to investigate the option to come up with a more robust model by applying advanced ensembling techniques.

Bibliography

[1] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[2] S. Albahli, "A deep ensemble learning method for effort-aware just-in-time defect prediction," *Future Internet*, vol. 11, no. 12, p. 246, 2019.

[3] L. Qiao and Y. Wang, "Effort-aware and just-in-time defect prediction with neural network," *PloS one*, vol. 14, no. 2, p. e0211359, 2019.

[4] M. B. R. Pandit and N. Varma, "A deep introduction to ai based software defect prediction (sdp) and its current challenges," in *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*. IEEE, 2019, pp. 284–290.

[5] M. W. Thant and N. T. T. Aung, "Software defect prediction using hybrid approach," in *2019 International Conference on Advanced Information Technologies (ICAIT)*. IEEE, 2019, pp. 262–267.

[6] K. Punitha and S. Chitra, "Software defect prediction using software metrics-a survey," in *2013 International Conference on Information Communication and Embedded Systems (ICICES)*. IEEE, 2013, pp. 555–558.

[7] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok, "Recent catastrophic accidents: Investigating how software was responsible," in *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*. IEEE, 2010, pp. 14–22.

[8] R. Skeel, "Roundoff error and the patriot missile," *SIAM News*, vol. 25, no. 4, p. 11, 1992.

[9] R. Malhotra, L. Bahl, S. Sehgal, and P. Priya, "Empirical comparison of machine learning algorithms for bug prediction in open source software," in *2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*. IEEE, 2017, pp. 40–45.

[10] D. Sharma and P. Chandra, "Efficient fault prediction using exploratory and causal techniques," in *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*. IEEE, 2018, pp. 193–197.

[11] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, 2020.

[12] F. Akiyama, "An example of software system debugging," in *Proceedings. IFIP Congress*, 1971, pp. 353–8.

[13] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised vs. supervised models," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 344–353.

[14] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A developer centered bug prediction model," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 5–24, 2017.

[15] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019.

[16] F. Aktaş and F. Buzluca, "A learning-based bug predicition method for object-oriented systems," in *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*. IEEE, 2018, pp. 217–223.

[17] J. Ferzund, S. N. Ahsan, and F. Wotawa, "Software change classification using hunk metrics," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 471–474.

[18] M. Hanafi and A. Abdel-Raouf, "Software maintenance from the change theory perspective," *Proceedings of the 2014 International CSSCC*, pp. 66–71, 2014.

[19] K. Herzig, S. Just, A. Rau, and A. Zeller, "Predicting defects using change genealogies," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 118–127.

[20] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[21] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[22] P. Singh and S. Verma, "Empirical investigation of fault prediction capability of object oriented metrics of open source software," in *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 2012, pp. 323–327.

[23] G. M. Ubayawardana and D. D. Karunaratna, "Bug prediction model using code smells," in *2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer)*. IEEE, 2018, pp. 70–77.

[24] K. Kaur and P. Kaur, "Evaluation of sampling techniques in software fault prediction using metrics and code smells," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2017, pp. 1377–1387.

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

[26] C. Aggrawal, "Neural networks and deep learning: A textbook," 2018.

[27] R. E. Schapire, "Explaining adaboost," in *Empirical inference*. Springer, 2013, pp. 37–52.

[28] A. Gupta, S. Sharma, S. Goyal, and M. Rashid, "Novel xgboost tuned machine learning model for software bug prediction," in *2020 International Conference on Intelligent Engineering and Management (ICIEM)*. IEEE, 2020, pp. 376–380.

[29] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[30] S. Misra, H. Li, and J. He, *Machine Learning for Subsurface Characterization*, K. Hammon, Ed. Gulf Professional Publishing, 2020.

[31] R. Wang, "Adaboost for feature selection, classification and its relation with svm, a review," *Physics Procedia*, vol. 25, pp. 800–807, 2012.

[32] P. L. Bartlett and M. Traskin, "Adaboost is consistent," *Journal of Machine Learning Research*, vol. 8, no. Oct, pp. 2347–2368, 2007.

[33] H. Osman, M. Ghafari, and O. Nierstrasz, "Hyperparameter optimization to improve bug prediction accuracy," in *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2017, pp. 33–38.

[34] Y. A. Alshehri, K. Goseva-Popstojanova, D. G. Dzielski, and T. Devine, "Applying machine learning to predict software fault proneness using change metrics, static code metrics, and a combination of them," in *SoutheastCon 2018*. IEEE, 2018, pp. 1–7.

[35] A. A. Al-Jadaa, "Software metrics," *None*, None.

[36] J. Nam, "Survey on software defect prediction," *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[37] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, 2018.

[38] X. Yang, H. Yu, G. Fan, K. Shi, and L. Chen, "Local versus global models for just-in-time software defect prediction," *Scientific Programming*, vol. 2019, 2019.

[39] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?" in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2010, pp. 59–73.

[40] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 4–14.

[41] K. Beck, M. Fowler, and G. Beck, "Bad smells in code," *Refactoring: Improving the design of existing code*, vol. 1, pp. 75–88, 1999.

[42] G. Catolino, F. Palomba, F. A. Fontana, A. De Lucia, A. Zaidman, and F. Ferrucci, "Improving change prediction models with code smell-related information," *Empirical Software Engineering*, vol. 25, no. 1, pp. 49–95, 2020.

[43] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Deep semantic feature learning with embedded static metrics for software defect prediction," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 244–251.

[44] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic lstm model for software defect prediction," *IEEE Access*, vol. 7, pp. 83 812–83 824, 2019.

[45] K. Muthukumaran, A. Choudhary, and N. B. Murthy, "Mining github for novel change metrics to predict buggy files in software systems," in *2015 International Conference on Computational Intelligence and Networks*. IEEE, 2015, pp. 15–20.

[46] J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," *IEEE Transactions on nuclear science*, vol. 44, no. 3, pp. 1464–1468, 1997.

[47] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *Journal of Big Data*, vol. 6, no. 1, p. 27, 2019.

[48] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[49] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 17–26.

[50] F. Melo, *Receiver Operating Characteristic (ROC) Curve*. New York, NY: Springer New York, 2013, pp. 1818–1823. [Online]. Available: https://doi.org/10.1007/978-1-4419-9863-7_242

Appendices

Chapter A

Program Source Codes

In this section we put the fragment of the python source code that are written to accomplish different parts of this thesis work. One important thing that requires a due consideration is that in order to save space, we exclude the entire import statements from the codes that implement the machine learning algorithms and compute the importance of features. Whenever testing of codes is required, the first step must be including the required import statements into the fragment codes which are given below.

A.1 Machine Learning Models Fragment Codes

A.1.1 AdaBoost Model Python Code

```
xt=pd.read_csv('/home/user/Documents/JN/FinalDataForAugmentation/JDT_Augmenteted_Clean_Data_F.csv
bbb=np.transpose([np.log(xt.iloc[:,2]+1), np.log(xt.iloc[:,3]+1),
    np.log(xt.iloc[:,4]+1), np.log(xt.iloc[:,5]+1), np.log(xt.iloc[:,6]+1),
    np.log(xt.iloc[:,7]+1), np.log(xt.iloc[:,8]+1), xt.iloc[:,9],
    np.log(xt.iloc[:,10]+1),np.log(xt.iloc[:,11]+1),np.log(xt.iloc[:,12]+1),
   np.log(xt.iloc[:,13]+1),np.log(xt.iloc[:,14]+1),np.log(xt.iloc[:,15]+1),
   np.log(xt.iloc[:,16]+1),np.log(xt.iloc[:,17]+1),np.log(xt.iloc[:,18]+1),
   np.log(xt.iloc[:,19]+1),np.log(xt.iloc[:,20]+1),np.log(xt.iloc[:,21]+1),
       xt.iloc[:,22]])
bab=pd.DataFrame(bbb)
baab=bab[np.isfinite(bab).all(1)]
scaler = MinMaxScaler(feature_range=(0, 1))
rescaled = scaler.fit_transform(baab)
rs=pd.DataFrame(rescaled)
xs=rs.iloc[:,0:20]
df=pd.DataFrame(xs)
df = df.reset_index()
```

```
df.isna()
ys=rs.iloc[:,20]
dff=pd.DataFrame(ys)
oversample = SMOTE(sampling_strategy='minority')
X, Y = oversample.fit_resample(xs, ys)
seed = 31
numpy.random.seed(seed)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
cvscores = []
a_s=[]
pre=[]
rec=[]
f1_S=[]
roc_auc_c=[]
tpr_r=[]
fpr_r=[]
tnr_r=[]
fnr_r=[]
for train, test in kfold.split(X, Y):
    abc = AdaBoostClassifier(n_estimators=600, learning_rate=1)
    model = abc.fit(X[train], Y[train])
    y_pred = model.predict(X[test])
    report1 = classification_report(Y[test], y_pred)
    print(report1)
    plt.rcParams["figure.figsize"] = (5,5)
    cnf_matrix = metrics.confusion_matrix(Y[test], y_pred)
    p=np.round(model.predict(X[test]))
    accuracy=accuracy_score(Y[test],p)
    precision=precision_score(Y[test], p)
    recall=recall_score(Y[test], p)
    tnr=recall_score(Y[test], p, pos_label=0)
    tpr=recall_score(Y[test], p)
    fpr=1-tnr
    fnr=1-tpr
```

```
    f1s=f1_score(Y[test], p)
    roc_auc=roc_auc_score(Y[test], p)
    print("%s: %.2f%%" % ('acc', accuracy*100))
    print("%s: %.2f%%" % ('f1s',f1s*100))
    print("%s: %.2f%%" % ('p',precision*100))
    print("%s: %.2f%%" % ('rec',recall*100))
    print(roc_auc*100)
    print(". . . . . . . . ")
    print("%s: %.2f%%" % ('tpr',tpr*100))
    print("%s: %.2f%%" % ('fpr',fpr*100))
    print("%s: %.2f%%" % ('tnr',tnr*100))
    print("%s: %.2f%%" % ('fnr',fnr*100))
    a_s.append(accuracy*100)
    pre.append(precision*100)
    rec.append( recall*100)
    f1_S.append(f1s*100)
    roc_auc_c.append(roc_auc*100)
    tpr_r.append(tpr*100)
    fpr_r.append(fpr*100)
    tnr_r.append(tnr*100)
    fnr_r.append(fnr*100)
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(a_s), numpy.std(a_s)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(pre), numpy.std(pre)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(rec), numpy.std(rec)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(f1_S), numpy.std(f1_S)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(roc_auc_c), numpy.std(roc_auc_c)))
print('TPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tpr_r), numpy.std(tpr_r)))
print('FPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fpr_r), numpy.std(fpr_r)))
print('TNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tnr_r), numpy.std(tnr_r)))
print('FNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fnr_r), numpy.std(fnr_r)))
```

A.1.2 XGBoost Model Python Code

```
xt=pd.read_csv('/home/user/Documents/JN/FinalDataForAugmentation/JDT_Augmenteted_Clean_Data_F.csv
bbb=np.transpose([np.log(xt.iloc[:,2]+1), np.log(xt.iloc[:,3]+1),
    np.log(xt.iloc[:,4]+1), np.log(xt.iloc[:,5]+1), np.log(xt.iloc[:,6]+1),
    np.log(xt.iloc[:,7]+1), np.log(xt.iloc[:,8]+1), xt.iloc[:,9],
    np.log(xt.iloc[:,10]+1),np.log(xt.iloc[:,11]+1),np.log(xt.iloc[:,12]+1),
   np.log(xt.iloc[:,13]+1),np.log(xt.iloc[:,14]+1),np.log(xt.iloc[:,15]+1),
   np.log(xt.iloc[:,16]+1),np.log(xt.iloc[:,17]+1),np.log(xt.iloc[:,18]+1),
   np.log(xt.iloc[:,19]+1),np.log(xt.iloc[:,20]+1),np.log(xt.iloc[:,21]+1),
       xt.iloc[:,22]])
bab=pd.DataFrame(bbb)
baab=bab[np.isfinite(bab).all(1)]
scaler = MinMaxScaler(feature_range=(0, 1))
rescaled = scaler.fit_transform(baab)
rs=pd.DataFrame(rescaled)
xs=rs.iloc[:,0:20]
df=pd.DataFrame(xs)
df = df.reset_index()
df.isna()
ys=rs.iloc[:,20]
dff=pd.DataFrame(ys)
oversample = SMOTE(sampling_strategy='minority')
X, Y = oversample.fit_resample(xs, ys)
seed = 8
numpy.random.seed(seed)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
cvscores = []
a_s=[]
pre=[]
rec=[]
f1_S=[]
los=[]
roc_auc_c=[]
```

```
tpr_r=[]
fpr_r=[]
tnr_r=[]
fnr_r=[]
for train, test in kfold.split(X, Y):
    xg_reg = xgb.XGBRegressor(objective ='reg:linear', colsample_bytree = 0.3,
        learning_rate = 0.001,
                max_depth = 200, alpha = 1, n_estimators = 600)
    xg_reg.fit(X[train],Y[train])
    predicted=np.round(xg_reg.predict(X[test]))
    report1 = classification_report(Y[test], predicted)
    print(report1)
    plt.rcParams["figure.figsize"] = (5,5)
    cnf_matrix = metrics.confusion_matrix(Y[test], predicted)
    p=np.round(xg_reg.predict(X[test]))
    accuracy=accuracy_score(Y[test],p)
    precision=precision_score(Y[test], p)
    recall=recall_score(Y[test], p)
    f1s=f1_score(Y[test], p)
    roc_auc=roc_auc_score(Y[test], p)
    tnr=recall_score(Y[test], p, pos_label=0)
    tpr=recall_score(Y[test], p)
    fpr=1-tnr
    fnr=1-tpr
    print("%s: %.2f%%" % ('acc', accuracy*100))
    print("%s: %.2f%%" % ('f1s',f1s*100))
    print("%s: %.2f%%" % ('p',precision*100))
    print("%s: %.2f%%" % ('rec',recall*100))
    print(roc_auc*100)
    print(". . . . . . . . ")
    print("%s: %.2f%%" % ('tpr',tpr*100))
    print("%s: %.2f%%" % ('fpr',fpr*100))
    print("%s: %.2f%%" % ('tnr',tnr*100))
    print("%s: %.2f%%" % ('fnr',fnr*100))
```

```
    a_s.append(accuracy*100)
    pre.append(precision*100)
    rec.append( recall*100)
    f1_S.append(f1s*100)
    roc_auc_c.append(roc_auc*100)
    tpr_r.append(tpr*100)
    fpr_r.append(fpr*100)
    tnr_r.append(tnr*100)
    fnr_r.append(fnr*100)
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(a_s), numpy.std(a_s)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(pre), numpy.std(pre)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(rec), numpy.std(rec)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(f1_S), numpy.std(f1_S)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(roc_auc_c), numpy.std(roc_auc_c)))
print('TPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tpr_r), numpy.std(tpr_r)))
print('FPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fpr_r), numpy.std(fpr_r)))
print('TNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tnr_r), numpy.std(tnr_r)))
print('FNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fnr_r), numpy.std(fnr_r)))
```

A.1.3 Random Forest Model Python Code

```
xt=pd.read_csv('/home/user/Documents/JN/FinalDataForAugmentation/JDT_Augmenteted_Clean_Data_F.csv
bbb=np.transpose([np.log(xt.iloc[:,2]+1), np.log(xt.iloc[:,3]+1),
    np.log(xt.iloc[:,4]+1), np.log(xt.iloc[:,5]+1), np.log(xt.iloc[:,6]+1),
    np.log(xt.iloc[:,7]+1), np.log(xt.iloc[:,8]+1), xt.iloc[:,9],
    np.log(xt.iloc[:,10]+1),np.log(xt.iloc[:,11]+1),np.log(xt.iloc[:,12]+1),
   np.log(xt.iloc[:,13]+1),np.log(xt.iloc[:,14]+1),np.log(xt.iloc[:,15]+1),
   np.log(xt.iloc[:,16]+1),np.log(xt.iloc[:,17]+1),np.log(xt.iloc[:,18]+1),
   np.log(xt.iloc[:,19]+1),np.log(xt.iloc[:,20]+1),np.log(xt.iloc[:,21]+1),
       xt.iloc[:,22]])
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
rescaled = scaler.fit_transform(baab)
rs=pd.DataFrame(rescaled)
rs.hist()
rs.shape
xs=rs.iloc[:,0:20]
df=pd.DataFrame(xs)
df = df.reset_index()
df.isna()
ys=rs.iloc[:,20]
dff=pd.DataFrame(ys)
oversample = SMOTE(sampling_strategy='minority')
X, Y = oversample.fit_resample(xs, ys)
seed = 8
numpy.random.seed(seed)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
cvscores = []
a_s=[]
pre=[]
rec=[]
f1_S=[]
los=[]
roc_auc_c=[]
tpr_r=[]
fpr_r=[]
tnr_r=[]
fnr_r=[]
for train, test in kfold.split(X, Y):
    RF = RandomForestClassifier(n_estimators=600, max_depth=200, random_state=0)
    RF.fit(X[train], Y[train])
    predicted=RF.predict(X[test])
    report1 = classification_report(Y[test], predicted)
    print(report1)
    plt.rcParams["figure.figsize"] = (5,5)
```

```
    cnf_matrix = metrics.confusion_matrix(Y[test], predicted)
    p=RF.predict(X[test])
    accuracy=accuracy_score(Y[test],p)
    precision=precision_score(Y[test], p)
    recall=recall_score(Y[test], p)
    f1s=f1_score(Y[test], p)
    roc_auc=roc_auc_score(Y[test], p)
    tnr=recall_score(Y[test], p, pos_label=0)
    tpr=recall_score(Y[test], p)
    fpr=1-tnr
    fnr=1-tpr
    print("%s: %.2f%%" % ('acc', accuracy*100))
    print("%s: %.2f%%" % ('f1s',f1s*100))
    print("%s: %.2f%%" % ('p',precision*100))
    print("%s: %.2f%%" % ('rec',recall*100))
    print(roc_auc*100)
    a_s.append(accuracy*100)
    pre.append(precision*100)
    rec.append( recall*100)
    f1_S.append(f1s*100)
    roc_auc_c.append(roc_auc*100)
    tpr_r.append(tpr*100)
    fpr_r.append(fpr*100)
    tnr_r.append(tnr*100)
    fnr_r.append(fnr*100)
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(a_s), numpy.std(a_s)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(pre), numpy.std(pre)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(rec), numpy.std(rec)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(f1_S), numpy.std(f1_S)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(roc_auc_c), numpy.std(roc_auc_c)))
print('TPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tpr_r), numpy.std(tpr_r)))
print('FPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fpr_r), numpy.std(fpr_r)))
```

```
print('TNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tnr_r), numpy.std(tnr_r)))
print('FNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fnr_r), numpy.std(fnr_r)))
```

A.1.4 DNN Model Python Code

```
xt=pd.read_csv('/home/user/Documents/JN/FinalDataForAugmentation/JDT_Augmenteted_Clean_Data_F.csv
bbb=np.transpose([np.log(xt.iloc[:,2]+1), np.log(xt.iloc[:,3]+1),
    np.log(xt.iloc[:,4]+1), np.log(xt.iloc[:,5]+1), np.log(xt.iloc[:,6]+1),
    np.log(xt.iloc[:,7]+1), np.log(xt.iloc[:,8]+1), xt.iloc[:,9],
    np.log(xt.iloc[:,10]+1),np.log(xt.iloc[:,11]+1),np.log(xt.iloc[:,12]+1),
   np.log(xt.iloc[:,13]+1),np.log(xt.iloc[:,14]+1),np.log(xt.iloc[:,15]+1),
   np.log(xt.iloc[:,16]+1),np.log(xt.iloc[:,17]+1),np.log(xt.iloc[:,18]+1),
   np.log(xt.iloc[:,19]+1),np.log(xt.iloc[:,20]+1),np.log(xt.iloc[:,21]+1),
       xt.iloc[:,22]])
bab=pd.DataFrame(bbb)
baab=bab[np.isfinite(bab).all(1)]
scaler = MinMaxScaler(feature_range=(0, 1))
rescaled = scaler.fit_transform(baab)
rs=pd.DataFrame(rescaled)
xs=rs.iloc[:,0:20]
df=pd.DataFrame(xs)
df = df.reset_index()
df.isna()
ys=rs.iloc[:,20]
dff=pd.DataFrame(ys)
a=pd.DataFrame(xs)
oversample = SMOTE(sampling_strategy='minority')
X, Y = oversample.fit_resample(xs, ys)
seed = 33
numpy.random.seed(seed)
kfold = StratifiedKFold(n_splits=3, shuffle=True, random_state=seed)
cvscores = []
```

```
a_s=[]
pre=[]
rec=[]
f1_S=[]
los=[]
roc_auc_c=[]
tpr_r=[]
fpr_r=[]
tnr_r=[]
fnr_r=[]
for train, test in kfold.split(X, Y):
    model = Sequential()
    model.add(Dense(20, input_dim=20, init= 'uniform', activation= 'relu' ))
    model.add(Dense(28, init= 'uniform' , activation= 'relu' ))
    model.add(Dense(28, init= 'uniform' , activation= 'relu' ))
    model.add(Dense(28, init= 'uniform' , activation= 'relu' ))
    model.add(Dense(1, init= 'uniform' , activation= 'sigmoid' ))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])
    model.fit(X[train], Y[train], epochs=100, batch_size=64, verbose=0)
    y_pred_1 = model.predict(X[test], batch_size=1, verbose=0)
    y_pred_bool_1 = np.round(y_pred_1)
    accuracy=accuracy_score(Y[test], y_pred_bool_1)
    precision=precision_score(Y[test], y_pred_bool_1)
    recall=recall_score(Y[test], y_pred_bool_1)
    f1s=f1_score(Y[test], y_pred_bool_1)
    roc_auc=roc_auc_score(Y[test], y_pred_bool_1)
    scores = model.evaluate(X[test], Y[test], verbose=0)
    tnr=recall_score(Y[test], y_pred_bool_1, pos_label=0)
    tpr=recall_score(Y[test], y_pred_bool_1)
    fpr=1-tnr
    fnr=1-tpr
    print("%s: %.2f%%" % ('acc', accuracy*100))
    print("%s: %.2f%%" % ('f1s',f1s*100))
    print("%s: %.2f%%" % ('p',precision*100))
```

```
    print("%s: %.2f%%" % ('rec',recall*100))
    print(roc_auc*100)
    a_s.append(accuracy*100)
    pre.append(precision*100)
    rec.append( recall*100)
    f1_S.append(f1s*100)
    roc_auc_c.append(roc_auc*100)
    tpr_r.append(tpr*100)
    fpr_r.append(fpr*100)
    tnr_r.append(tnr*100)
    fnr_r.append(fnr*100)
    print(classification_report(Y[test], y_pred_bool_1))
    print(confusion_matrix(Y[test], y_pred_bool_1))
    df_cccc=pd.DataFrame(y_pred_bool_1)
    df_cccc.to_csv('/home/user/Desktop/ppp/new_vddd'+str(i)+'.csv')
    print('---------------
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(a_s), numpy.std(a_s)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(pre), numpy.std(pre)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(rec), numpy.std(rec)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(f1_S), numpy.std(f1_S)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(roc_auc_c), numpy.std(roc_auc_c)))
print('TPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tpr_r), numpy.std(tpr_r)))
print('FPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fpr_r), numpy.std(fpr_r)))
print('TNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tnr_r), numpy.std(tnr_r)))
print('FNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fnr_r), numpy.std(fnr_r)))
```

A.1.5 Logistic Regression

```
 xt=pd.read_csv('/home/user/Documents/JN/FinalDataForAugmentation/JDT_Augmenteted_Clean_Data_F.cs
bbb=np.transpose([np.log(xt.iloc[:,2]+1), np.log(xt.iloc[:,3]+1),
```

```
    np.log(xt.iloc[:,4]+1), np.log(xt.iloc[:,5]+1), np.log(xt.iloc[:,6]+1),
    np.log(xt.iloc[:,7]+1), np.log(xt.iloc[:,8]+1), xt.iloc[:,9],
    np.log(xt.iloc[:,10]+1),np.log(xt.iloc[:,11]+1),np.log(xt.iloc[:,12]+1)
   ,np.log(xt.iloc[:,13]+1),np.log(xt.iloc[:,14]+1),np.log(xt.iloc[:,15]+1),
   np.log(xt.iloc[:,16]+1),np.log(xt.iloc[:,17]+1),np.log(xt.iloc[:,18]+1)
   ,np.log(xt.iloc[:,19]+1),np.log(xt.iloc[:,20]+1),np.log(xt.iloc[:,21]+1),
       xt.iloc[:,22]])
bab=pd.DataFrame(bbb)
baab=bab[np.isfinite(bab).all(1)]
scaler = MinMaxScaler(feature_range=(0, 1))
rescaled = scaler.fit_transform(baab)
rs=pd.DataFrame(rescaled)
xs=rs.iloc[:,0:14]
df=pd.DataFrame(xs)
df = df.reset_index()
df.isna()
ys=rs.iloc[:,20]
dff=pd.DataFrame(ys)
under= RandomUnderSampler(sampling_strategy='majority')
X,Y=under.fit_resample(xs,ys)
seed = 8
numpy.random.seed(seed)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
cvscores = []
a_s=[]
pre=[]
rec=[]
f1_S=[]
los=[]
roc_auc_c=[]
tpr_r=[]
fpr_r=[]
tnr_r=[]
fnr_r=[]
```

```
i=0
for train, test in kfold.split(X, Y):
    model = LogisticRegression()
    model.fit(X[train], Y[train])
    predicted=model.predict(X[test])
    report1 = classification_report(Y[test], predicted)
    print(report1)
    plt.rcParams["figure.figsize"] = (5,5)
    cnf_matrix = metrics.confusion_matrix(Y[test], predicted)
    p=model.predict(X[test])
    accuracy=accuracy_score(Y[test],p)
    precision=precision_score(Y[test], p)
    recall=recall_score(Y[test], p)
    f1s=f1_score(Y[test], p)
    roc_auc=roc_auc_score(Y[test], p)
    tnr=recall_score(Y[test], p, pos_label=0)
    tpr=recall_score(Y[test], p)
    fpr=1-tnr
    fnr=1-tpr
    print("%s: %.2f%%" % ('acc', accuracy*100))
    print("%s: %.2f%%" % ('f1s',f1s*100))
    print("%s: %.2f%%" % ('p',precision*100))
    print("%s: %.2f%%" % ('rec',recall*100))
    print(roc_auc*100)
    a_s.append(accuracy*100)
    pre.append(precision*100)
    rec.append( recall*100)
    f1_S.append(f1s*100)
    roc_auc_c.append(roc_auc*100)
    tpr_r.append(tpr*100)
    fpr_r.append(fpr*100)
    tnr_r.append(tnr*100)
    fnr_r.append(fnr*100)
```

```
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(a_s), numpy.std(a_s)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(pre), numpy.std(pre)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(rec), numpy.std(rec)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(f1_S), numpy.std(f1_S)))
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(roc_auc_c), numpy.std(roc_auc_c)))
print('TPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tpr_r), numpy.std(tpr_r)))
print('FPR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fpr_r), numpy.std(fpr_r)))
print('TNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(tnr_r), numpy.std(tnr_r)))
print('FNR')
print("%.2f%% (+/- %.2f%%)" % (numpy.mean(fnr_r), numpy.std(fnr_r)))
```

A.2 Feature Importance Fragment Codes

A.2.1 Feature Importance Using AdaBoost

```
# xx represents the preprocessed feature and yy represent the output class of the data
xx=pd.DataFrame(X)
yy=pd.DataFrame(Y)
plt.title('Eclipse JDT Feature Importance Chart')
xg_reg = AdaBoostClassifier(n_estimators=50, learning_rate=1)
xg_reg.fit(xx, yy)
(pd.Series(xg_reg.feature_importances_, index=xx.columns)
   .nlargest(20)
   .plot(kind='pie'))
```

A.3 Metrics Extraction (Web Scrapping) Sample Code

```
import bs4
from urllib.request import urlopen as uReq
from bs4 import BeautifulSoup as soup
import pandas as pd
import datetime
from datetime import timedelta
import csv
#platform_bid=[4376, 4365, 4086, 4283, 4354, 4385, 1531, 4916, 4358, 4951, 3309, 3452,
    1756, 4972]
numberOfDeveAssig=1;
CSVFileName="jdtAllInOneAgain"
csv_writer=csv.writer(open(CSVFileName,'w'))
csvHeader=['bid','timeToFix', 'priotity','severity', 'nDevAInHis', 'NoIE', 'lenDes']
csv_writer.writerow(csvHeader)
print("BID      TTF(D)       priority severity  NDevAssHis NoIE    lenDes")

for i in range(len(platform_bid)):

   #developer assigned in the time span
   numberOfDeveAssig=1;
   csvData=[];
   #-----------to scrapp the bug activity page ------------------
   bugActivityUrl='https://bugs.eclipse.org/bugs/show_activity.cgi?id='+str(platform_bid[i])
   bugActivityClient = uReq(bugActivityUrl)
   bugActivityHtml=bugActivityClient.read()
   bugActivityClient.close()
   #page_soup = soup(page_html, "html.parser")
   bugActivityPageSoup = soup(bugActivityHtml, 'lxml')

   for tr in bugActivityPageSoup.find_all('tr'):
      data=[]
```

```
    for th in tr.find_all('th'):
        data.append(th.text)
    for td in tr.find_all('td'):
        data.append(td.text.strip())
    if(data):
        x=str(data).split()
        for count in range(len(x)):
            x[count]=x[count].replace('\'','').replace('[','').replace('\"','').replace('\n','').r
        for na in range (len(x)):
            if(x[na]=="Assignee"):
                numberOfDeveAssig=numberOfDeveAssig+1
#---------------to scrapp the main bug page----------------------
mainPageUrl='https://bugs.eclipse.org/bugs/show_bug.cgi?id='+str(platform_bid[i])
mainPageClient = uReq(mainPageUrl)
mainPageHtml=mainPageClient.read()
mainPageClient.close()
mainPageSoup = soup(mainPageHtml, 'lxml')
#-----------------------------------
r=mainPageSoup.find_all("td",{"id":"bz_show_bug_column_2"})[0].find("table")
pr=mainPageSoup.find_all("td",{"id":"bz_show_bug_column_1"})[0].find("table")
devAssig=mainPageSoup.find_all("table",{"id":"bug_activity"})
prioData=[]
pxx=[]
priority=0;
for ptr in pr.find_all('tr'):
    for pth in ptr.find_all('td'):
        prioData.append(pth.text.strip())
px=str(prioData).split()
for l in range (len(px)):
    pxx.append(px[l].replace('\'','').replace('[','').replace('\"','').replace('\n','').replace
for p in range(len(pxx)):
    if(pxx[p].startswith('P1')):
        priority=1
        break
```

```
elif (pxx[p].startswith('P2')):
   priority=2
   break
elif (pxx[p].startswith('P3')):
   priority=3
   break
elif (pxx[p].startswith('P4')):
   priority=4
   break
elif (pxx[p].startswith('P5')):
   priority=5
   break
else:
   priority=0
for p in range(len(pxx)):
   if(pxx[p].startswith('blo')):
      severity=7
      break
   elif (pxx[p].startswith('crit')):
      severity=6
      break
   elif (pxx[p].startswith('majo')):
      severity=5
      break
   elif (pxx[p].startswith('norm')):
      severity=4
      break
   elif (pxx[p].startswith('mino')):
      severity=3
      break
   elif (pxx[p].startswith('triv')):
      severity=2
      break
   elif (pxx[p].startswith('enha')):
```

```
            severity=1
            break
        else:
            severity=0
data=[]
for tr in r.find_all('tr'):
    for th in tr.find_all('td'):
        data.append(th.text.strip())
x=str(data).split()
    if(((x[3].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
    xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
        "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
    yy=x[3].replace('\'','').replace('[','').replace('\"','')+"
        "+x[4].replace('\'','').replace('[','').replace(' ','').replace('\"','')
elif(((x[4].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
    xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
        "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
    yy=x[4].replace('\'','').replace('[','').replace('\"','')+"
        "+x[5].replace('\'','').replace('[','').replace(' ','').replace('\"','')
elif(((x[5].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
    xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
        "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
    yy=x[5].replace('\'','').replace('[','').replace('\"','')+"
        "+x[6].replace('\'','').replace('[','').replace(' ','').replace('\"','')
elif(((x[6].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
    xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
        "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
    yy=x[6].replace('\'','').replace('[','').replace('\"','')+"
        "+x[7].replace('\'','').replace('[','').replace(' ','').replace('\"','')
elif(((x[7].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
    xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
        "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
    yy=x[7].replace('\'','').replace('[','').replace('\"','')+"
        "+x[8].replace('\'','').replace('[','').replace(' ','').replace('\"','')
```

```
elif(((x[8].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
   xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
       "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
   yy=x[8].replace('\'','').replace('[','').replace('\"','')+"
       "+x[9].replace('\'','').replace('[','').replace(' ','').replace('\"','')
elif(((x[9].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
   xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
       "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
   yy=x[9].replace('\'','').replace('[','').replace('\"','')+"
       "+x[10].replace('\'','').replace('[','').replace(' ','').replace('\"','')
elif(((x[10].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
   xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
       "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
   yy=x[10].replace('\'','').replace('[','').replace('\"','')+"
       "+x[11].replace('\'','').replace('[','').replace(' ','').replace('\"','')
elif(((x[11].replace('\'','').replace('[','').replace('\"','').startswith('20')))):
   xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
       "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
   yy=x[11].replace('\'','').replace('[','').replace('\"','')+"
       "+x[12].replace('\'','').replace('[','').replace(' ','').replace('\"','')
else:
   xx=x[0].replace('\'','').replace('[','').replace('\"','')+"
       "+x[1].replace('\'','').replace('[','').replace(' ','').replace('\"','')
   yy=x[6].replace('\'','').replace('[','').replace('\"','')+"
       "+x[7].replace('\'','').replace('[','').replace(' ','').replace('\"','')

datetimeFormat = '%Y-%m-%d %H:%M'
date1 = datetime.datetime.strptime(str(xx),datetimeFormat)
date2 = datetime.datetime.strptime(str(yy),datetimeFormat)
diff  = datetime.datetime.strptime(str(yy),
    datetimeFormat)-datetime.datetime.strptime(str(xx), datetimeFormat)

#print(str(pagesToBeScrapped[i])+"    "+str(yy)+"      "+str(xx)+"
    "+str(diff.days)+"          "+str(priority))
```

```
   #-----------code to extract the number of idea exchanges--------------
   ideaExchageHtmlElems=mainPageSoup.findAll("pre",{"class":"bz_comment_text"})
   #-----------code to extract the length of ideas (word count)---------
   numOfWords=""
   #listOfWords=ideaExchageHtmlElems.text.strip();
   for iter in range(len(ideaExchageHtmlElems)):
      numOfWords=numOfWords+ideaExchageHtmlElems[iter].text.strip()
   csvData.append(platform_bid[i])
   csvData.append(diff.days)
   csvData.append(priority)
   csvData.append(severity)
   csvData.append(numberOfDeveAssig)
   csvData.append(len(ideaExchageHtmlElems))
   csvData.append(len(numOfWords.split()))

   csv_writer.writerow(csvData)
   print(str(platform_bid[i])+"     "+str(diff.days)+"       "+str(priority)+"
       "+str(severity)+"      "+str(numberOfDeveAssig)+"
       "+str(len(ideaExchageHtmlElems))+"  "+str(len(numOfWords.split())))
#======================GAT,PAW,AD,DA,AD,MAG,HAG,MAG,GAG,ZAG,BAG-YARIMH==================#
```