Software fault prediction and test data generation using artificial intelligent techniques

Ph. D. Thesis

by

Yeresime Suresh

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela - 769008, India

August 2015

Software fault prediction and test data generation using artificial intelligent techniques

A dissertation submitted to the department of

Computer Science and Engineering

of

National Institute of Technology Rourkela

in partial fulfilment of the requirements

for the degree of

Doctor of Philosophy

by

Yeresime Suresh

(Roll No- 510CS102)

under the supervision of

Prof. Santanu Kumar Rath

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela - 769008, India

August 2015

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela - 769 008, India. www.nitrkl.ac.in

Dr. Santanu Kumar Rath
Professor & Head

August 7, 2015

Certificate

This is to certify that the work in the thesis entitled ***"Software fault prediction and test data generation using artificial intelligent techniques"*** by ***Yeresime Suresh***, bearing **Roll No: 510CS102**, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Doctor of Philosophy* in *Computer Science and Engineering*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

(Santanu Kumar Rath)

Acknowledgment

No work goes unfinished without thanksgiving to our beloved Teachers.

I take this opportunity to thank all those who have contributed in this journey.

I would like to express my sincere thanks to my supervisor Prof. Santanu Kumar Rath, for his valuable guidance, and encouragement during the course of this thesis. His eagerness and constant encouragement have instilled in me the confidence to complete this thesis. I am greatly indebted for his help throughout the thesis work.

I am very much indebted to Prof. Bansidhar Majhi, Chairman Doctoral Scrutiny Committee (DSC). I am also thankful to Prof. Durga Prasad Mohapatra, and all the DSC members, and faculty members of the department for their in time support, advise and encouragement.

I owe the heartfelt thanks to my parents Shri. Yeresime Komarappa and Smt. Yeresime Pushpavathi for their unconditional love, patience and cooperation during my research work. Also I would like to thank my younger sister Dr. Yeresime Surekha, who has been the constant source of inspiration to me.

Yeresime Suresh

Abstract

The complexity in requirements of the present-day software, which are often very large in nature has lead to increase in more number of lines of code, resulting in more number of modules. There is every possibility that some of the modules may give rise to varieties of defects, if testing is not done meticulously. In practice, it is not possible to carry out white box testing of every module of any software. Thus, software testing needs to be done selectively for the modules, which are prone to faults. Identifying the probable fault-prone modules is a critical task, carried out for any software. This dissertation, emphasizes on design of prediction and classification models to detect fault prone classes for object-oriented programs. Then, test data are generated for a particular task to check the functionality of the software product.

In the field of object-oriented software engineering, it is observed that Chidamber and Kemerer (CK) software metrics suite is more frequently used for fault prediction analysis, as it covers the unique aspects of object - oriented programming such as the complexity, data abstraction, and inheritance. It is observed that one of the most important goals of fault prediction is to detect fault prone modules as early as possible in the software development life cycle (SDLC). Numerous authors have used design and code metrics for predicting fault-prone modules. In this work, design metrics are used for fault prediction. In order to carry out fault prediction analysis, prediction models are designed using machine learning methods. Machine learning methods such as Statistical methods, Artificial neural network, Radial basis function network, Functional link artificial neural network, and Probabilistic neural network are deployed for fault prediction analysis. In the first phase, fault prediction is performed using the CK metrics suite. In the next phase, the reduced feature sets of CK metrics suite obtained by applying principal component analysis and rough set theory are used to perform fault prediction. A comparative approach is drawn to find a suitable prediction model among the set of designed models for fault prediction.

Prediction models designed for fault proneness, need to be validated for their efficiency. To achieve this, a cost-based evaluation framework is designed to evaluate the effectiveness of the designed fault prediction models. This framework, is based on the classification of classes as faulty or not-faulty. In this cost-based analysis, it is observed that fault prediction is found to be suitable where normalized estimated fault removal cost (NEcost) is less than certain threshold value. Also this indicated that any prediction model having NEcost greater than the threshold value are not suitable for fault prediction, and then further these classes are unit tested. All the prediction and classifier models used in the fault prediction analysis are applied on a case study viz.,

Apache Integration Framework (AIF). The metric data values are obtained from PROMISE repository and are mined using Chidamber and Kemerer Java Metrics (CKJM) tool.

Test data are generated for object-oriented program for withdrawal task in Bank ATM using three meta-heuristic search algorithms such as Clonal selection algorithm, Binary particle swarm optimization, and Artificial bee colony algorithm. It is observed that Artificial bee colony algorithm is able to obtain near optimal test data when compared to the other two algorithms. The test data are generated for withdrawal task based on the fitness function derived by using the *branch distance* proposed by Bogdan Korel. The generated test data ensure the proper functionality or the correctness of the programmed module in a software.

Keywords: branch distance, extended control flow graph, chidamber and kemerer metrics, fault prediction, fitness function, meta-heuristic, neural network, principal components, rough set, test data.

Contents

Certificate **iii**

Acknowledgment **iv**

Abstract **v**

List of Acronyms / Abbreviations **x**

List of Figures **xiii**

List of Tables **xv**

List of Symbols **xviii**

1 Introduction **1**
- 1.1 Motivation 4
- 1.2 Research Objectives 5
- 1.3 Summary of Contributions 5
- 1.4 Outline of the Thesis 6

2 Literature Review **8**
- 2.1 Fault prediction 9
  - 2.1.1 Observations 15
- 2.2 Test data generation for traditional methods 16
  - 2.2.1 Results and analysis 26
- 2.3 Summary 27

3 Effectiveness of Machine Learning Methods in Fault Prediction Analysis **29**
- 3.1 Introduction 30
- 3.2 Research Background 31
  - 3.2.1 Empirical Data Collection 31
  - 3.2.2 Data Normalization 33

3.2.3 Dependent and Independent Variables 33
3.3 Machine Learning Methods 33
3.3.1 Statistical Methods 34
3.3.2 Artificial Neural Network 36
3.3.3 Radial Basis Function Network 39
3.3.4 Functional Link Artificial Neural Network 42
3.3.5 Probabilistic Neural Network 43
3.4 Fault Prediction using Feature Reduction Techniques 44
3.4.1 Application of Principal Component Analysis 45
3.4.2 Application of Rough Set Theory 46
3.5 Performance Evaluation Parameters 46
3.6 Results and Analysis 49
3.6.1 Fault Data 49
3.6.2 Metrics Data 51
3.6.3 Descriptive Statistics and Correlation Analysis 52
3.6.4 Attribute Reduction 54
3.6.5 Machine Learning Methods 58
3.6.6 Comparison of Fault Prediction Models 75
3.6.7 Comparison with existing methods 78
3.7 Complexity analysis of prediction models 79
3.8 Threats to validity 81
3.9 Relation between fault prediction and test data generation . . . 82
3.10 Summary 83

4 Cost-Based Evaluation Framework for Software Fault Classification 84
4.1 Introduction 85
4.2 Cost-Based Evaluation Framework 86
4.2.1 Estimated fault removal cost (E_{cost}) 87
4.2.2 Estimated testing cost (T_{cost}) 89
4.2.3 Normalized fault removal cost (NE_{cost}) 90
4.3 Performance Evaluation Parameters 92
4.4 Results and Analysis 93
4.4.1 Neural network as a classifier 93
4.5 Summary 99

5 Test Data Generation for Object-Oriented Program using Meta-heuristic Search Algorithms 100
5.1 Introduction 101
5.2 Meta-heuristic Search Algorithms 101

5.2.1 Role of meta-heuristic search based algorithms in software testing . . . 102
5.2.2 Need for automated test data generation . . . 103
5.3 Extended Control Flow Graph (ECFG) . . . 103
5.3.1 ECFG features . . . 104
5.3.2 Cyclomatic complexity computation for ECFG . . . 105
5.4 Fitness Function based on Korel's Branch Distance Function . . . 110
5.4.1 Case study: Bank Automatic Teller Machine (ATM) . . . 112
5.5 Test Data Generation for Bank ATM . . . 114
5.5.1 Construction of CFG for an individual method . . . 115
5.5.2 Construction of ECFG . . . 116
5.5.3 Test data generation using meta-heuristic search algorithms . . . 117
5.6 Results . . . 121
5.6.1 Case study: Bank ATM . . . 122
5.6.2 Experimental settings . . . 123
5.6.3 Interpretation of results . . . 124
5.6.4 Code coverage analysis . . . 126
5.7 Summary . . . 127

6 Conclusions and future scope of work **128**
6.1 Future scope of work . . . 132

Bibliography **133**

Dissemination **149**

A Bank ATM pseudocode **151**

B Code Coverage Analysis **154**

List of Acronyms/Abbreviations

ABC	Artificial Bee Colony Optimization
ACO	Ant Colony Optimization
AE	Artificial Evolution
AIF	Apache Integration Framework
ANN	Artificial Neural Network
AI	Artificial Intelligence
ATM	Automatic Teller Machine
AMC	Average Method Complexity
AUC	Area Under the receiver operating characteristics Curve
BFA	Bacterial Foraging Algorithm
BPSO	Binary Particle Swarm Optimization
Ca	Afferent Coupling
CAE	Coupling on Advice Execution
CAM	Cohesion Among Methods of Class
CBM	Coupling Between Modules
CBO	Coupling Between Objects
CC	Cyclomatic Complexity
Ce	Efferent Coupling
CFG	Control Flow Graph
CK	Chidamber and Kemerer
CKJM	Chidamber and Kemerer Java Metrics Tool
CSA	Clonal Selection Algorithm
DAM	Data Access Metric
DFCT	Count of Defects per Class
DIT	Depth of Inheritance
EA	Evolutionary Algorithm
ECFG	Extended Control Flow Graph
E-CC	Extended Cyclomatic Complexity
Ecost	Estimated fault removal cost using fault prediction
EVNT	Count of Events per class in the state model
FFNN	Feed Forward Neural Network

FLANN	Functional Link Artificial Neural Network
FN	False Negative
FOUT	Number of Method Calls
FP	False Positive
GA	Genetic Algorithm
GSAA	Genetic Simulated Annealing Algorithm
IC	Inheritance Coupling
LCOM	Lack of Cohesion among Methods
LCOM3	Lack of Cohesion in Methods (Henderson-Sellers version)
LM	Levenberg Marquardt
LMS	Least Mean Square Algorithm
LMSE	Least Mean Square Error
LOC	Lines of Code
MAE	Mean Absolute Error
MARE	Mean Absolute Relative Error
Max	Maximum
MFA	Measure of Functional Abstraction
Min	Minimum
MLOC	Method Lines of Code
MOA	Measure of Aggregation
MOOD	Metrics for Object Oriented Design
MSE	Mean Square Error
NCM	Number of Class Methods
NEcost	Normalized Estimated fault removal cost
NIM	Number of Instance Methods
NLM	Number of Local Methods
NOC	Number of Children
NPM	Number of Public Methods
NTM	Number of Trivial Methods
PCA	Principal Component Analysis
PCs	Principal Components
PNN	Probabilistic Neural Network
PSO	Particle Swarm Optimization
QMOOD	Quality Model for Object Oriented Design
RBFN	Radial Basis Function Network
RFC	Response For Class
ROC	Receiver Operating Characteristic
RMSE	Root Mean Square Error
RST	Rough Set Theory
RWD	Read/Write/Deletes
SA	Simulated Annealing

SD	Standard Deviation
SDLC	Software Development Life Cycle
SDMC	Standard Deviation Method Complexity
SLOC	Source Lines of Code
SEM	Standard Error of the Mean
SVM	Support Vector Machine
Tcost	Estimated fault removal cost without using fault prediction
TN	True Negative
TP	True Positive
TS	Tabu Search
UML	Unified Modeling Language
WMC	Weighted Method per Class

List of Figures

2.1 CFG for ATM Withdrawal task 26
2.2 Fitness variation for test data 27

3.1 A typical FFNN 37
3.2 Architecture of RBF Network 40
3.3 Flat net structure of FLANN 42
3.4 Basic architecture of PNN 44
3.5 Histogram for WMC 51
3.6 Histogram for DIT 51
3.7 Histogram for NOC 51
3.8 Histogram for CBO 51
3.9 Histogram for RFC 51
3.10 Histogram for LCOM 51
3.11 Logistic graph 60
3.12 Convergence characteristics for Gradient Descent 66
3.13 Convergence characteristics for PCA based Gradient Descent . . 67
3.14 Convergence characteristics for RST based Gradient Descent . . 67
3.15 Convergence characteristics for Levenberg-Marquardt 68
3.16 Convergence characteristics for PCA based Levenberg Marquardt 68
3.17 Convergence characteristics for RST based Levenberg Marquardt 68
3.18 Convergence characteristics for FLANN 69
3.19 Convergence characteristics for PCA based FLANN 70
3.20 Convergence characteristics for RST based FLANN 70
3.21 Convergence characteristics for Gradient RBFN 72
3.22 Convergence characteristics for PCA based Gradient RBFN . . 72
3.23 Convergence characteristics for RST based Gradient RBFN . . . 72
3.24 Convergence characteristics for Hybrid RBFN 73
3.25 Convergence characteristics for PCA based Hybrid RBFN . . . 74
3.26 Convergence characteristics for RST based Hybrid RBFN 74
3.27 Varying accuracy rate for smoothing parameter in PNN 74
3.28 Varying accuracy rate for smoothing parameter in PCA based PNN 75

3.29 Varying accuracy rate for smoothing parameter in RST based PNN 75
3.30 A typical feed forward neural network 80

4.1 Cost-based evaluation framework for software fault classification 91

5.1 Basic ECFG 105
5.2 Methods association in ECFG 106
5.3 Sequence diagram for ATM withdrawal task 112
5.4 Basic CFG 113
5.5 ECFG for Bank ATM 122
5.6 ECFG for Bank ATM withdraw task 122
5.7 Fitness variation for test data 126

List of Tables

2.1 Literature survey for fault prediction 10
2.2 Literature survey for test data generation using meta-heuristic search algorithms 21
2.3 Alphabetical representation of nodes in control flow graph for Figure 2.1. 25
2.4 Percentage of class of test data having maximum fitness values in GA, PSO, CSA, BPSO, GSAA, ABC and BFA respectively. . 26

3.1 CK metrics suite 32
3.2 Radial functions available in literature 40
3.3 Confusion matrix to classify a class as faulty or not-faulty . . . 46
3.4 Distribution of bugs for AIF version 1.6 50
3.5 Descriptive statistics of classes 52
3.6 Correlations between metrics 53
3.7 Principal components 56
3.8 Linear regression analysis 58
3.9 Coefficients of features for PCA based Linear regression analysis 59
3.10 Linear regression analysis for AIF Version 1.6 after applying PCA 59
3.11 Coefficients for Linear regression analysis after applying RST . . 59
3.12 Linear regression analysis for AIF Version 1.6 after applying RST 60
3.13 Analysis of univariate regression for AIF Version 1.6 61
3.14 Multivariate logistic regression analysis for AIF Version 1.6 . . . 61
3.15 Before applying regression 61
3.16 After applying regression 61
3.17 Precision, Correctness, Completeness, and Accuracy for AIF version 1.6 62
3.18 Result of multivariate logistic regression 63
3.19 Confusion matrix for PCA based regression 63
3.20 Precision, Correctness, Completeness, Accuracy for AIF Version 1.6 after applying PCA (in terms of %) 63
3.21 Analysis of univariate regression for AIF Version 1.6 after applying reduct data of RST 64

3.22 Result of multivariate logistic regression after applying RST . . 64
3.23 Confusion matrix for RST based regression 64
3.24 Precision, Correctness, Completeness, Accuracy for AIF Version 1.6 after applying RST (in terms of %) 65
3.25 Accuracy prediction for Gradient Descent using full feature set . 65
3.26 Accuracy prediction for PCA and RST based Gradient Descent 66
3.27 Accuracy prediction for LM method using full feature set 67
3.28 Accuracy prediction for PCA and RST based LM method 68
3.29 Accuracy prediction for FLANN using full feature set 69
3.30 Accuracy prediction for PCA and RST based FLANN 69
3.31 Accuracy prediction for Basic RBFN using full feature set . . . 70
3.32 Accuracy prediction for PCA and RST based Basic RBFN . . . 71
3.33 Accuracy prediction for Gradient RBFN using full feature set . . 71
3.34 Accuracy prediction for PCA and RST based Gradient RBFN . 72
3.35 Accuracy prediction for Hybrid RBFN using full feature set . . . 73
3.36 Accuracy prediction for PCA and RST based Hybrid RBFN . . 73
3.37 Performance parameters for fault prediction models 76
3.38 Performance parameters for PCA based fault prediction models 77
3.39 Performance parameters for RST based fault prediction models . 77
3.40 Comparison of fault prediction accuracy for three data sets . . . 78
3.41 Accuracy comparison: Implemented models vs Existing models 79
3.42 Complexity expression for prediction models 81

4.1 Cost evaluation framework for fault classification 85
4.2 Removal costs of test techniques (in staff hour per defect) 87
4.3 Fault identification efficiencies of different test phases 87
4.4 Confusion matrix 94
4.5 Confusion matrix for Logistic classifier 94
4.6 Confusion matrix for Gradient Descent 95
4.7 Confusion matrix for Levenberg Marquardt 95
4.8 Confusion matrix for Basic RBFN 96
4.9 Confusion matrix for Gradient RBFN 96
4.10 Confusion matrix for Hybrid RBFN 97
4.11 Confusion matrix for FLANN 97
4.12 Confusion matrix for PNN 97
4.13 Fault removal cost for AIF 1.6 using various classifier models . . 98

5.1 Equivalent predicate of branch function 111
5.2 Parameters used in CSA for test data generation. 123
5.3 Parameters used in BPSO, and ABC for test data generation. . 123

5.4 Percentage of class of test data having maximum fitness values in CSA, BPSO, and ABC respectively. 124
5.5 Affinity and the respective test data generated for meta-heuristic techniques 125
5.6 Code coverage analysis for proposed methodology 126
5.7 Code coverage analysis for existing methodology 127

List of Symbols

α	Learning Parameter
β	Multiplying Factor
β_0	Constant
β_1	Co-efficient value of a variable
P_c	Probability of Crossover
P_m	Probability of Mutation
μ	Combination coefficient
R^2	Coefficient of multiple determination
ϕ	Radial Function
σ	Smoothing Parameter
C_i	Initial setup cost of used fault-prediction technique
C_u	Normalized fault removal cost in unit testing
C_s	Normalized fault removal cost in system testing
C_f	Normalized fault removal cost in field testing
M_p	Percentage of classes unit tested
δ_u	Fault identification efficiency of unit testing
δ_s	Fault identification efficiency of system testing
N_r	Renewed antibodies
N_s	Worst antibodies

Chapter 1

Introduction

Improving reliability of the desired software is one of the most sought out research areas in software engineering. Software developers lay emphasis on designing a reliable software, so that poorly designed softwares can be detected in the preliminary stages of the software development life cycle (SDLC) to avoid delivering low quality software product to the stakeholder. Thus, software quality acts as a crucial factor in determining the reliability of a software. So, there is a need for design of prediction models to predict fault prone modules or classes in software developed based on object-oriented development methodology.

In literature it is observed that, several quality models have been proposed and studied such as McCall's quality model [1], Boehm's quality model [2], Dromey's quality model [3], etc. to evaluate the quality of a software product. It is a fact that, a large software consists of large number of lines of code in turn leading to the presence of a huge number of modules. It is quite difficult to carry out unit testing of each and every module. In order to check the functionality and to ensure reliability of the software, a limited number of important logical paths in a module should be selected and testing should be exercised on those modules, where probability of faults are high [4]. Thus there

is a need for identifying fault prone modules, which needs to be carried out prior to the testing phase. By doing so, it becomes convenient for testers to perform effective program testing.

Software metrics play a crucial role in predicting the quality of the software. They provide a quantitative basis, and a process for validating the models during SDLC [5]. The usefulness of these metrics lies in their ability to predict the reliability of the developed software. In practice, software quality of a software system can be best determined based on the FURPS model, which characterizes parameters such as Functionality, Usability, Reliability, Performance and Supportability [6]. Quality of any product is mostly decided on the basis of an important parameter like reliability. Reliability is generally measured by the number of faults identified in the developed software during a time span. Developers intend to predict faults in modules apriori so as to deliver a software with minimum number of faults. A number of models have been developed for fault prediction as available in literature. Still, fault prediction remains as a challenging task in software engineering. There is a need for designing efficient models to predict software prone modules more accurately.

Artificial intelligence

Artificial intelligent (AI) techniques are the science, and engineering of making intelligent machines, especially intelligent computer programs. These techniques have the ability of computer, software and firmware to do those things that we, as humans, recognize as intelligent behavior [7].

Techniques based on artificial intelligence (AI) have proved to be ideal for prediction models as observed in literature. AI techniques cover wide range of topics such as Artificial neural networks (ANN), Evolutionary computation, Swarm intelligence (Particle swarm optimization, Ant colony optimization, Bacterial foraging algorithm), Fuzzy systems, and Artificial immune systems (AIS).

Artificial Neural Networks

This technique involves learning strategies inspired by modeling neurons in the brain [8]. The learning strategy is categorized into supervised and unsupervised learning, which manage feedback. Hence, the learning process is considered as adaptive learning and are applied to function approximation, prediction/estimation and pattern recognition domains.

Meta-heuristic methods

Meta-heuristics are strategies that âĂIJguideâĂİ the search process, are approximate, and usually non-deterministic. In general 'heuristic' methods are trade-off concerns such as precision, quality, and accuracy. The correctness or optimality of the solution is not a matter of concern for heuristics while finding the solutions. Similarly, meta-heuristics are also the generic algorithmic frameworks which can be applied to numerous optimization problems [9]. This involves less changes to be made for a specific problem. One or more heuristics are coupled to form meta-heuristic approaches to enhance their capabilities.

Evolutionary Computation Process

Evolutionary computation process is inspired by the "Theory of Natural Evolution". More often Evolutionary Algorithms include Genetic Algorithm, Evolution Strategy, Genetic and Evolutionary Programming, and Differential Evolution [10]. The evolutionary computation is considered as an adaptive strategy, and is typically applied to search and optimization problems.

Swarm Intelligence

This paradigm consists of two dominant sub-sets [11]:

1. Ant Colony Optimization (ACO): investigates probabilistic algorithms inspired by the foraging behavior of ants, and

2. Particle Swarm Optimization (PSO): investigates probabilistic algorithms inspired by the flocking and foraging behavior of birds and fish.

Similar to evolutionary computation, swarm intelligence based techniques are considered as adaptive strategies and are typically applied to search and optimization problems. Hence these algorithms are used for optimal test data generation.

Artificial Immune Systems

This technique has evolved from the structure and function of the immune system of vertebrates. Some of the popular approaches of Artificial Immune Systems (AIS) include: clonal selection algorithm (CSA) [12], negative selection [13] and the immune network algorithms [14]. The AIS algorithms show similarity between Evolutionary computation and Artificial neural networks, and are typically used in solving optimization and pattern recognition problems.

1.1 Motivation

In the present day, it is observed that in many software organizations emphasis is laid on reducing the development cost, effort, time consumed for development, and produce reliable software by increasing the software quality. Due to the presence of large lines of code constituting to a huge number of modules in a program, has lead to increase in complexity. This lead to the difficulty in producing reliable software without faults. The other obvious reason for failing to produce reliable software is due to the lack of proper testing activities and time.

This sort of problem can be better handled by predicting certain quality attributes such as fault proneness, maintenance effort, and the testing effort during the early stages of software design. To achieve these objectives, sufficient testing of the software product needs to be carried out. Also exhaustive testing is not possible because it leads to more testing cost to be incurred, and will be very time consuming due to the large size of the product. Thus, it is

very much essential to recognize the classes which are often quite fault prone. There are many approaches to identify such fault prone classes and software metrics are one such indicators. The fault prone models predicted using these software metrics can be used in early stages of SDLC. This will benefit the developers to emphasize on reducing the utilization of testing resources on the predicted faulty classes only. Hence, this will significantly benefit in saving time and resources during the development of a software.

1.2 Research Objectives

The research objectives outlined in this thesis are as follows:

1. To find whether design metrics are good enough for fault prediction models by establishing the relationship between object-oriented metrics and fault proneness.

2. To find the suitable model for fault prediction among the predictor models used based on certain performance parameters.

3. To evaluate the effectiveness of fault prediction based on fault removal cost (cost based evaluation framework).

4. To generate effective test data for object-oriented program by using various meta-heuristic search techniques.

1.3 Summary of Contributions

This thesis, investigates the design of various models for fault prediction analysis, and the application of meta-heuristic search algorithms to automatically generate test data for both traditional and object-oriented programs. Five models for fault prediction analysis along with the study of effectiveness of features reduction techniques are presented. Contributions of this thesis are summarized as follows:

- Designing fault prediction models using full feature set and reduced feature set by considering Chidamber and Kemerer metric suite as input.
- Evaluating the effectiveness fault prediction models using the proposed cost based evaluation framework.
- Generating effective test data by extending the concept of control flow graph for object-oriented program. Optimal test data are generated by application of the meta-heuristic search techniques.

1.4 Outline of the Thesis

This thesis is organized into six different chapters including introduction. Each chapter is discussed below in a nutshell.

Chapter 2: Literature Review

This chapter focuses on the state-of-art of various models (predictors, and classifiers) for fault prediction analysis and meta-heuristic search techniques for test data generation. A tabular representation of various artificial intelligence based schemes along with their application are presented for fault prediction and test data generation.

Chapter 3: Effectiveness of Machine Learning Methods in Fault Prediction Analysis

This chapter focuses on designing fault prediction models using various statistical and machine learning methods. Fault prediction analysis is performed by using full feature set and reduced feature set of Chidamber and Kemerer metric suite. Later, the chapter draws a comparative analysis for the fault prediction accuracy obtained by using the full feature and reduced feature datasets, has been carried out for critical assessment.

Chapter 4: Cost Based Evaluation Framework for Software Fault Classification

This chapter focuses on inspecting the usability of fault prediction. A cost based evaluation framework is proposed to assess the usability. This chapter highlights on the accuracy for classification of faults using logistic regression and various neural network models as classifiers.

Chapter 5: Test Data Generation for Object-Oriented Program using Meta-heuristic Search Algorithms

In this chapter, an automated approach in generating test data for Object-Oriented program using meta-heuristic search algorithms is presented. Control flow graph for Object-Oriented programs named as Extended Control Flow Graph (ECFG) is automatically generated. From the generated ECFG, test data are generated for a case study on Bank ATM withdrawal task [15] using three meta-heuristic search techniques.

Chapter 6: Conclusions

This chapter presents the conclusions drawn from the proposed work with much emphasis on the work done. The scope for further research work has been discussed at the end.

Chapter 2

Literature Review

This chapter focuses on the state-of-the-art of various models (predictors, classifiers) and optimization algorithms. The review has been performed in two broad aspects of software engineering with respect to objectives of the thesis.

This chapter highlights on the research work carried out by various authors on fault prediction. The survey includes various criteria chosen by the authors such as the metric set used, the methods employed, and the data set used for fault prediction analysis. The results on the survey work done for fault prediction conclude that a good number of researchers and practitioners have employed Chidamber and Kemerer metrics suite for fault prediction.

The later part of the chapter highlights on the use of meta-heuristic optimization techniques as available in literature for test data generation. In this thesis, seven meta-heuristic search techniques such as Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Clonal Selection Algorithm (CSA), Binary Particle Swarm Optimization (BPSO), Genetic Simulated Annealing Algorithm (GSAA), Artificial Bee Colony (ABC), and Bacterial Foraging Algorithm (BFA) are applied to traditional methods for test data generation.

2.1 Fault prediction

A good number of software metrics have been proposed by different researchers, e.g. McCabe [16], Halstead [17], Li and Henry [18], Chidamber and Kemerer metric suite [19], Abreu MOOD metric suite [20], Lorenz and Kidd [21], Robert C. Martin's metric suite [22], Tegarden *et al.* [23], Melo *et al.*[24], Briand *et al.* [25], Etzkorn *et al.* [26] etc. to study the quality of traditional and object-oriented systems.

Numerous empirical studies have used different combination and subsets of these metrics, and have analyzed the relationship between the object-oriented metrics and the fault proneness. In this thesis, literature review has been provided for all such previous studies. In this review, emphasis is laid on the metrics, dataset, and the evaluation techniques used to carry out the fault prediction analysis. In all the studies, independent variables are the subset of the object-oriented metrics and the dependent variable is the fault proneness.

There have been various empirical studies done in the field of fault prediction [27, 28, 29, 30, 31, 32, 33, 34]. In this work, a study on the influence of object-oriented metrics on quality attributes, and design of relevant models which help to predict these quality attributes are considered. These metrics are widely used in most of the studies as independent variables. Some of the studies have also defined their own software metrics and have carried out the fault prediction analysis based on them. Also, it is noticed that the statistical method, i.e.,the logistic regression has been often used by good number of authors. Machine learning methods have also been used in some of the studies.

To obtain the results, i.e., to find the relationship between the software metrics and the fault proneness, there are various machine learning methods such as artificial neural network, decision tree, genetic programming, logistic regression, naive bayes network, random forest, support vector machine, etc. Each study makes use of different evaluation methods which have been listed in our review.

In Table 2.1, the first column indicates the name of the author, and the year in which the work was carried out. The second column indicate the metrics set, and the third column represents the methods employed for fault prediction. Last column of the table represents the dataset used, in which different methods were applied to obtain the results. From Table 2.1, it can be observed that Chidamber and Kemerer [19] metrics suite is widely used in most of the studies.

Table 2.1: Literature survey for fault prediction

Author	**Metric used**	**Method used**	**Data set**
Briand (2000)	28 coupling, 10 cohesion and 11 inheritance metrics	Logistic regression, Principal component analysis	Hypothetical video rental business [35].
Cartwright (2000)	ATTRIB, STATES, EVNT, READS, WRITES, DELS, RWD. DIT, NOC, LOC, DFCT	Spearmans rank correlation	Large European telecommunication industry, which consists of 32 classes and 133KLOC [27].
Emam (2001)	CK and Briand metrics	Logistic regression	Java application which implements a word processor. Used two versions: Ver 0.5 and Ver 0.6 consisting of 69 and 42 classes respectively [36].

Gyimothy (2005)	CK metrics suite, LCOM, and LOC	Linear and Logistic regression, Decision tree, and Neural network	Source code of Mozilla with the use of Columbus framework [37].
Nachiappan (2005)	STREW-J metric suite (Number of Assertions, Number of Test cases)	Multiple linear regression, Principal component analysis	Open source eclipse plugin developed at North Carolina State University (NCSU) [38].
Zhou (2006)	CK metrics suite and SLOC	Logistic regression, Naive Bayes, Random forest	NASA, consisting of 145 classes, 2107 methods and 40K LOC [31].
Olague H. M (2007)	CK metrics, MOOD and QMOOD metrics	Logistic regression	Mozilla - Rhino project [30].
Kanmani (2007)	10 cohesion, 18 inheritance, 29 coupling and 7 size measures (total 64 metrics)	Logistic regression, Neural network, Probabilistic neural network	Library management system, which consists of 1185 classes [39].

Pai (2007)	CK metric suite and SLOC	Multiple regression, Ordinary least square, Bayesian linear regression, Bayesian poisson regression	Public domain dataset of KCI (implemented in C++), consisting of 2107 methods, 145 classes and 43 KLOC [33].
Tomaszewksi (2007)	Used CK metrics suite at both class and component level	Linear regression, Expert estimation	Two telecommunication project developed by Ericsson, consisting of 800 classes (500KLOC) and 1000 classes(600KLOC) respectively [40].
Tomaszewksi (2007)	Coupling, WMC, DIT, NOC, RFC, Cyclomatic complexity, comment ratio, number of executable statements, number of comment lines	Linear regression, Stepwise linear regression	Used two telecommunication projects developed by Ericsson, consisting of 800 classes (500 KLOC) and 1000 classes (600 KLOC) respectively [41].
Shatnawi (2008)	CK metrics suite, Lorenz and Kidd	Logistic regression	Eclipse project: Bugzilla database and Change log [42].

Aggarwal (2009)	Coupling, Cohesion, Inheritance and Size metrics	Statistical regression analysis and Principal component analysis	Student projects at University School of Information Technology (USIT) [29]
Singh (2009)	CK metrics suite and SLOC	Support vector machine	NASA, consisting of 145 classes, 2107 methods and 40K LOC [43].
Cruz (2009)	RFC, CBO, and WMC	Logistic regression	638 classes of Mylyn software [44].
Burrows (2010)	Baseline metrics such as CAE, CBM, DIT and coupling metrics	Logistic regression	iBATIS, Health watcher, Mobile media [28].
Singh (2010)	CK metric suite and SLOC	Logistic regression, ANN and Decision tree	NASA, consisting of 145 classes, 2107 methods and 40K LOC [34].
Zhou (2010)	AMC, aVGloc, CCMax, LOC, NIM, NCM, NTM, NLM, SDMC, WMC,	Logistic regression	Three releases of Eclipse, consisting of 6751, 7909, 10635 java files and 796, 988, 1306 KLOC respectively [45].
Fangjun (2011)	WMC, NOC, LCOM, CBO	Decision tree analysis	NASA data set [32].

Malhotra (2011)	CK metrics suite, Ca, Ce, NPM, LCOM3, LOC, DAM, MOA, MFA, CAM, IC, CBM, $\text{Max}_c c$ and $\text{Avg}_c c$	ANN, Bagging, Random forest, Boosting, Naive Bayes and KStar	Open source software [46].
Mishra (2012)	Complexity metrics (FOUT, MLOC) and Abstract syntax tress based metrics	SVM, Fuzzy, and Naive Bayes	Eclipse and Equinox data sets [47].
Malhotra (2012)	CK metrics suite, Ca, Ce, NPM, LCOM3, LOC, DAM, MOA, MFA, CAM, IC, CBM, Max cc and $\text{Avg}_c c$	Logistic regression, random forest, Adaboost, Bagging, Multilayer perceptron, SVM, Genetic programming	Apache POI [48].
Heena (2013)	CBO and NOC	Bayesian inference	Open Source Eclipse System [49].

2.1.1 Observations

The survey with respect to fault prediction can be summarized in the form of following results:

i. **Metric suites**:

Numerous metric suites have been proposed in the literature such as CK metric suite, MOOD, QMOOD, Lorenz and Kidd, etc. But it is observed that the CK metrics suite is more widely used than other metric suites. Also some researchers have defined their own new metrics and have used them in fault prediction. Some researchers have used large number of metrics, e.g. Kanmani *et al.* have used 64 metrics for fault prediction analysis [39].

ii. **Category**:

Different approaches for fault, are mainly categorized into statistical and machine learning methods. It is observed that researchers explored the potential features of machine learning methods to predict fault prone classes. The results obtained, indicate the effectiveness of machine learning systems. Thus, machine learning methods such as artificial neural network, bagging, decision tree, random forest etc. should be widely used for further studies. Among the statistical methods, linear and logistic regression techniques are also widely used by the researchers. It is further observed that most of the studies have used both, the machine learning methods and the statistical methods to obtain the fault prediction results.

iii. **Data set**:

Many researchers have used different types of datasets which are mostly public datasets, commercial datasets, open source or students/university datasets. It is observed from the literature survey that the data sets

from PROMISE and NASA repositories (public datasets) are the most commonly used datasets in the study of fault prediction.

These are the main areas on which the literature review is summarized. There are various other details in most of the articles such as the validation method used to evaluate the model (e.g., the holdout method, K-cross validation, leave-one-out method etc.), the evaluation criteria used etc. There are other various evaluation criteria used by different studies such as Receiver Operating Characteristic (ROC) curve, statistical parameters, Mean Absolute Error, Mean Absolute Relative Error, Root Mean Square Error and Standard Error of the Mean etc. in fault prediction analysis.

2.2 Test data generation for traditional methods

In this section, the role of meta-heuristic search techniques are analyzed in generating test data. Test data are generated for traditional methods using various meta-heuristic techniques such as Genetic algorithm (GA), Particle swarm optimization (PSO), Clonal selection algorithm (CSA), Binary particle swarm optimization (BPSO), Genetic simulated annealing algorithm (GSAA), Artificial bee colony (ABC), and Bacterial foraging algorithm (BFA). These techniques are applied on a case study i.e., withdrawal task in Bank ATM [15], and it is observed that clonal selection algorithm is able to generate suitable test data in a more efficient manner. This section further, gives the brief details about the meta-heuristic techniques used for automatic test data generation.

Genetic Algorithm

Genetic algorithm (GA) is a population-based search method and was introduced by Holland [50]. GA are a family of computational models inspired by evolution.

Candidate solutions are represented as chromosomes, with the solution represented as genes in the chromosomes. The possible chromosomes form a search space, and are associated with a fitness function representing the value of solutions encoded in the chromosome. The search proceeds by evaluating the fitness of each of the chromosome in the population, and then performing point mutations and recombination of the successful chromosomes. GA can defeat random search in finding solutions to complex problems. GA has been successfully used to automate the generation of test data. Table 2.2 gives the literature survey on the use of GA in test data generation.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) technique was introduced in 1995 by Kennedy *et al.* [51]. In comparison with GA search, the PSO is a relatively recent optimization technique of the swarm-intelligence paradigm. Inspired by social metaphors of behavior and swarm theory, simple methods were developed for efficiently optimizing non-linear mathematical functions.

Similar to GA search, the system is initialized with a population of random solutions, called particles. Each particle maintains its own current position, its present velocity and its personal best position explored so far. The swarm is also aware of the global best position achieved by all its members. Swarm updates it's personal and global best in each time stamp, by moving to a new position. Table 2.2 gives the literature survey on the use of PSO in test data generation.

Clonal Selection Algorithm

Clonal Selection Algorithm (CSA) is an optimization algorithm based on biological immune system, in which the antigen corresponds to the problem under-solved and the antibody is referred to as a solution to the problem [12]. CSA is a branch of artificial immune system algorithms with unique inherent

property (hyper mutation) that makes it an efficient optimization technique.

In basis path testing, the aim of the tester is to find suitable test data that will satisfy the given target path. To achieve this, the random test data (input values) are encoded as antibodies and the antigens as the ones satisfying the test data requirements. The CSA begins with a randomly generated initial population. The test data are evaluated based on the affinity function. This affinity function is a description of how best the individual test data perform in code coverage. Table 2.2 gives a brief overview of the literature survey for test data generation using CSA.

Binary Particle Swarm Optimization

Binary Particle Swarm Optimization (BPSO) was introduced by Kennedy and Eberhart in 1997 [52]. When compared to PSO, in the binary version of PSO (BPSO), every particle is represented as a bit. Each bit of a particle in BPSO is associated with a velocity, which is the probability of changing the bit to 1. Particles are updated bit by bit and velocity is restricted within the range [0,1].

Consider P to be the probability of changing a bit from 0 to 1, then $1 - P$ will be the probability of not changing the bit to 1. This probability can be given as:

$$P(x_{id}(t) = 1) = f(x_{id}(t), v_{id}(t-1), p_{id}, p_{gd}) \tag{2.1}$$

The possibility that an individual particle i for the dth site in the bit string will choose a bit 1 is represented by $P(x_{id}(t) = 1)$. The current state of the ith particle at bit d is determined by $x_{id}(t)$. The current probability that a particle in the string will choose 1 is measured by $V_{id}(t-1)$. A p_{id} value of 1 or 0 determines the best state found so far for bit d of individual i. The value p_{gd} may be 1 or 0 depending on what value does the bit d possess in the global best particle. The commonly used measure for 'f'' is the sigmoid

function, which is defined as:

$$f(V_{id}(t)) = \frac{1}{1 + e^{-v_{id}(t)}} \tag{2.2}$$

where,

$$V_{id}(t) = wv_{id}(t-1) + (\phi_1)(p_{id} - x_{id}(t-1) + (\phi_2)(p_{gd} - x_{id}(t-1))) \tag{2.3}$$

Equation 2.3 gives the update rule for the velocity of each bit, where ϕ_1 and ϕ_2 are random numbers drawn from the uniform distributions, and $[V_{min}, V_{max}]$ are the constant parameters. Table 2.2 gives the literature survey with respect to test data generation using BPSO.

Genetic Simulated Annealing Algorithm

Cerny proposed the use of SA in finding global minimum of a cost function which posses several local minima values [53]. In this work, the hybrid approach involving both genetic algorithm and simulated annealing (GASA) are used for test data generation. These two algorithms are the most promising heuristic search methods for their data type independence i.e., they have the ability to handle complex data types and generate qualitative test data. This helps in detecting undetected (unravel) unknown bugs/defects, which is not possible by typical test data generation techniques such as boundary value analysis and equivalence partitioning.

Also GA is less susceptible to local minima, which can cause a test data generation technique to halt without finding adequate input [54]. Even though GA and SA can produce test data with appropriate fault-prone ability [55, 56], they fail to produce test data fast due to their slow convergence speed. Table 2.2 gives a brief overview of various methods followed by researchers to generate test data using hybrid GA - SA approach.

Artificial Bee Colony

The Artificial Bee Colony (ABC) algorithm like GA and PSO, is biologically inspired technique of swarm intelligence. Seeley investigated the behavior of bees in distributing their work to optimize the collection of nectar and reported the working of bee colony to be robust and adaptive [57]. Table 2.2 shows the criteria used by researchers in generating test data using ABC technique.

Bacterial Foraging Algorithm

Bacterial Foraging Algorithm (BFA) is an optimization technique which mimics the behavior of bacteria forage, i.e., how the bacteria search for food over a landscape of nutrients to perform parallel non-gradient optimization[58].

Bacterial foraging optimization technique tends to eliminate poor foraging strategies and improve successful foraging strategies. After meeting the stopping criterion a foraging animal takes actions to maximize the energy obtained per unit time spent in foraging [59]. This phenomenon of foraging led to the use of this optimization technique by many researchers.

Each of the meta-heuristic techniques have their respective unique features which can be attributed as follows:

i. **Genetic algorithm (GA)**: helps to solve a good number of real time problems which cannot be solved in polynomial amount of time using deterministic algorithms. GA can obtain near optimal solution that can be generated quickly which is more desirable than optimal solution which requires huge amount of time.

ii. **Particle swarm optimization (PSO)**: provides solution to multi-modal problems like local optima problem. Good fitness function is available because every particle in the swarm updates its personal best by comparing with the global best particle in the swarm.

iii. **Clonal selection algorithm (CSA)**: helps in achieving diversified test data by performing hyper-mutation operation.

iv. **Binary particle swarm optimization (BPSO)**: involves exchange of information between individuals (particles) of the population (swarm).

v. **Genetic simulated annealing algorithm (GSAA)**: combines both the local search ability of SA and global search ability of GA to obtain optimal solution.

vi. **Artificial bee colony algorithm (ABC)**: algorithm involves less computational overhead as it uses only three control parameters.

vii. **Bacterial foraging algorithm (BFA)**: this probabilistic technique is helpful for problems such as numeric maximization or minimization, where it is extremely difficult to find approximate solutions.

The following table shows the list of various testing criteria used by researchers and practitioners for test data generation using meta-heuristic search techniques.

Table 2.2: Literature survey for test data generation using meta-heuristic search algorithms

	Author	**Criteria for testing**
GA	Pargas *et al.* (1999)	Generated test data and performed branch coverage [60].
	Lin *et al.* (2001)	Used ‘similarity’ fitness function to generate test data [61].
	Michael *et al.* (2001)	Automated test data generation using branch coverage [54].
	Mansour *et al.* (2004)	Used hamming distance as a fitness function for path coverage [62].

	Ahmed *et al.* (2008)	Generated test data for path coverage [63].
	Ciyong *et al.* (2009)	Generated test data for branch coverage [64].
	Srivastava *et al.* (2009)	Generated test cases for path coverage [65].
	Alsmadi *et al.* (2010)	Effective generation of test data [66].
	Rauf *et al.* (2010)	The ratio of number of paths followed out of the total number of paths were used as a fitness function [67].
	Hitesh *et al.* (2010)	Generated test data using heuristic approach [68].
	Sharma *et al.* (2013)	Presented survey on test data generation using GA [69].
	Swagatika *et al.* (2013)	Generated test data using GA [70].
	Alberto *et al.* (2013)	Generated test sequences for complex time systems using GA [71].
PSO	Andreas *et al.* (2007)	Generated test data and showed that PSO outperforms GA for complex programs [72].
	Aiguo *et al.* (2009)	All path test data generation [73].
	Huanhuan *et al.* (2010)	Efficient automated test data generation method [74].
	Sheng *et al.* (2010)	Hybrid approach using GA and PSO for automatic test data generation [75].
	Chengying *et al.* (2012)	Test data generation for structural program using swarm intelligence [76].

	Rui *et al.* (2012)	Automatic test data generation based on hybrid particle swarm genetic algorithm [77].
	Shaukat *et al.* (2014)	Test data generation by coupling integration testing with PSO [78].
CSA	Xiaofeng *et al.* (2008)	Clonal selection algorithm for test data generation based on basis paths [79].
	Konstantinos *et al.* (2008)	Generated test data for data-flow coverage [80].
	Ankur *et al.* (2012)	Test data generation based on branch distance using clonal selection algorithm [81].
BPSO	Khushboo *et al.* (2008)	Generated test data and compared the performance with GA [82].
	Gursaran *et al.* (2012)	Generated test data using BPSO and GA [83].
GASA	Haichang *et al.* (2005)	Test data were generated for benchmark programs with combination of GA and SA approach [84].
	Bao-Lin *et al.* (2007)	Automatic test data generation based on Length-N coverage criterion [85].
	Tan *et al.* (2009)	Analyzed the drawbacks in usage of GA and SA for test data generation [86].
	Bo Zhang *et al.* (2011)	Combined adaptive GA and SA to generate test data [87].
	Gentiana *et al.* (2012)	Generated test data using PSO, SA and compared with GA[88].
ABC	Jeyamala *et al.* (2009)	Demonstrated the superiority of the proposed approach over the existing GA approach [89].
	Surender *et al.* (2010)	Generated test data for structural programs using branch distance as fitness function [90].
	Arivnder *et al.* (2011)	Test data optimization for code coverage [91].

	Srikanth *et al.* (2011)	Generated test data for test case optimization [92].
	Shekar *et al.* (2012)	Generated test data for feasible independent test paths and compared the efficiency of ABC approach with other optimization techniques [93].
	Ranjeet *et al.* (2012)	Showed that ABC obtained near global optimal solution [94].
	Sandeep *et al.* (2013)	Generated test data using ABC [95].

The scenario considered here for design of fitness function is that the customer tries to withdraw certain amount from the ATM machine (this withdrawal amount is the initial test data generated randomly, with an assumption that the withdrawal amount entered by the customer is random).

The ATM system sends a message with the details of the amount and the account number to the bank system. The bank system retrieves the current balance of the corresponding account and compares it with the entered amount. If the balance amount is found to be greater than the entered withdrawal amount then the amount can be withdrawn and the bank system returns true, after which the customer can withdraw the money, otherwise it checks for balance. If the entered amount is less than the total amount (current balance) then the message is intimated as 'false'. Depending on the return value, the ATM machine dispenses the cash and prints the receipt or displays the failure message.

In this thesis, for automated test data generation using meta-heuristic search techniques, a small segment of code for ATM withdrawal scenario is considered.

1. $net_amt = 25000$, $min_bal = 1000$;
2. $bal(1,i) = net_amt - wd_amt(1,i)$;
3. if $wd_amt(1,i) < net_amt$
4. if $bal(1,i) < min_bal$
5. $fail_bal(1,k) = bal(1,i)$;
   else
6. $suc_bal(1,p) = bal(1,i)$;
7. $test_data(1,p) = wd_amt(1,i)$;

Table 2.3: Alphabetical representation of nodes in control flow graph for Figure 2.1.

Nodes in CFG	**Alphabetical Notation**
wd_amt	A
net_amt	X
bal	B
min_bal	C
$fail_bal$	D
suc_bal	E
$test_data$	F

The variables in the code such as net_amt represents the total amount as balance in a customers account, wd_amt represents the withdrawal amount entered by the customer during withdrawal task, min_bal corresponds to the least possible balance to be left over in a customers account after withdrawal, test_data is the set of all successful transactions done by the customer, suc_bal and fail_bal variables contain the successful and failed transactions test data respectively.

Figure 2.1 shows the generated CFG for Bank ATM withdrawal task and Table 2.3 shows the alphabetical representation of predicate nodes for Figure 2.1. Test data are generated by using Equation 2.4 (fitness function). The fitness function for Bank ATM withdrawal task is given as:

$$F = 1/((abs(suc_bal(i) - min_bal) + 0.05)^2) \qquad (2.4)$$

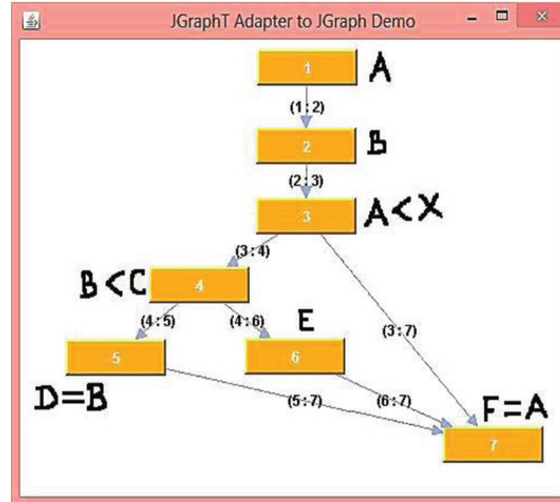

Figure 2.1: CFG for ATM Withdrawal task

2.2.1 Results and analysis

In this section, the results obtained for automatic test data generation are presented. The obtained test data are tabulated in Table 2.4. This table presents the comparison of percentage of test data having maximum fitness values (search space) for the the applied meta-heuristic technique.

Table 2.4: Percentage of class of test data having maximum fitness values in GA, PSO, CSA, BPSO, GSAA, ABC and BFA respectively.

Fitness value range	**% of test data**						
f(x)(Search space)	**GA**	**PSO**	**CSA**	**BPSO**	**GSAA**	**ABC**	**BFA**
$0 \leq f(x) < 0.3$	61	50	13	48	61	49	42
$0.3 \leq f(x) < 0.7$	01	07	17	05	02	01	07
$0.7 \leq f(x) < 1.0$	38	43	70	47	37	50	51

Table 2.4 gives a clear indication that the individuals having higher fitness (affinity) value, lie in the range $0.7 \leq f(x) < 1.0$. These fitness values are an indication of optimal test data obtained. Table 2.4 depicts that highest percentage of test data having maximum fitness value is achieved by CSA. Out of the 100 individuals in the population, test data having maximum fitness

value found in search space $0.7 \leq f(x) < 1.0$ can be chosen for testing purpose.

This inference helps a tester to choose suitable solution (test data) based on the affinity values out of the total population size (N) i.e., tester can choose test data from the categorized search space (based on affinity value).

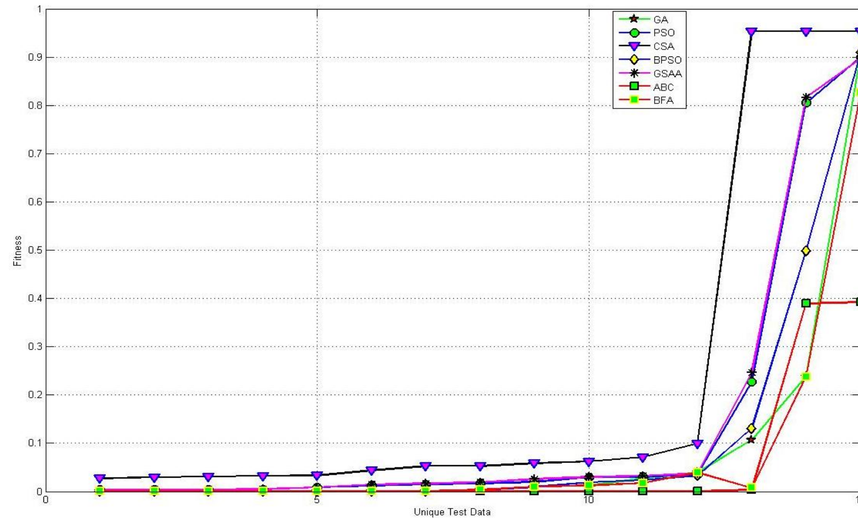

Figure 2.2: Fitness variation for test data

Figure 2.2 depicts the variation of fitness values of unique test data generated when the seven meta-heuristic techniques are applied. The techniques are run for 1000 generations. The graph shows that CSA has better fitness values for the generated test data when compared with other six meta-heuristic techniques.

2.3 Summary

In fault prediction analysis, it can be summarized that CK metrics suite has been used by a number of researchers as input in design of prediction models. Also out of these, a good number of authors have used statistical methods for design of these respective prediction models. It is noticed that the public datasets are also most commonly used in fault prediction analysis.

In the study related to test data generation, the behavior of seven meta-heuristic search techniques such as GA, PSO, CSA, BPSO, GSAA, ABC and BFA were applied for automated test data generation. As part of this work, the performance of each of the seven techniques was presented. It is observed from the obtained results, that CSA helps to generate suitable test data more efficiently. Even the maximum number of unique test data had higher fitness values in CSA when compared with other six techniques as shown in Figure 2.2.

Chapter 3

Effectiveness of Machine Learning Methods in Fault Prediction Analysis

---

Fault prediction models designed using various machine learning methods, use CK metrics suite as requisite input. In the first phase, full feature set of CK metric suite is used to compute the fault prediction accuracy. In the second phase, the proposed work for fault prediction is extended by applying feature reduction techniques such as Principal Component Analysis (PCA) and Rough Set Theory (RST). These attribute reduction techniques are applied to study the effectiveness of reduced attribute set. At the end, the chapter draws a comparative analysis of the fault prediction accuracy obtained by full feature set and reduct feature set of PCA and RST.

3.1 Introduction

Present day software development is mostly based on Object-Oriented paradigm. The quality of Object-Oriented software can be best assessed by the use of software metrics. Since decades, many software metrics have been proposed to evaluate the quality of software. These metrics help the practitioners as well as researchers to verify the quality attributes of a software such as effort, maintainability and fault proneness.

The usefulness of these metrics lies in their ability to predict the reliability of the developed software. In practice, software quality mainly refers to reliability, maintainability and understandability. Reliability is defined as the probability that software will not cause the failure of a system for a specified time under specified conditions (IEEE Std 982.2-1988) [96], and is generally measured by the number of faults found in the developed software. The faults occur mainly due to the presence of large number of lines of code which constitute a very huge number of modules in the developed software. Software fault prediction is a challenging task for researchers before the software is released. Prediction of fault-prone modules is one of the major goals so as to make a software fault free before its delivery to the client.

Several researchers have worked on building prediction models for software fault prediction. This chapter aims to assess the influence of CK metrics, keeping in view of predicting faults for an open-source software product. Machine learning methods such as statistical methods (linear regression, logistic regression) are very often used for classification of faulty classes, and methods such as Artificial neural network (ANN), Functional link artificial neural network (FLANN), Radial basis function network (RBFN), Probabilistic neural network (PNN) are applied for prediction of fault rate. It is observed in literature that metric suites have been validated for small data sets [97]. In this work, the results achieved for an input data set of 965 classes are validated by comparing with the results obtained by Basili *et al.* [97] for statistical analysis.

In this chapter, the following queries are investigated:

- Q1: Are design metrics good enough to be considered for fault prediction models.
- Q2: Which independent variable should be included in fault prediction models.
- Q3: Which modeling technique performs best when used in fault prediction.
- Q4: Does feature reduction techniques such as PCA and RST influence/increase the accuracy rate of fault prediction for the implemented techniques.

3.2 Research Background

The following sub-sections highlight on the data set used for fault prediction. Data are normalized to obtain better accuracy and then dependent as well as independent variables are chosen for fault prediction.

3.2.1 Empirical Data Collection

Metric suites are applied for different goals such as fault prediction, effort estimation, re-usability and maintainability. In this study, CK metrics suite is used for fault prediction [19].

The CK metrics suite consists of six metrics viz., Weighted Method per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response For Class (RFC) and Lack of Cohesion (LCOM) [19]. In this suite, WMC, DIT and NOC indicate the class hierarchy, CBO and RFC indicate the class coupling whereas LCOM represents cohesion. Table 3.1 gives a short note on the six CK metrics and the threshold for each of the six metrics.

Table 3.1: CK metrics suite

CK Metric	**Description**	**Value**
WMC	Sum of the complexities of all class methods.	Low
DIT	Maximum length from the node to the root of the tree.	< six
NOC	Number of immediate sub-classes subordinate to a class in the class hierarchy.	Low
CBO	Count of the number of other classes to which it is coupled.	Low
RFC	A set of methods that can potentially be executed in response to a message received by an object of that class.	Low
LCOM	Measures the dissimilarity of methods in a class via instanced variables.	Low

The metric values of the suite are extracted using Chidamber and Kemerer Java Metrics (CKJM) tool. It follows the Unix tradition, i.e., it does not offer any GUI interface to the user.

CKJM tools extracts Object-Oriented metrics by processing the byte code of compiled Java classes. The programs are executed by specifying the class files (or pairs of jar/class files) on its command line. The CKJM tool, on its standard output line displays a line for each class containing the complete name of the class and the values of its metrics.

This tool is used to extract metric values for AIF (an open source framework) version 1.6 available in the Promise data repository [98]. The versions of the AIF used from the repository were developed in Java language. The CK metrics values of the AIF are used for fault prediction analysis.

3.2.2 Data Normalization

Normalization of the data set is necessary in order to apply ANN models, which accept normalized data which lie in the range: 0 to 1. In literature, techniques such as Min-Max normalization, Z-Score normalization and Decimal scaling are available for normalizing the data. In this thesis, Min-Max normalization technique is used to normalize the data [99]. Min-Max normalization performs a linear transformation on the original data. Each of the actual data d of attribute p is mapped to a normalized value d′ which lies in the range of 0 to 1. The Min-Max normalization is calculated by using the following equation:

$$Normalized(d) = d' = \frac{d - min(P)}{max(p) - min(p)} \tag{3.1}$$

where min(p) and max(p) represent the minimum and maximum value of the attribute respectively.

3.2.3 Dependent and Independent Variables

The goal of this study is to explore the relationship between Object-Oriented metrics and fault proneness at the class level. In this thesis, a fault in a class is considered as a dependent variable and each of the CK metric as an independent variable. It is intended to develop a function between fault of a class and CK metrics (WMC, DIT, NOC, CBO, RFC, LCOM). Fault is a function of WMC, DIT, NOC, CBO, RFC and LCOM and can be represented as shown in the following equation:

$$Faults = f(WMC, DIT, NOC, CBO, RFC, LCOM) \tag{3.2}$$

3.3 Machine Learning Methods

In this section, machine learning methods such as statistical methods, artificial neural network, radial basis function, functional neural network, prob-

abilistic neural network models are applied for fault prediction analysis using CK metrics as requisite input data.

3.3.1 Statistical Methods

This section describes the application of statistical methods for fault prediction. Regression analysis methods such as linear regression and logistic regression analysis are applied. In regression analysis, the value of unknown variable is predicted based on the value of one or more known variables.

3.3.1.1 Linear Regression Analysis

Linear regression is the commonly used statistical technique. It is the study of linear relationship between variables. This analysis technique is used when faults are distributed over a wide range of classes.

Linear regression analysis is of two types:

a. Univariate linear regression, and

b. Multivariate linear regression.

Univariate linear regression is based on:

$$Y = \beta_0 + \beta_1 X \tag{3.3}$$

where Y is the dependent variable (accuracy rate) and X is the independent variables (CK metrics).

In case of multivariate linear regression, the linear regression is based on:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + ... + \beta_p X_p \tag{3.4}$$

where X_i is the independent variable, β_0 is a constant and y is the dependent variable. Table 3.8 shows the result of linear regression analysis for AIF version 1.6.

3.3.1.2 Logistic Regression Analysis

Logistic regression analysis is used when the data set is not linear in nature. Here logistic regression analysis is used for predicting the outcome of dependent variable based on one or more independent variable(s). A dependent variable can take only two values. So the dependent variable of a class containing bugs is divided into two groups, one group containing zero bugs and the other having at least one bug.

Logistic regression analysis can be divided into two groups:

a. Univariate logistic regression, and

b. Multivariate logistic regression.

a. Univariate logistic regression analysis

Univariate logistic regression is carried out to find the impact of an individual metric in predicting the faults of a class. It is based on the following formula:

$$\pi(x) = \frac{e^{\beta_0+\beta_1 X_1}}{1+e^{\beta_0+\beta_1 X_1}} \tag{3.5}$$

where x is an independent variable and β_0, β_1 represent the constant and coefficient values respectively. Taking logarithm of the function, it can be expressed as:

$$logit[\pi(x)] = \beta_0 + \beta_1 X \tag{3.6}$$

where π represents the probability of a fault found in the class during validation phase.

The results of univariate logistic regression for AIF version 1.6 are tabulated in Table 3.13. The values of obtained coefficient are the estimated regression coefficients. The probability of faults being detected for a class is dependent on the coefficient value (positive or negative). Higher coefficient value means greater is the probability of a fault being detected. The significance of coefficient value is determined by the p-value. The p-value is assessed based on the

significance level (α). 'R' coefficient is the proportion of the total variation in the dependent variable explained in the regression model. High value of R is an indication of greater correlation between faults and the CK metrics.

b. Multivariate logistic regression analysis

Multivariate logistic regression is used to construct a prediction model for the fault proneness of classes. In this method, metrics are used in combination. The multivariate logistic regression model is based on the following equation:

$$\pi(x) = \frac{e^{\beta_0+\beta_1 X_1+\beta_2 X_2+\beta_3 X_3+......+\beta_p X_p}}{1+e^{\beta_0+\beta_1 X_1+\beta_2 X_2+\beta_3 X_3+......+\beta_p X_p}} \tag{3.7}$$

where x_i is the independent variable, π represents the probability of a fault found in the class during validation phase and p represents the number of independent variables. Taking logarithm of the function, it yields:

$$logit[\pi(x)] = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + ... + \beta_p X_p \tag{3.8}$$

Equation 3.8 shows that logistic regression is just a standard linear regression model, where the Dichotomous outcome of the result is transformed by the *logit* transform. The value of $\pi(x)$ lies in the range: $-\infty < \pi(x) < +\infty$. After the logit transform the value of $\pi(x)$ lies in the range: $0 < \pi(x) < 1$.

3.3.2 Artificial Neural Network

Artificial Neural Network (ANN) is a network of simulated neurons. ANN is inspired by the examination of central nervous systems. Warren *et al.* in 1943 created a computational model for neural networks based on mathematics and algorithms [8]. ANN is one of the artificial intelligence (AI) techniques, most commonly used for prediction and classification. This computational features involved in ANN architecture can be very well applied for fault prediction.

Figure 3.30 shows the architecture of ANN model, which contains three layers viz., input layer, hidden layer and output layer. Computational features involved in ANN architecture can be very well applied for fault prediction.

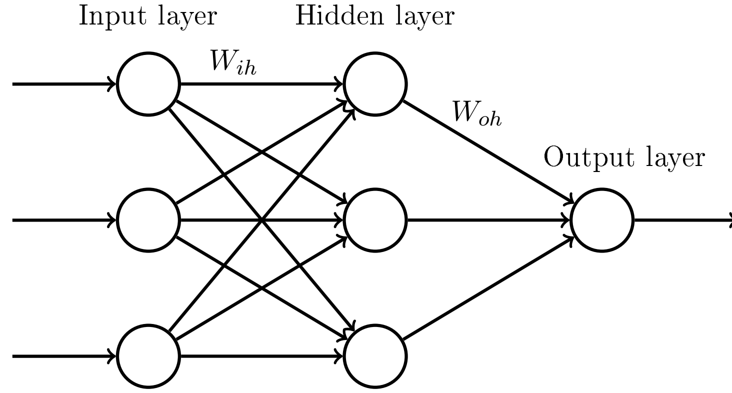

Figure 3.1: A typical FFNN

In this network, for input layer the linear activation function is used i.e., the output of the input layer 'O_i' is input of the input layer 'I_i', which is represented as :

$$O_i = I_i \tag{3.9}$$

For hidden layer and output layer, sigmoidal (squashed-S) function is used. The output of hidden layer O_h for input of hidden layer I_h is represented as:

$$O_h = \frac{1}{1 + e^{-I_h}} \tag{3.10}$$

Output of the output layer 'O_o' for the input of the output layer 'O_i' is represented as:

$$O_o = \frac{1}{1 + e^{-O_i}} \tag{3.11}$$

A neural network can be represented as:

$$Y' = f(W, X) \tag{3.12}$$

where X is the input vector, Y' is the output vector, and W is the weight vector. The weight vector W is updated in every iteration so as to reduce the Mean Square Error (MSE) value. MSE is formulated as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y'_i - y_i)^2 \tag{3.13}$$

where y is the actual output and y' is the expected output.

In literature, different methods are available to update weight vector ('W') such as Gradient Descent, Newton's method, Quasi-Newton method, Gauss Newton Conjugate-Gradient method, and Levenberg Marquardt method. In this chapter, Gradient Descent and Levenberg Marquardt methods are used for updating the weight vector W.

3.3.2.1 Gradient Descent learning method

Gradient Descent (GD) method is used for updating the weight during learning phase [100]. GD method uses first-order derivative of total error to find the *minima* in error space. Normally, Gradient vector G is defined as the first order derivative of error function. The error function is represented as:

$$E_k = \frac{1}{2}(T_k - O_k)^2 \tag{3.14}$$

and G is given as:

$$G = \frac{\partial}{\partial W}\big(E_k\big) = \frac{\partial}{\partial W}\big(\frac{1}{2}(T_k - O_k)^2\big) \tag{3.15}$$

After computing the value of Gradient vector G in each iteration, weighted vector W is updated as:

$$W_{k+1} = W_k - \alpha G_k \tag{3.16}$$

where W_{k+1} is the updated weight, W_k is the current weight, G_k is a Gradient vector, and α is the learning parameter.

3.3.2.2 Levenberg - Marquardt method

Levenberg - Marquardt (LM) method locates the minimum of multivariate function in a iterative manner. It is expressed as the sum of squares of non-linear real-valued functions [101, 102]. This method is used for updating the weights during learning phase. LM method is fast and stable in terms of its execution when compared with Gradient Descent method (LM is a combination

of Steepest Descent and Gauss Newton methods). In LM method, weight vector W is updated as:

$$W_{k+1} = W_k - (J_k^T J_k + \mu I)^{-1} J_k e_k \tag{3.17}$$

where W_{k+1} is the updated weight, W_k is the current weight, J is Jacobian matrix and μ is the combination coefficient i.e., when μ is very small it acts as Gauss Newton method, and if μ is very large, it acts as Gradient Descent method. Jacobian matrix is calculated as:

$$J = \begin{bmatrix} \frac{\partial}{\partial W_1}(E_{1,1}) & \frac{\partial}{\partial W_2}(E_{1,1}) & \cdots & \frac{\partial}{\partial \mathrm{W}_N}(E_{1,1}) \\ \frac{\partial}{\partial W_1}(E_{1,2}) & \frac{\partial}{\partial W_2}(E_{1,2}) & \cdots & \frac{\partial}{\partial W_N}(E_{1,2}) \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial}{\partial W_1}(E_{P,M}) & \frac{\partial}{\partial W_2}(E_{P,M}) & \cdots & \frac{\partial}{\partial W_N}(E_{P,M}) \end{bmatrix} \tag{3.18}$$

where 'N' is the number of weights, 'P' is the number of input patterns and 'M' is the number of output patterns.

3.3.3 Radial Basis Function Network

Radial Basis Function Network (RBFN) was first formulated by Broomhead *et al.* [103], and subsequently extended by Moody *et al.* [104]. RBFN is a feed forward neural network (FFNN), trained using supervised training algorithm. RBFN is generally configured by a single hidden layer, where the activation function is chosen from a class of functions called as basis functions.

Figure 3.2 shows the structure of a typical RBFN in its basic form involving three entirely different layers. RBFN is a form of ANN technique which contains three layers viz., input, hidden and output layer. RBFN contains h number of hidden centers represented as $C_1, C_2, C_3, ..., C_h$.

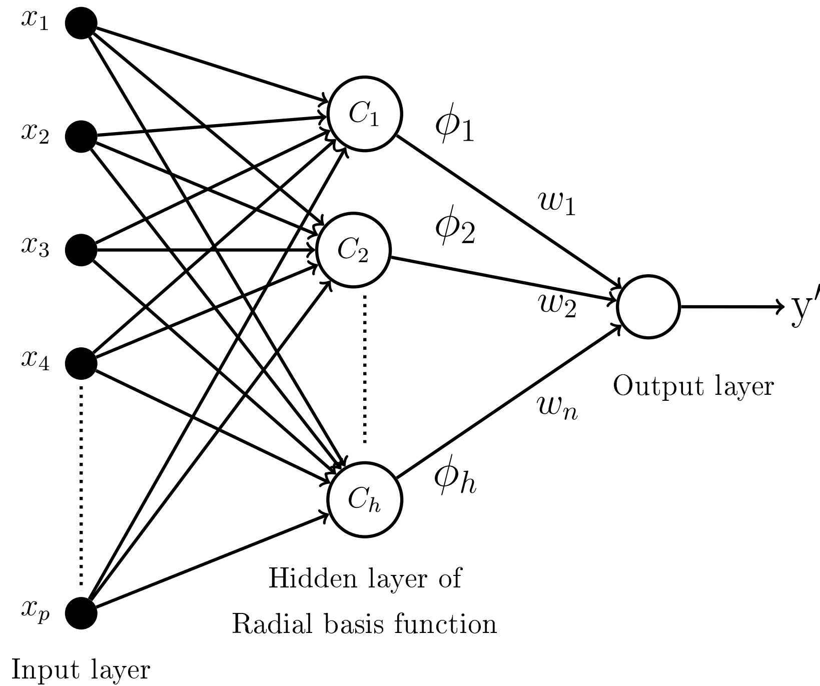

Figure 3.2: Architecture of RBF Network

The target output is computed as:

$$y' = \sum_{i=1}^{n} \phi_i W_i \tag{3.19}$$

where W_i is the weight of the ith center, ϕ is the radial function and y' is the target output. Table 3.2 shows the various radial functions available in literature.

Table 3.2: Radial functions available in literature

Radial function	**Mathematical expression**
Gaussian radial function	$\phi(z) = e^{-(z^2/2\sigma^2)}$
Thin plate spline	$\phi(z) = z^2 logz$
Quadratic	$\phi(z) = (z^2 + r^2)^{1/2}$
Inverse quadratic	$\phi(z) = \frac{1}{(z^2 + r^2)^{1/2}}$

In this analysis, Gaussian function is used as a radial function, and the distance vector z is calculated as:

$$z = ||x_j - c_j|| \tag{3.20}$$

where x_j is input vector that lies in the receptive field for center c_j. In this study, gradient descent learning and hybrid learning techniques are used for updating weight and center respectively.

The advantage of using RBFN technique lies in its training rate which is faster when compared with other propagation networks and is less susceptible to problem with non-stationary inputs.

3.3.3.1 Gradient RBFN method

Gradient Descent learning is a method used for updating the weight W and center C. The center C in Gradient learning is updated as:

$$C_{ij}(k+1) = C_{ij}(k) - \eta_1 \frac{\partial}{\partial C_{ij}}(E_k) \tag{3.21}$$

and weight W is updated as:

$$W_i(k+1) = W_i(k) - \eta_2 \frac{\partial}{\partial W_i}(E_k) \tag{3.22}$$

where η_1 and η_2 are the learning coefficients for updating center and weight respectively.

3.3.3.2 Hybrid RBFN method

In Hybrid learning method, radial function relocates its center in a self organized manner, while the weights are updated using learning algorithm. In this analysis, Least Mean Square (LMS) algorithm is used for updating the weights while the center is updated only when it satisfies the following conditions:

a. Euclidean distance between the input pattern and the nearest center is greater than the threshold value, and

b. Mean Square Error (MSE) is greater than the desired accuracy.

After satisfying the above conditions, the Euclidean distance is used to find the centers close to x and then the centers are updated as:

$$C_i(k+1) = C_i(k) + \alpha(x - C_i(k)) \tag{3.23}$$

After every update, the center moves closer to the training sample.

3.3.4 Functional Link Artificial Neural Network

Functional Link Artificial Neural Network (FLANN) was initially proposed by Pao [105]. It is a flat network having a single layer i.e., the hidden layers are omitted. Input variables generated by linear links of neural network are linearly weighed. Functional links act on elements of input variables by generating a set of linearly independent functions. These links are evaluated as functions with the variables as the arguments. Figure 3.3 shows the single layered architecture of FLANN. FLANN architecture offers less computational overhead and higher convergence speed when compared with other ANN techniques.

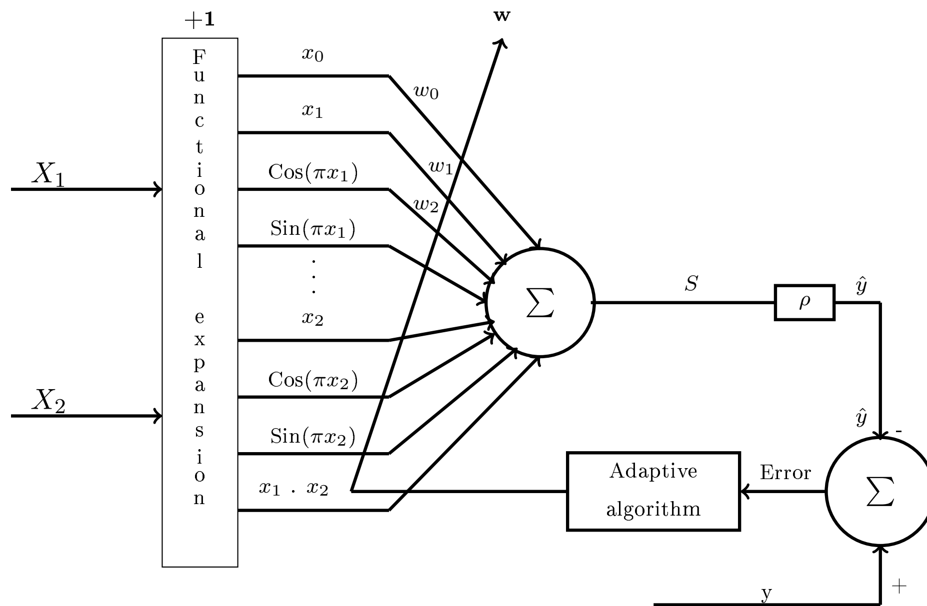

Figure 3.3: Flat net structure of FLANN

Using FLANN, output is calculated as:

$$\hat{y} = \sum_{i=1}^{n} W_i X_i \tag{3.24}$$

where $\hat{y}$ is the predicted value, W is the weight vector and X is the functional block which is defined as:

$$X = [1,\ x_1,\ sin(\pi x_1),\ cos(\pi x_1),\ x_2,\ sin(\pi x_2)\ , cos(\pi x_2), ...] \tag{3.25}$$

and weight is updated as:

$$W_i(k+1) = W_i(k) +\ \alpha\ e_i(k)\ x_i\ (k) \tag{3.26}$$

having 'α' as the learning rate and e_i as the error value, formulated as:

$$e_i = y_i - \hat{y}_i \tag{3.27}$$

here y and $\hat{y}$ represent actual and the obtained (predicted) value respectively.

3.3.5 Probabilistic Neural Network

Probabilistic Neural Network (PNN) was introduced by Donald F Specht [106]. It is a feedforward neural network, basically derived from Bayesian network and Statistical algorithm. Figure 3.4 shows the basic architecture of PNN.

In PNN, the network is organized as multilayered feedforward network with four layers: input, hidden, summation and output layer. The input layer first computes the distance from input vector to the training input vectors. The second layer consists of a Gaussian function which is formed using the given set of data points as centers. The summation layers sums contribution of each class of input and produces a net output which is vector of probabilities. The fourth layers determines the fault prediction rate.

PNN works more faster when compared to multilayer perceptron networks and also is more accurate. The major concern lies in finding an accurate

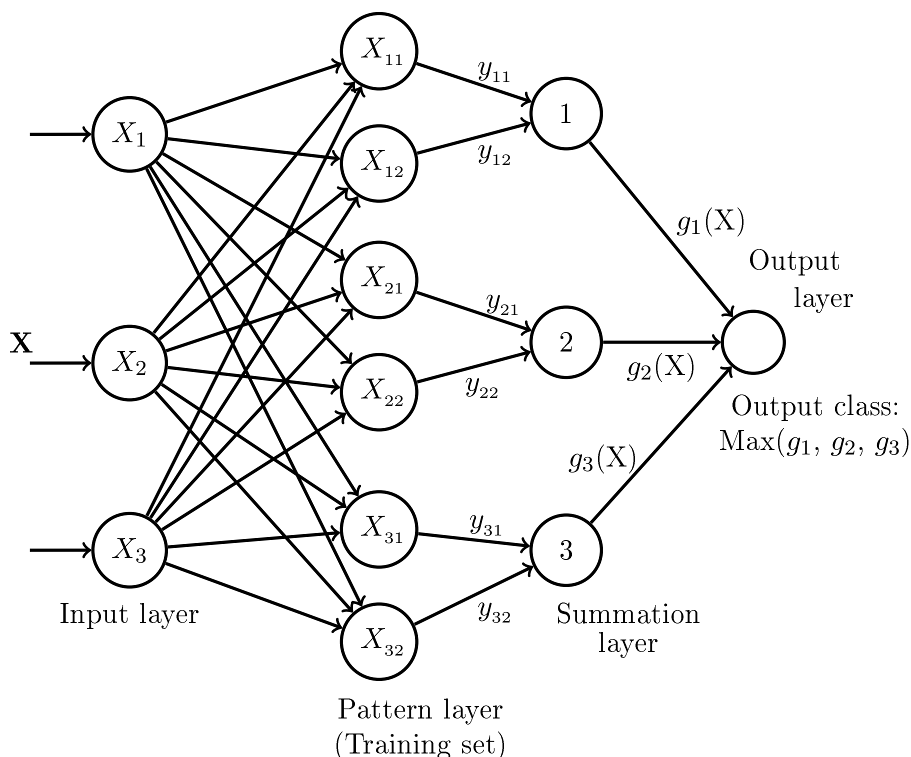

Figure 3.4: Basic architecture of PNN

smoothing parameter 'σ' to obtain better classification. The following function is used in hidden layer:

$$\phi(z) = e^{-(z^2/\sigma^2)} \tag{3.28}$$

where $z = ||x-c||$, x is the input, c is the center, and z is the euclidean distance between the center and the input vector.

3.4 Fault Prediction using Feature Reduction Techniques

In this fault prediction analysis, two feature reduction techniques such as Principal component analysis (PCA) and Rough set theory (RST) have been applied. PCA and RST attribute reduction techniques are applied on full feature set. This reduct set obtained is used in fault prediction by applying

various statistical and neural network techniques.

3.4.1 Application of Principal Component Analysis

Principal Component Analysis (PCA) is a statistical technique used to transform a data space of high dimension into a feature space of lower dimension having the most significant features. Features (or inputs) that have little variance are thereby removed. It is a standard technique to identify the underlying, orthogonal dimensions that explain relations between the variables in a data set. The concept of PCA was developed by Karl Pearson [107].

PCA is an orthogonal transformation of the coordinate system in which the data are represented. The new transformed coordinate values by which data are represented are referred to as principal components. A small number of principal components (PCs) are sufficient enough to represent most of the significant patterns in the data. These are also referred as factors or latent variables of the data. PCA rigidly rotates the axes of the p-dimensional space to new positions (principal axes) such that the highest variance is possessed by 'axis 1', and the second axis has the next highest variance and so on. The covariance among each pair of the principal axes is zero, and hence the principal axes are uncorrelated.

First, the covariance matrix 'S' is computed and the eigen values are found. The eigen values are then sorted in decreasing order, and denoted as: $\lambda_1 \geq \lambda_2 \geq ... \lambda_M$. The corresponding eigen vectors are denoted as a_1, a_2, ... a_M. The first 'd' eigen vectors are chosen from 'M' vectors such that d << M. Finally, the data set is projected into lower dimension, and is given as:

$$G \leftarrow [a_1, a_2, a_3, ...a_d] \quad where \quad d << M \tag{3.29}$$

if x is a test point

$$x \in R^M \rightarrow G^T x \in R^d \tag{3.30}$$

where, G represents a set of eigen vectors and R represents lower dimension data set.

3.4.2 Application of Rough Set Theory

Rough Set Theory (RST) is a formal approximation method of a conventional (CRISP) set [108]. This formal approximation, represents the lower and upper bound of the original set. Rough set helps in adequate analysis of various types of data, especially when dealing with inexact, vague and uncertain data. Rough set captures two unique features of imperfection in knowledge i.e., indiscernibility and vagueness.

Rough set execution is based on the concept that lowering the 'degree of precision' in the data set makes data pattern more visible [109]. In general, rough set approach can be viewed as a formal framework for mining facts from imperfect data. The results achieved by application of rough set concept can be represented in the form of classification, decision rules or inform of reduced data set. In this analysis of fault prediction, RST is used to obtain reduced data set.

3.5 Performance Evaluation Parameters

The following sub-sections give the basic definitions of the performance parameters used in statistical and machine learning methods for fault prediction.

The performance parameters for statistical analysis can be determined based on the confusion matrix [110] as shown in Table 3.3. The confusion matrix contains information about actual and predicted classifications done by a fault-prediction model.

Table 3.3: Confusion matrix to classify a class as faulty or not-faulty

	NO (Prediction)	**YES (Prediction)**
NO (Actual)	True Negative (TN)	False Positive (FP)
YES (Actual)	False Negative (FN)	True Positive (TP)

The confusion matrix has four categories:

1. True Positive (TP): refers to whether modules are correctly classified as faulty modules.
2. False Positive (FP): refers to not-faulty modules incorrectly labeled as faulty modules.
3. True Negative (TN): corresponds to not-faulty modules correctly classified as such.
4. False Negative (FN): refers to faulty modules incorrectly classified as not-faulty modules.

The following are the performance measures used in statistical analysis.

- Precision

  Is the degree to which the repeated measurements under unchanged conditions show the same results [110].

$$Precision = \frac{TP}{FP + TP} \tag{3.31}$$

- Correctness

  Correctness as defined by Briand *et al.* is the ratio of the number of modules correctly classified as fault prone to the total number of modules classified as fault prone [35].

$$Correctness = \frac{TP}{FP + TP} \tag{3.32}$$

- Completeness

  According to Briand *et al.*, completeness is the ratio of number of faults in classes classified as fault prone to the total number of faults in the system [35].

$$Completeness = \frac{TP}{FN + TP} \tag{3.33}$$

- Accuracy

  Accuracy as defined by Yuan *et al.* is the proportion of predicted fault prone modules that are inspected out of all modules [111].

$$Accuracy = \frac{TN + TP}{TN + FP + FN + TP} \tag{3.34}$$

- R^2 statistic

  R^2, known as coefficient of multiple determination, is a measure of power of correlation between predicted and actual number of faults [110]. Higher the value of this statistic, more is the accuracy of the predicted model. $R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$

  where y_i is the actual number of faults, $\hat{y}_i$ is the predicted number of faults, and $\overline{y}$ is the average number of faults.

All these performance parameters are computed in order to evaluate the result of goodness of regression models.

Fault prediction accuracy for four ANN models is determined by using performance evaluation parameters such as Mean Absolute Error (MAE), Mean Absolute Relative Error (MARE), Root Mean Square Error (RMSE) and Standard Error of the Mean (SEM).

- Mean Absolute Error (MAE)

  This performance parameter determines how close the values of predicted and actual fault (accuracy) rate differ.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - y_i'| \tag{3.35}$$

- Mean Absolute Relative Error (MARE)

$$MARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - y_i'|}{y_i} \tag{3.36}$$

In Equation 3.36, a numerical value of 0.05 is added in the denominator in order to avoid numerical overflow (division by zero). The modified MARE is formulated as:

$$MARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - y'_i|}{y_i + 0.05} \tag{3.37}$$

- Root Mean Square Error (RMSE)

  This performance parameter determines the differences in the values of predicted and actual fault (accuracy) rate differ.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - y'_i)^2} \tag{3.38}$$

  In Equation 3.36, 3.37 and 3.38; y_i is the actual value, and y'_i is the expected value.

- Standard Error of the Mean (SEM)

  It is the deviation of the predicted value from the actual fault (accuracy) rate.

$$SEM = \frac{SD}{\sqrt{n}} \tag{3.39}$$

  where SD is sample standard deviation and 'n' is the number of samples.

3.6 Results and Analysis

In this section, the relationship between value of metrics and the fault found in a class is determined. In this approach, the six CK metrics are used as input nodes, and the output is the achieved fault prediction rate for AIF version 1.6.

3.6.1 Fault Data

To perform statistical analysis, the values of the bugs are collected from Promise data repository [98]. Table 3.4 shows the distribution of bugs based on

the number of occurrences (in terms of percentage of class containing number of bugs) for Apache Integration Framework (version 1.6), considered here as a case study.

Table 3.4: Distribution of bugs for AIF version 1.6

# Classes	% of bugs	# of associated bugs
777	80.5181	0
101	10.4663	1
32	3.3161	2
16	1.6580	3
14	1.4508	4
6	0.6218	5
2	0.2073	6
3	0.3109	7
5	0.5181	8
1	0.1036	9
1	0.1036	10
3	0.3109	11
1	0.1036	13
1	0.1036	17
1	0.1036	18
1	0.1036	28
965	100.00	142

AIF version 1.6 contains 965 number of classes, in which 777 classes contain zero bugs (80.5181%), 10.4663% of classes contain at least one bug, 3.3161% of classes contain a minimum of two bugs, 1.6580% of classes contain three bugs, 1.4508% classes contain four bugs, 0.6218% of classes contain five bugs, 0.2073% of the classes contain six bugs, 0.3109% classes contain seven and eleven bugs, 0.5181% of classes contain eight bugs, 0.1036% of the class contain nine, thirteen, seventeen, eighteen and twenty eight bugs.

3.6.2 Metrics Data

CK metric values for WMC, DIT, NOC, CBO, RFC and LCOM respectively for AIF version 1.6 are graphically represented in Figure 3.5 to Figure 3.10.

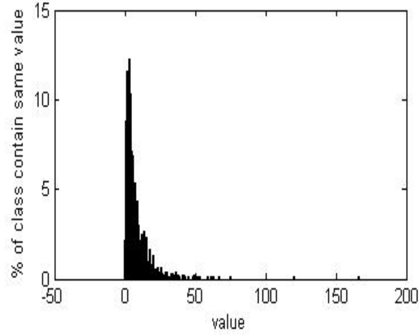

Figure 3.5: Histogram for WMC

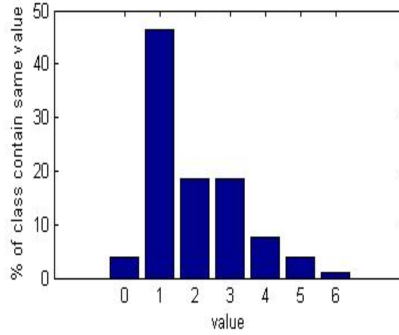

Figure 3.6: Histogram for DIT

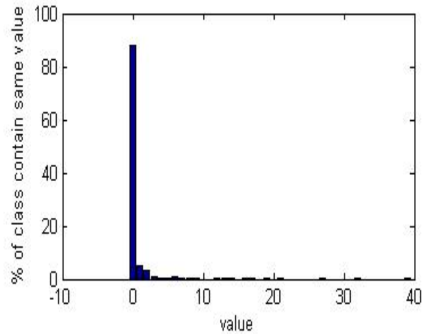

Figure 3.7: Histogram for NOC

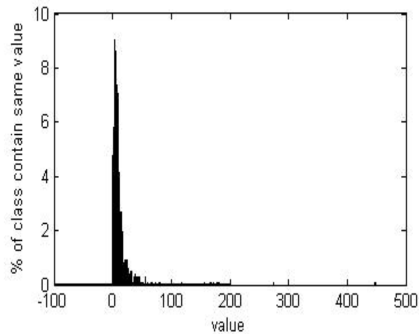

Figure 3.8: Histogram for CBO

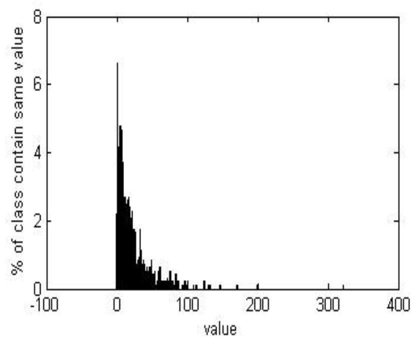

Figure 3.9: Histogram for RFC

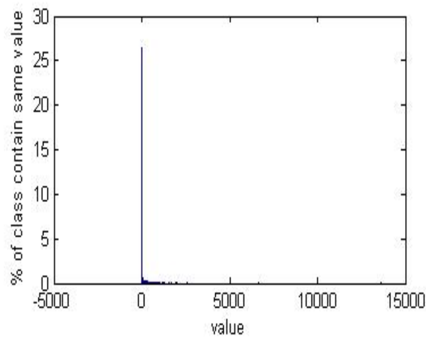

Figure 3.10: Histogram for LCOM

3.6.3 Descriptive Statistics and Correlation Analysis

This subsection presents the comparative analysis of the fault data, descriptive statistics of classes and the correlation among the six metrics with the results obtained by Basili *et al.* [97].

Basili *et al.* carried out an experiment for Object-Oriented systems written in C/C++, consisting of eight project groups each consisting of three students. Each group had the same task of developing small/medium-sized software system. Since all the necessary documentation (for instance, reports about faults and their fixes) were available, they could search for relationships between fault density and metrics. They used the same CK metric suite. Logistic regression is employed to analyze the relationship between metrics and the fault proneness of classes. Table 3.5 shows the comparative statistical analysis results obtained for Basili *et al.* and AIF version 1.6 for CK metrics indicating Max, Min, Median and Standard deviation.

Table 3.5: Descriptive statistics of classes

Basili *et al.* [97]	WMC	DIT	NOC	CBO	RFC	LCOM
Max.	99.00	9.00	105.00	13.00	30.00	426.00
Min.	1.00	0.00	0.00	0.00	0.00	0.00
Median	9.50	0.00	19.50	0.00	5.00	0.00
Mean	13.40	1.32	33.91	0.23	6.80	9.70
Std Dev.	14.90	1.99	33.37	1.54	7.56	63.77
AIF Version 1.6	WMC	DIT	NOC	CBO	RFC	LCOM
Max.	166.00	6.00	39.00	448.00	322.00	13617
Min.	0.00	0.00	0.00	0.00	0.00	0.00
Median	5.00	1.00	0.00	7.00	14.00	4.00
Mean	8.57	1.95	0.052	11.10	21.42	79.33
Std Dev.	11.20	1.27	2.63	22.52	25.00	523.75

The obtained CK metric values of AIF version 1.6 are compared with the results of Basili *et al.* [97]. In comparison to the work done by Basili *et al*, the total number of classes considered is much greater i.e., in this study 965 classes are considered as compared to 180 classes used by Basili *et al* [97]. Table 3.6 shows the *Pearson's* correlation for the data set used by Basili *et al.* [97] and the correlation metrics of AIF version 1.6.

Table 3.6: Correlations between metrics

Basili *et al.* [97]	WMC	DIT	NOC	CBO	RFC	LCOM
WMC	1.00	0.02	0.24	0.00	0.13	0.38
DIT		1.00	0.00	0.00	0.00	0.01
NOC			1.00	0.00	0.00	0.00
CBO				1.00	0.31	0.01
RFC					1.00	0.09
LCOM						1.00
AIF version 1.6	WMC	DIT	NOC	CBO	RFC	LCOM
WMC	1.00	0.00	0.03	0.10	0.77	0.60
DIT		1.00	0.00	0.00	0.00	0.01
NOC			1.00	0.024	0.025	0.027
CBO				1.00	0.08	0.05
RFC					1.00	0.42
LCOM						1.00

The dependency between CK metrics is computed using Coefficient of determination (R^2) and compared with Basili *et al.* [97] for AIF version 1.6. The coefficient of determination i.e., R^2, is useful because it gives the proportion of the variance (fluctuation) of one variable that is predictable from the other variable. It is a measure that allows a researcher to determine how certain one can be in making predictions from a certain model/graph.

From Table 3.6, for AIF version 1.6, it is observed that the correlation between WMC and RFC is 0.77 which indicates that they are highly correlated i.e., these two metrics are very much linearly dependent on each other. Similarly, correlation between WMC and DIT is 0 which indicates that they are loosely correlated i.e., there is no dependency between these two metrics.

3.6.4 Attribute Reduction

In this subsection, the procedure followed to obtain reduct data set by applying PCA and RST approaches are discussed.

3.6.4.1 Principal Component Analysis

Attribute reduction through Principal Component Analysis (PCA) is achieved by ignoring the directions in which the covariance is small. The following steps give the procedure followed in application of PCA to the data set. Algorithm 1 shows the steps followed to obtain the reduct data set using PCA.

Table 3.7 shows the obtained principal components for the CK metrics suite i.e., the relationship between the original object-oriented metrics and the PCA domain metrics. The initial size of the data set was 965×6 i.e., it represents 965 classes of AIF version 1.6 with their respective metric values of the CK metrics suite. After applying PCA, a transformed data values of size 965×3 was obtained. This implies that PCA data set contained 965 classes of CK metric values along with three principal component values of the PCA.

Algorithm 1 $PCA(\ )$

Input: 'n × m' feature matrix X where 'n' represents number of samples and 'm' represents the number of features.

Output: 'n ×k' reduced feature matrix ($k << m$).

Step 1: Matrix 'X' is normalized.

Let training set= $x^1, x^2 \cdots x^m$.

Evaluate $\mu_j = \frac{1}{n}\sum_{i=1}^{n} x_i^j$ vary j for all feature values i.e., 1 to m.

Replace x^j with $(x^j - \mu_j)$ vary x^j for all samples i.e., from 1 to n.

Step 2: Compute covariance matrix of the normalized matrix.

$\sum(sigma) = \frac{1}{m}(X^T X)$

Step 3: Compute the eigen vectors of matrix using MATLAB command as:

$eign = eig(sigma)$

Step 4: Choose the first 'k' number of principal components from the covariance matrix using the following criteria:

for *(every eigen vector i=1 to m)* **do**

Evaluate $cumvar = \frac{\sum_{i=1}^{k}\lambda_{ii}}{\sum_{i=1}^{m}\lambda_{ii}}$ {cumvar (denotes cumulative variance) and (λ) is eigen values in descending order}

if $(cumvar \geq 0.99) or (1 - cumvar \leq 0.01)$

return k {99% of variance is retained}

end if

end for

Step 5: Reduce the matrix dimension, taking the first k columns (1 to k) of eign matrix as eign(:,1:k) and assign to $eign_{red}$.

Step 6: Evaluate $Z = X * eign_{red}$.

where Z is the new matrix with reduced feature dimension retaining 99% of the variance.

Step 7: Stop.

Table 3.7: Principal components

	pc1	pc2	pc3
WMC	**0.2110**	-0.5837	0.1756
DIT	**0.2990**	-0.3665	**-0.2419**
NOC	0.0481	-0.0243	**0.4084**
CBO	0.0353	0.0609	-0.8560
RFC	-0.9210	-0.0137	0.0054
LCOM	**0.2530**	**0.7215**	0.1053
Eigen value	0.0490	0.0018	0.0058
Variance (%)	19.8985	19.8588	19.4579
Cumulative (%)	19.89	39.75	59.21

The values above 0.2 (shown bold in Table 3.7) are the metrics that are used to interpret the PCs. For each PC, the eigenvalue, variance percent, and cumulative percent is presented. Eigenvalue is associated with a principal component. When the sum of squared values of loadings relating to dimension is considered, then the sum is referred to as eigenvalue. Eigenvalue indicates the relative importance of each dimension for the particular set of variables analyzed. The interpretations of pc's are given as follows:

pc1: WMC, DIT and LCOM are size, inheritance and cohesion metrics of CK metric suite. The values indicate that classes exist in the module with high internal (methods defined in the class) and external methods (methods called by the class). The cohesion and coupling metrics are related to number of methods and attributes in the class.

pc2: LCOM, a cohesion metric which measures the dissimilarity in classes based on instanced variables.

pc3: NOC and DIT are inheritance metrics that count number of children and depth of inheritance tree in a class.

3.6.4.2 Rough Set Theory

In this analysis of fault prediction, the reduced feature set of the CK metrics suite is used. In order to apply rough set, the data extracted from a particular class requires the data to be classified. This classification of data is achieved by applying K-means clustering algorithm, where the data available in particular cluster are grouped under a single class. After obtaining this clustered data for the CK metrics suite, RST is applied.

Following are the steps followed to obtain reduced feature set using RST.

Step 1. Collection of data.

Data is extracted from Promise data repository [98].

Step 2. Discretization of data.

The data extracted from the repository is discretized by using K-means clustering algorithm.

Step 3. Lower and upper approximation of all possible set is calculated.

Lower approximation is defined as the union of all these elementary sets which are contained in X.

$$\underline{B}X = \{x_i \in U| \ [x_i]_{Ind(B)} \subset X\} \tag{3.40}$$

Upper approximation is the union of these elementary sets, which have a non-empty intersection with X.

$$\bar{B}X = \{x_i \in U| \ [x_i]_{Ind(B)} \cap X \neq 0\} \tag{3.41}$$

Step 4. Accuracy of all possible sets is calculated.

An accuracy measure of the set X in B $\subseteq$ A is defined as:

$$\mu_B = \frac{Card(\underline{B}X)}{Card(\bar{B}X)} \tag{3.42}$$

The cardinality of a set is the number of objects contained in the lower/ upper approximation of the set X.

Step 5. All possible sets are selected on the basis such that, their accuracy is equal to the accuracy of universal set.

Step 6. The set with least possible value of cardinality is chosen as reduct set from all possible selected set.

Rough set obtained a reduced feature set of four metrics out of the six metrics initially proposed in CK metric suite. The metrics such as DIT and NOC are omitted from the CK metric suite. Now, the prediction model to be designed for computing the required fault rate uses the reduced feature set of CK metrics suite (i.e., WMC, CBO, RFC, LCOM) obtained by applying RST.

3.6.5 Machine Learning Methods

Following machine learning methods are applied on three data sets viz., full feature set, and reduct data set of PCA and RST.

3.6.5.1 Statistical Methods

A. Linear Regression Analysis

In linear regression analysis, a fault is considered as a dependent variable and the CK metrics are taken as independent variables. Table 3.8 shows the results obtained for linear regression analysis using full feature data set.

Table 3.8: Linear regression analysis

r-value	p-value	Std. Error
0.5154	0.000	0.0834

In Table 3.8, 'r-value' represents the coefficient of correlation; 'p-value' refers to the significance of the metric value. If $p < 0.001$, then the metrics are of very great significance in fault prediction.

i. Results of Linear Regression Analysis using PCA

Table 3.9 and Table 3.10 show the coefficients of the features and performance parameters such as r-value, p-value and standard error respectively after applying PCA. From Table 3.10 it is evident that the ‘p’ value is less than 0.001, which highlights that six CK metrics are of very great significance in fault prediction.

Table 3.9: Coefficients of features for PCA based Linear regression analysis

	Feature-1	Feature-2	Feature-3	Constant
Coefficient	0.0271	0.0864	-0.0014	0.0205

Table 3.10: Linear regression analysis for AIF Version 1.6 after applying PCA

r-value	p-value	Std. Error
0.1094	6.6483e-04	2.3274e-04

ii. Results of Linear Regression Analysis using RST

When RST feature reduction technique is applied, two metrics viz., DIT and NOC are omitted from the CK metrics suite. The coefficients of the reduced metric suite consisting of WMC, CBO, RFC and LCOM are shown in Table 3.11. Table 3.12 shows the r-value, p-value and standard error for linear regression analysis after applying RST.

Table 3.11: Coefficients for Linear regression analysis after applying RST

	WMC	CBO	RFC	LCOM	Constant
Coefficient	0.0590	0.0323	7.9869e-04	-8.0090e-04	-0.2648

Table 3.12: Linear regression analysis for AIF Version 1.6 after applying RST

r-value	p-value	Std. Error
0.5124	1.0577e-65	0.1132

B. Logistic regression analysis

The logistic regression method helps to indicate whether a class is faulty or not, but does not convey anything about the possible number of faults in the class. Univariate and multivariate logistic regression techniques are applied to predict whether the class is faulty or not. Univariate regression analysis is used to examine the effect of each metric on on the faulty class while multivariate regression analysis is used to examine the common effectiveness of metrics on faulty classes. The results of AIF version 1.6 are compared considering these two statistical techniques. Figure 3.11 shows the typical 'S' curve obtained (similar to sigmoid function (f)) for the AIF version 1.6 using multivariate logistic regression.

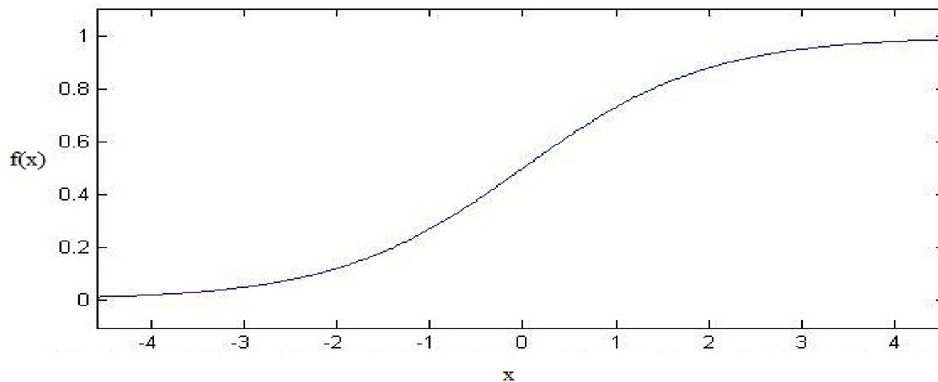

Figure 3.11: Logistic graph

Table 3.13 and Table 3.14 contain the tabulated values for the results obtained by applying univariate and multivariate regression analysis respectively. From Table 3.13, it can be observed that all metrics of CK suite are highly significant except for DIT. The p-value for AIF version 1.6 (w.r.t DIT) is 0.3527 respectively. Higher values of 'p' is an indication of less significance.

Table 3.13: Analysis of univariate regression for AIF Version 1.6

	Coefficient	Constant	p - value	r - value
WMC	0.03	-1.77	0.00	0.18
DIT	0.05	-1.53	0.3257	0.02
NOC	0.13	-1.50	0.00	0.16
CBO	0.02	-1.66	0.00	0.17
RFC	0.01	-1.79	0.00	0.17
LCOM	0.007	-1.48	0.0007	0.11

Table 3.14: Multivariate logistic regression analysis for AIF Version 1.6

	WMC	DIT	NOC	CBO	RFC	LCOM	CONSTANT
Coefficient	0.0320	0.00	0.00	0.001	0.0109	0.00	-2.1570

Univariate and multivariate logistic regression statistical methods are used for classifying a class as faulty or not faulty. Logistic regression applied with a threshold value 0.5 i.e., $\pi > 0.5$ indicates that a class is classified as 'faulty', otherwise it is categorized as 'not faulty' class.

Table 3.15 and Table 3.16 represent the confusion matrix for 'before' and 'after' applying regression respectively for AIF version 1.6. From Table 3.15 it is clear that before applying the logistic regression, a total number of 777 classes contain zero bugs and 188 classes contained at least one bug. After applying logistic regression (Table 3.16), a total of 783 (767+16) classes are correctly classified as not-faulty with a accuracy of 81.13%.

Table 3.15: Before applying regression

	Not-Faulty	Faulty
Not-Faulty	777	0
Faulty	188	0

Table 3.16: After applying regression

	Not-Faulty	Faulty
Not-Faulty	767	10
Faulty	172	16

The performance parameters for AIF version 1.6 are shown in Table 3.17, obtained by applying univariate and multivariate logistic regression. Here Precision, Correctness, Completeness and Accuracy [35], [37], [97], [112] are taken as performance parameters. By using multivariate logistic regression, accuracy of AIF version 1.6 is found to be 81.13%.

Table 3.17: Precision, Correctness, Completeness, and Accuracy for AIF version 1.6

	Precision (%)	Correctness (%)	Completeness (%)	Accuracy (%)
WMC	57.14	57.14	4.25	**81.71**
DIT	-	-	0	80.51
NOC	66.66	66.66	5.31	81.03
CBO	77.77	77.77	3.72	81.03
RFC	50	50	2.12	80.51
LCOM	60	0.6	1.59	80.62
MULTI	61.53	61.53	8.51	81.13

From the results obtained by applying linear and logistic regression analysis, it is found that out of the six metrics WMC appears to have more impact in predicting faults.

Now the effectiveness of the logistic regression analysis is compared with the use of reduced data set by applying two reduction techniques such as PCA and RST.

i. Results of logistic regression analysis using PCA

The results achieved by applying PCA for logistic regression analysis are as follows:

Table 3.18 shows the coefficients of features obtained through PCA for multivariate logistic regression analysis.

Table 3.18: Result of multivariate logistic regression

	Feature-1	Feature-2	Feature-3	Constant
Co-efficient	0.3781	1.2241	-2.8299	-1.4133

Table 3.19 shows the confusion matrix obtained after applying PCA. In comparison with Table 3.15, PCA is able to classify 779 (776+3) classes as not-faulty with an accuracy rate of 80.72 % (Table 3.19).

Table 3.19: Confusion matrix for PCA based regression

	Not-Faulty	Faulty
Not-Faulty	776	1
Faulty	185	3

The performance parameters of AIF version 1.6 obtained by applying PCA for logistic regression analysis is shown in Table 3.20.

Table 3.20: Precision, Correctness, Completeness, Accuracy for AIF Version 1.6 after applying PCA (in terms of %)

	Precision	Correctness	Completeness	Accuracy
Feature - 1	-	-	0	80.5181
Feature - 2	-	-	0	80.5181
Feature - 3	75	75	1.5957	80.7254
MULTI	75	75	1.5957	80.7254

Now the classification of faults in logistic regression analysis by applying RST is discussed as follows:

ii. Results of Logistic Regression using RST

Table 3.21 and Table 3.22 contain the tabulated values for results obtained

by applying univariate and multivariate logistic regression analysis respectively using RST reduced data set.

Table 3.21: Analysis of univariate regression for AIF Version 1.6 after applying reduct data of RST

	Coefficient	Constant	p - value	r - value
WMC	0.0350	-1.77	4.05e-04	0.03
CBO	0.02	-1.66	6.11e-08	0.17
RFC	0.01	-1.79	1.90e-08	0.0291
LCOM	0.007	-1.48	7.03e-04	0.0161

From Table 3.21, it can be observed that out of the four metrics set are highly significant except for LCOM. The p-value for the AIF versions 1.6 in case of LCOM is 7.03e-04. Higher values of ‘p’ is an indication of less significance.

Table 3.22: Result of multivariate logistic regression after applying RST

	WMC	DIT	NOC	CBO	RFC	LCOM	CONSTANT
Coefficient	0.0343	0.0128	0.0044	-4.3622e-04	0	0	-1.9702

Table 3.23 shows that after applying logistic regression analysis using the reduced data set of RST, a total of 781 (771+10) classes are correctly classified as not-faulty with a accuracy rate of 80.92 % when compared with that of 80.72 % of accuracy obtained with logistic regression of PCA data set and 81.13 % for full feature set. Table 3.24 shows the performance parameters for logistic regression by applying the reduct data set of RST.

Table 3.23: Confusion matrix for RST based regression

	Not-Faulty	Faulty
Not-Faulty	771	6
Faulty	178	10

Table 3.24: Precision, Correctness, Completeness, Accuracy for AIF Version 1.6 after applying RST (in terms of %)

	Precision	Correctness	Completeness	Accuracy
MULTI	62.50	62.50	5.31	80.93

3.6.5.2 Artificial Neural Network

Artificial Neural Network (ANN) is an interconnected group of nodes. In this work, three layers of ANN are considered, in which six nodes act as input nodes, nine nodes represent the nodes of hidden layer and one node acts as output node.

ANN is a three phase network; the phases are learning, validation and testing. In this analysis, 70% of total input pattern is considered for learning phase, 15% for validation and the rest 15% for testing.

In this experiment, six CK metrics are taken as input, and output is the fault prediction accuracy rate. The network is trained using Gradient Descent method and Levenberg Marquardt method.

A. Gradient Descent method

Gradient Descent (GD) method is used for updating the weights using Equation 3.15 and Equation 3.16 . Table 3.25 shows the performance metrics of AIF version 1.6. Figure 3.12 shows the convergence characteristics for gradient descent method the AIF version 1.6 case study.

Table 3.25: Accuracy prediction for Gradient Descent using full feature set

MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
0.0594	1.093	0.0617	-0.2038	0.0044	0.0048	94.04

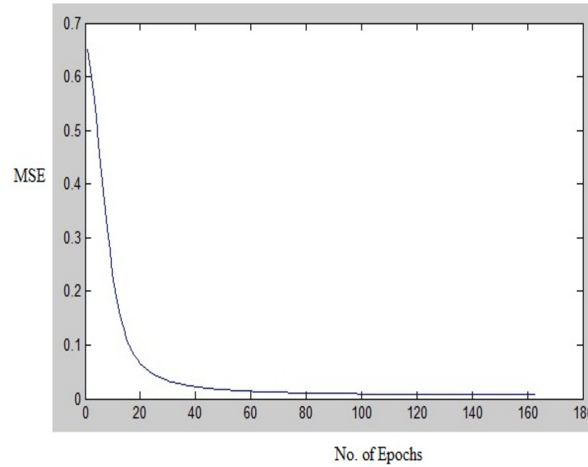

Figure 3.12: Convergence characteristics for Gradient Descent

Now, feature reduction techniques such as PCA and RST are applied on the six set metric suite, where PCA obtained three principal components, and RST obtained a reduct data set of four metrics (out of the six) in which DIT and NOC are omitted.

Table 3.26 shows the various performance parameters obtained for fault prediction for Gradient Descent using the reduced feature set obtained from PCA and RST respectively.

Table 3.26: Accuracy prediction for PCA and RST based Gradient Descent

Gradient Descent	MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
PCA	0.0245	0.2078	0.0324	0.1371	0.0567	6.1920e-04	97.55
RST	0.0254	0.2128	0.0450	0.2349	4.0879e-04	0.0059	97.43

Figure 3.13 and Figure 3.14 show the convergence characteristics for PCA and RST based Gradient Descent.

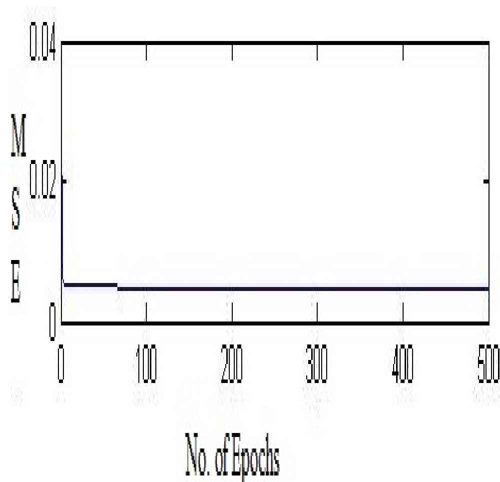

Figure 3.13: Convergence characteristics for PCA based Gradient Descent

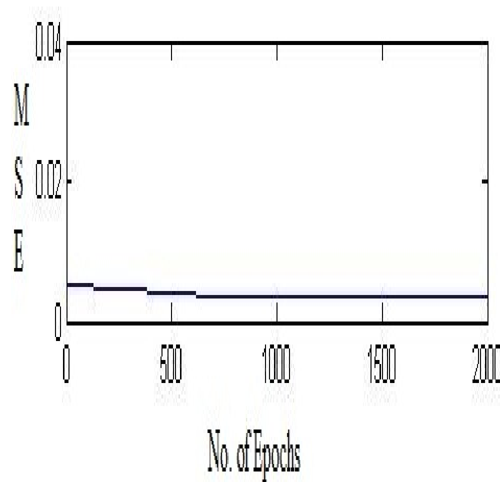

Figure 3.14: Convergence characteristics for RST based Gradient Descent

B. Levenberg Marquardt method

Levenberg Marquardt (LM) method is a technique for updating weights [101, 102]. In case of Gradient Descent method, learning rate α is constant but in Levenberg Marquardt method, learning rate α varies in every iteration. So this method consumes less number of iterations to train the network. Table 3.27 shows the performance metrics for AIF version 1.6 using LM method.

Table 3.27: Accuracy prediction for LM method using full feature set

MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
0.0023	1.1203	0.0308	-0.2189	0.0022	0.0041	90.49

Table 3.28 shows the various performance parameters obtained for fault prediction for Levenberg Marquardt NN using the reduced attribute set obtained from PCA and RST respectively. Figure 3.16 and Figure 3.17 show the convergence characteristics for PCA and RST based Levenberg Marquardt method respectively. Figure 3.15 shows the convergence characteristics of Levenberg Marquardt method for the AIF version 1.6 case study.

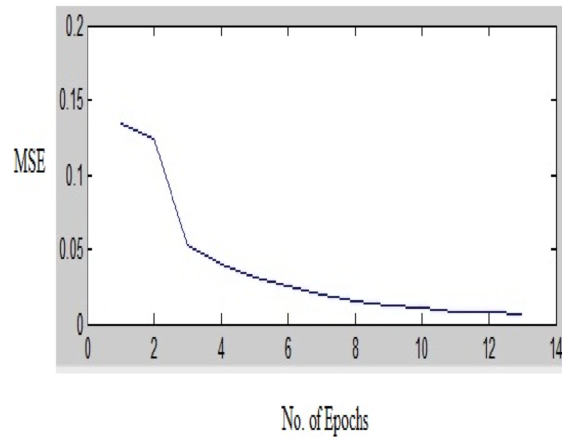

Figure 3.15: Convergence characteristics for Levenberg-Marquardt

Table 3.28: Accuracy prediction for PCA and RST based LM method

LM	MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
PCA	0.0114	0.0640	0.0328	0.1802	0.0119	8.3248e-06	98.85
RST	0.0145	0.1021	0.0301	-0.2191	0.0021	9.7489e-04	90.46

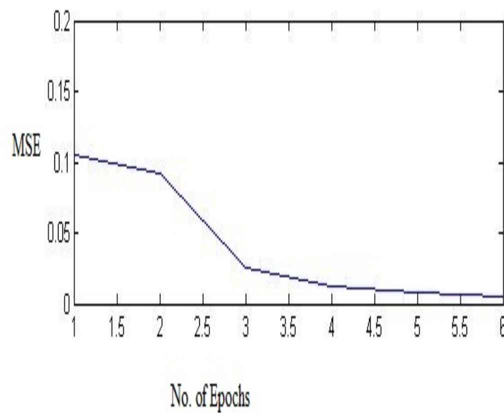

Figure 3.16: Convergence characteristics for PCA based Levenberg Marquardt

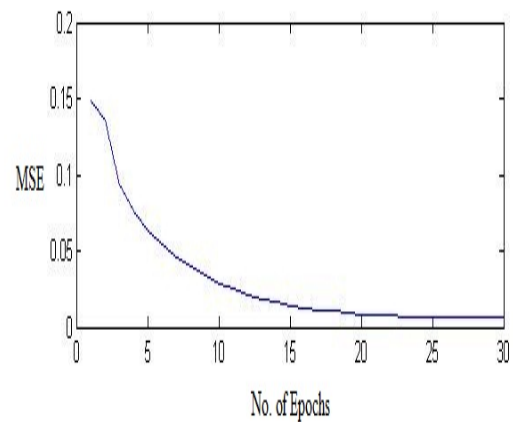

Figure 3.17: Convergence characteristics for RST based Levenberg Marquardt

3.6.5.3 Functional Link Artificial Neural Network

Functional Link Artificial Neural Network (FLANN) is a single-layer feed forward neural network consisting of an input and output layer. FLANN doesn't incorporate any hidden layer and hence has less computational cost. In this analysis, adaptive algorithm has been used for updating the weights as shown in Equation 3.26. Table 3.29 shows the performance metrics of FLANN and Figure 3.18 shows the convergence characteristics for FLANN.

Table 3.29: Accuracy prediction for FLANN using full feature set

MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
0.0304	0.7097	0.0390	0.3308	2.4601e-06	0.0050	96.37

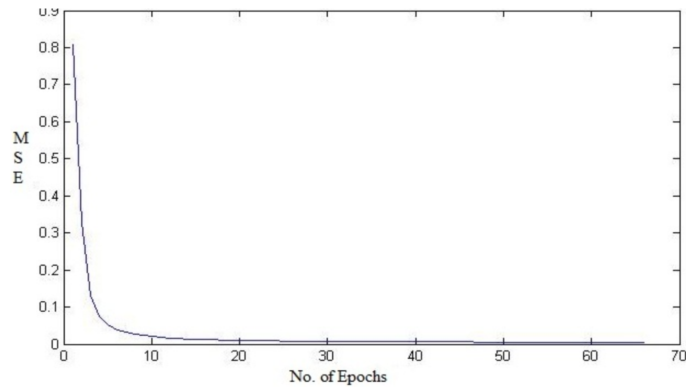

Figure 3.18: Convergence characteristics for FLANN

Table 3.30 shows the performance parameters for FLANN obtained using PCA and RST reduct data set.

Table 3.30: Accuracy prediction for PCA and RST based FLANN

FLANN	MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
PCA	0.0308	0.2708	0.0420	0.1830	0.0107	0.0040	94.48
RST	0.0325	0.2947	0.0393	0.1881	0.0086	0.0024	96.75

Figure 3.19 and Figure 3.20 show the convergence characteristics for PCA and RST based FLANN respectively.

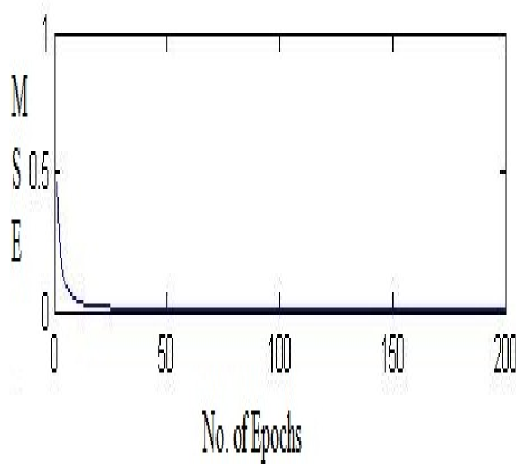

Figure 3.19: Convergence characteristics for PCA based FLANN

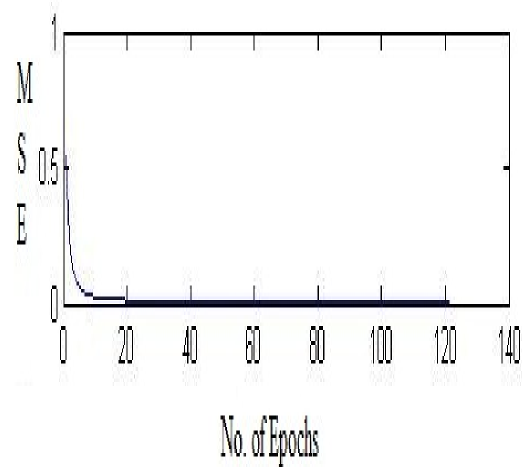

Figure 3.20: Convergence characteristics for RST based FLANN

3.6.5.4 Radial Basis Function Network

In Radial Basis Function Network (RBFN), Gaussian radial function is used as radial function. Gradient Descent learning and Hybrid learning methods are used for updating the centers and weights respectively.

A three layered RBFN is considered, in which six CK metrics are taken as input nodes, nine hidden centers are taken as hidden nodes and output is the fault prediction rate. Table 3.31 shows the performance metrics for AIF version 1.6.

Table 3.31: Accuracy prediction for Basic RBFN using full feature set

MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
0.0279	0.3875	0.0573	0.1969	0.059	0.006	97.27

Table 3.32 shows the performance metrics for AIF version 1.6 in case of Basic RBFN for the reduced feature set obtained by using PCA and RST.

Table 3.32: Accuracy prediction for PCA and RST based Basic RBFN

RBFN	MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
PCA	0.0269	0.2266	0.0377	-0.1736	0.0567	0.0013	97.30
RST	0.0262	0.2261	0.0475	0.2170	0.0024	0.0043	97.74

A. Gradient Descent Learning

Equation 3.21 and 3.22 are used for updating center and weight during training phase. After simplifying Equation 3.21, it is represented as:

$$C_{ij}(k+1) = C_{ij}(k) - \eta_1(y' - y)Wi\frac{\phi_i}{\sigma^2}(x_j - C_{ij}(k)) \tag{3.43}$$

and the modified Equation 3.22 is formulated as:

$$W_i(k+1) = W_i(k) + \eta_2(y' - y)\phi_i \tag{3.44}$$

where σ is the width of the center and k is the current iteration number. Table 3.33 shows the performance metrics for AIF version 1.6 and Figure 3.21 shows the convergence characteristics for Gradient RBFN.

Table 3.33: Accuracy prediction for Gradient RBFN using full feature set

MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
0.0207	0.2316	0.0323	0.3041	1.6302e-05	0.0041	97.24

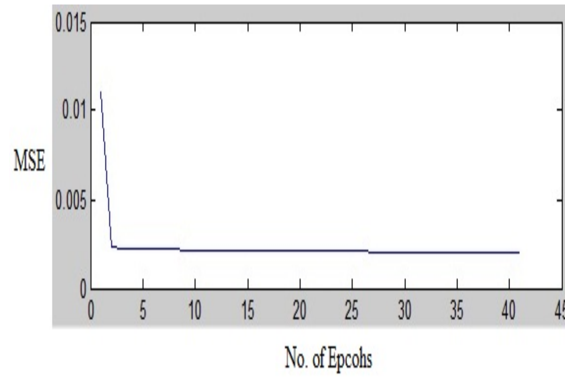

Figure 3.21: Convergence characteristics for Gradient RBFN

Table 3.34 shows the obtained performance metrics for Gradient RBFN using the reduced feature set of PCA and RST.

Table 3.34: Accuracy prediction for PCA and RST based Gradient RBFN

RBFN	MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
PCA	0.0229	0.1887	0.0340	0.1190	0.0983	0.0023	97.71
RST	0.0271	0.2266	0.0595	0.1503	0.0364	0.0054	97.29

Figure 3.22 and Figure 3.23 show the convergence characteristics for PCA and RST based Gradient RBFN respectively.

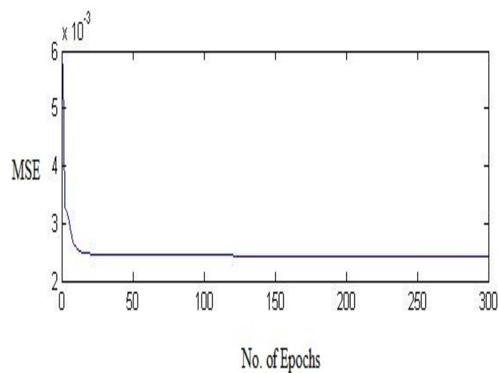

Figure 3.22: Convergence characteristics for PCA based Gradient RBFN

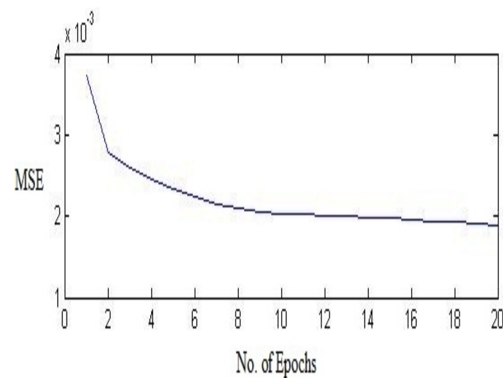

Figure 3.23: Convergence characteristics for RST based Gradient RBFN

B. Hybrid Learning

In Hybrid learning method, centers are updated using Equation 3.23 while weights are updated using supervised learning methods. In this analysis, Least Mean Square Error (LMSE) algorithm is used for updating the weights. Table 3.35 shows the performance matrix for AIF version 1.6. Figure 3.24 shows the convergence characteristics for Hybrid RBFN.

Table 3.35: Accuracy prediction for Hybrid RBFN using full feature set

MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
0.0614	0.1032	0.0316	0.9184	3.1834e-79	0.0013	98.47

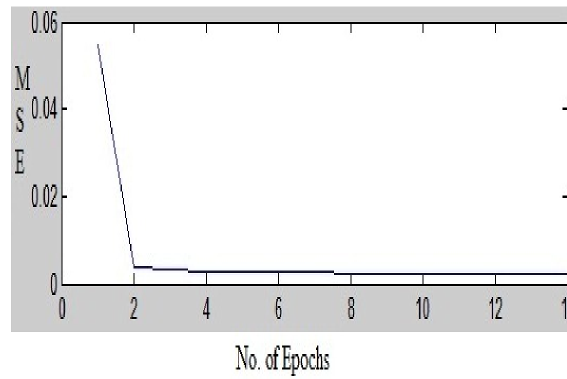

Figure 3.24: Convergence characteristics for Hybrid RBFN

Table 3.36 shows the obtained performance metrics for Hybrid RBFN using the reduced attribute set of PCA and RST.

Table 3.36: Accuracy prediction for PCA and RST based Hybrid RBFN

Hybrid RBFN	MAE	MARE	RMSE	r-value	p-value	Std. Error	Accuracy (%)
PCA	0.0150	0.1016	0.0329	-0.1572	0.0286	3.9944e-04	98.49
RST	0.0145	0.1021	0.0301	0.2905	3.9610e-05	9.7489e-04	98.54

Figure 3.25 and Figure 3.26 show the convergence characteristics for PCA and RST based Hybrid RBFN respectively.

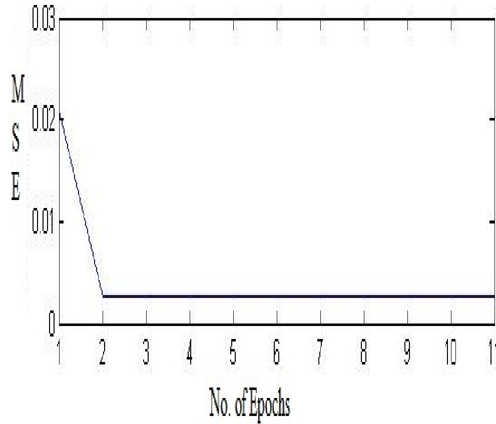

Figure 3.25: Convergence characteristics for PCA based Hybrid RBFN

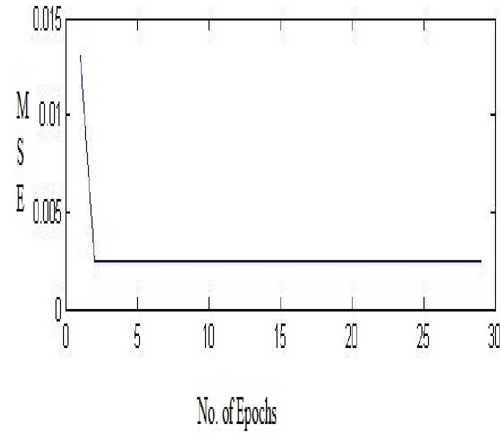

Figure 3.26: Convergence characteristics for RST based Hybrid RBFN

3.6.5.5 Probabilistic Neural Network (PNN)

As mentioned in section 3.3.5, PNN is multi-layered feed forward network with four layers such as input, hidden, summation and output layer.

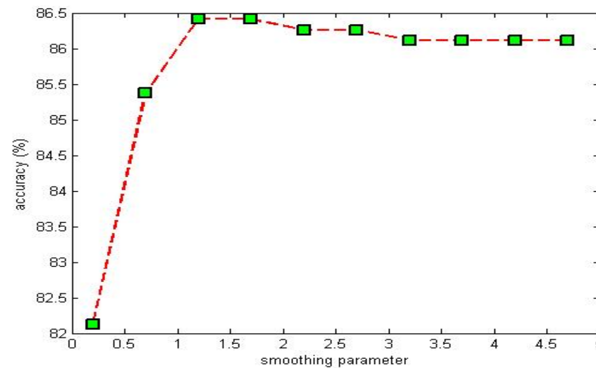

Figure 3.27: Varying accuracy rate for smoothing parameter in PNN

Figure 3.27 shows the variation of accuracy for different values of smoothing parameter for full feature set. In PNN, 50% of faulty and non-faulty classes are taken as input for hidden layers. Guassian elimination (Equation 3.28) is used as an hidden node function. The summation layers sums contribution

of each class of input patterns and produces a net output which is vector of probabilities. The output pattern having maximum summation value is classified into respective class.

Figure 3.28 and Figure 3.29 show the variation of accuracy for change in smoothing value parameters for the data set used by applying PCA and RST.

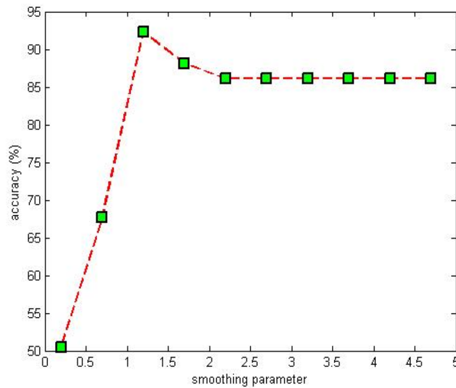

Figure 3.28: Varying accuracy rate for smoothing parameter in PCA based PNN

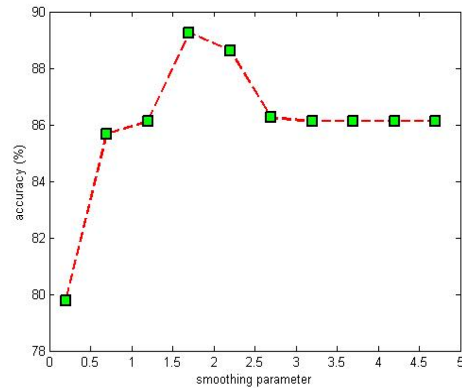

Figure 3.29: Varying accuracy rate for smoothing parameter in RST based PNN

3.6.6 Comparison of Fault Prediction Models

Table 3.37 shows the tabulated results for the obtained performance parameter values, number of epochs and accuracy rate by applying four neural network techniques for full feature set. This performance table is an indication of better fault prediction model.

In this comparative analysis, the performance parameter Mean Square Error (MSE) is considered as a criterion to compare the fault prediction accuracy rate of the models (based on MARE, MSE, number of epochs and accuracy rate) when four neural network models are applied. During this process the MSE value of 0.002 is set as a threshold for evaluation. Based on the number of iterations and the accuracy rate obtained by the respective NN model, best prediction model is determined.

Table 3.37: Performance parameters for fault prediction models

Prediction Model	Epoch	MAE	MARE	RMSE	Std. Error	Accuracy
Gradient Descent	162	0.0594	1.0930	0.0617	0.0048	94.04
LM	13	0.0023	1.1203	0.0308	0.0041	90.49
RBFN Basic	-	0.0279	0.3875	0.0573	0.0600	97.27
RBFN Gradient	41	0.0207	0.2316	0.0323	0.0041	97.24
RBFN Hybrid	14	0.0614	0.1032	0.0316	0.0013	98.47
FLANN	66	0.0304	0.7097	0.0390	0.0050	96.37

From Table 3.37 it is evident that, Gradient NN obtained an accuracy rate of 94.04% in 162 epochs (iterations). LM technique, which is an improvised model of ANN obtained 90.4% accuracy rate. This accuracy rate is less than Gradient NN but this approach (LM method) consumed only 13 epochs.

The three types of RBFN viz., Basic RBFN, Gradient and Hybrid methods obtained a prediction rate of 97.27%, 97.24% and 98.47% respectively. Considering the number of epochs, RBFN Hybrid method obtained better prediction rate of 98.47% in only 14 epochs when compared with Gradient (41 epochs) and Basic RBFN approaches.

FLANN architecture obtained 96.37% accuracy rate with less computational cost involved. FLANN obtained accuracy rate in 66 epochs as it has no hidden layer involved in its architecture.

The performance of PNN is shown in Figure 3.27 for full feature set. Highest accuracy in prediction is obtained for a smoothing parameter value of 1.7. PNN obtained a classification rate of 86.41%.

RBFN using hybrid learning model gives the least values for MAE, MARE, RMSE and high accuracy rate. Hence from the ANN analysis it can be concluded that RBFN Hybrid approach obtained the best fault prediction rate in less number of epochs when compared with the other three ANN techniques.

3.6.6.1 Comparison with PCA and RST

The criteria followed in comparing the fault prediction models in case of PCA and RST is the same as that followed in analyzing fault prediction model designed using full feature data set i.e., MSE value of 0.002 is set as a threshold for evaluation. Based on the number of iterations and the accuracy rate obtained by the respective NN model, best prediction model is determined.

Table 3.38 and Table 3.39 show the tabulated results for obtained performance parameter values, number of epochs and accuracy rate in case of PCA and RST reduct set respectively for the various machine learning methods used in this comparative analysis.

Table 3.38: Performance parameters for PCA based fault prediction models

Prediction Model	Epoch	MAE	MARE	RMSE	Std. Error	Accuracy
Gradient Descent	500	0.0245	0.2078	0.0324	6.1920e-04	97.55
LM	06	0.0114	0.0640	0.0328	8.3248e-06	98.85
RBFN Basic	-	0.0269	0.2266	0.0377	0.0013	97.30
RBFN Gradient	300	0.0229	0.1887	0.0340	0.0023	97.71
RBFN Hybrid	11	0.0150	0.1016	0.0329	3.9944e-04	98.49
FLANN	200	0.0308	0.2708	0.0420	0.0040	94.48

Table 3.39: Performance parameters for RST based fault prediction models

Prediction Model	Epoch	MAE	MARE	RMSE	Std. Error	Accuracy
Gradient Descent	500	0.0254	0.2128	0.0450	0.0059	97.43
LM	30	0.0145	0.1021	0.0301	9.7489e-06	90.46
RBFN Basic	-	0.0262	0.2261	0.0475	0.0043	97.74
RBFN Gradient	20	0.0271	0.2266	0.0595	0.0054	97.29
RBFN Hybrid	29	0.0145	0.1021	0.0301	9.7489-04	98.54
FLANN	122	0.0325	0.2947	0.0393	0.0024	96.75

Table 3.40 shows the tabulated fault prediction accuracy rate obtained for three data sets viz., full feature data set, PCA reduct data set and RST reduct data set respectively. The CK metrics suite with full feature data set and reduct data set (of PCA and RST) are used as requisite input for different machine learning methods.

Table 3.40: Comparison of fault prediction accuracy for three data sets

NN Model	Full feature set	PCA	RST
Gradient Descent	94.04	97.55	97.43
LM	90.49	98.85	90.46
RBFN Basic	97.27	97.30	97.74
RBFN Gradient	97.24	97.71	97.29
RBFN Hybrid	**98.47**	**98.49**	**98.54**
FLANN	96.37	94.48	96.75
PNN	86.41	88.12	89.12

From Table 3.40, it is evident in all the three cases i.e., use of full feature data set, PCA reduct data set and RST reduct data set, the Hybrid approach of RBFN obtained better fault prediction rate of 98.47, 98.49 and 98.54 in less number of epochs.

3.6.7 Comparison with existing methods

This section compares the obtained results of the proposed software fault prediction analysis with the existing models in literature [113].

Table 3.41: Accuracy comparison: Implemented models vs Existing models

Modeling Technique	**Accuracy**
SVM	86.9
Logistic Regression	86.7
C4.5 + PART	93.4
Neural network	91.6
Decorate C4.5	91.5
C4.5	91.2
Boost C4.5	90.3
PART	90.1

From Table 3.41 (in comparison with Table 3.40), it can be inferred that the proposed Gradient based ANN approach obtained better accuracy rate when compared to other models existing in literature such as support vector machine (SVM), Boost C4.5, PART etc. Where as in case of logistic regression the proposed methods obtained a bit lesser accuracy rate.

3.7 Complexity analysis of prediction models

This section provides the complexity analysis for the prediction models such as:

- Artificial Neural Network (ANN)
- Radial Basis Function Network (RBFN)
- Functional Link Artificial Neural Network (FLANN)
- Probabilistic Neural Network (PNN)
- Principal Component Analysis (PCA)

Figure 3.30 shows the architecture of Artificial Neural Network (ANN) model, which contains three layers viz., input layer (n_1), hidden layer (n_2) and output layer (n_3).

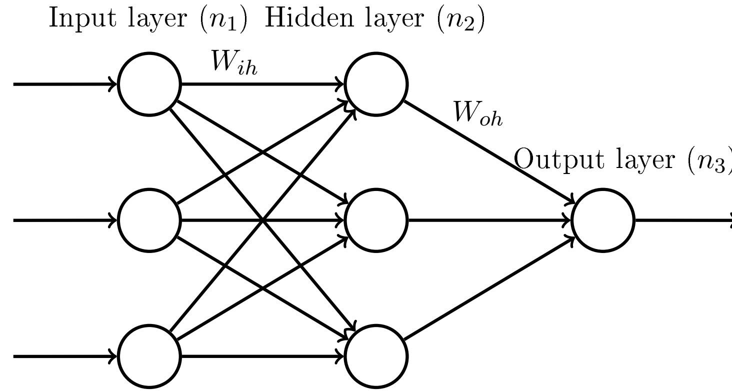

Figure 3.30: A typical feed forward neural network

The complexity analysis for these prediction models is discussed as follows:

Consider a neural network predictor with an architecture of three layers ($n_1 - n_2 - n_3$). While computing the complexity of the given model, it is necessary to know about the number of layers in it. In this work, a network with three layers was considered (except for PNN which contains four layers).

Given that the last layer only contains one neuron (output node) which is used to output the prediction of the input, only two 2 layers will be left out to distribute the remaining n-1 neurons (out of 'n' - total number of neurons). So, the number of multiplications needed to compute the activation of all neurons (vector product) in the i-th layer of the net equals: number of nodes in layer (n)* number of nodes in layer (n-1). The worst case would be that, if the nodes will be equally distributed, then complexity will be equal to:

$$= (\tfrac{n-1}{2}) * (\tfrac{n-1}{2})$$

$$= (\tfrac{n-1}{2})^2$$

$$= \mathcal{O}(n^2)$$

As the output functions of neurons are in general very simple to compute, it is assumed those costs to be a constant for every neuron. Given a network with n neurons, this step would be in $\mathcal{O}(n)$. Table 3.42 shows the complexity analysis expressions for various prediction models implemented in this thesis.

Table 3.42: Complexity expression for prediction models

Prediction Model	**Complexity expression**
Gradient ANN & Gradient RBFN	$\mathcal{O}(epoch * iterations * (n_1 n_2 + n_2 n_3))$
LM ANN & Hybrid RBFN	$\mathcal{O}(epochs(n_1 n_2 + m^3))$
FLANN	$\mathcal{O}(epochs * iterations * n_1 n_2)$
PNN	$\mathcal{O}(epochs * iterations * (n_1 n_2 + n_2 n_3))$
PCA	$\mathcal{O}(mn^2 + f^3)$

In Table 3.42, ‘m’ denotes number of classes (samples) and ‘f’ denotes number of features.

3.8 Threats to validity

This chapter has some limitations, which are not unique to our literature study but are common with most of the empirical studies in the literature.

Some of the threats to validity are:

- In this study, the results obtained are based on the historical data of one open source software, i.e., Apache Integration Framework (AIF, version 1.6), which has specific set of metrics (CK metric suite), and could not be generalized.
- AIF is designed in Java language. The models designed in this study are likely to be valid for other object-oriented programing language. Further, work can be extended in designing a based on other development paradigms.

- In this study, only CK metrics suite is used to design a model as it covers the basic principles of object-oriented paradigm such as complexity, inheritance and data abstraction. Some of the metrics which are widely used for object-oriented software can be further considered for fault prediction analysis.
- Softwares developed, depend on several psychological factors such as the expertise the developers have, the standard in which the software is developed, what kind of developers etc. In this study, these factors are not taken into account.
- In this study, the severity of faults is not taken into account (non-availability of data set w.r.t severity of faults). Severity of faults can vary from Mild to Infectious (on a scale of 1 to 10)

3.9 Relation between fault prediction and test data generation

The relationship between software fault prediction and test data generation approaches presented in the dissertation are as follows:

- Software fault prediction helps us to estimate the cost involved in removing the defects or to detect the number of fault prone components of a software. Further, this will help in effective utilization of resources to those defective components which are most likely to contain defects in testing and maintenance phases.
- This implies "greater the impact of fault prediction analysis, lesser is the effort & cost involved in testing". This enhances the software quality.
- "Fault-prone modules can be automatically identified before testing phase by using the software fault prediction models. Generally, these models

mostly use software metrics of earlier software releases and previous fault data collected during the testing phase".

- In this thesis, fault prediction was carried AIF with respect to both full feature set and reduced feature set. In the next phase, test data is generated for Bank ATM withdrawal task.

3.10 Summary

Prediction models are used to classify fault prone classes as faulty or not faulty, which is the needed for improving the whole testing process. In this chapter, machine learning techniques were applied for fault prediction. The application of machine learning methods in fault prediction used a good amount of data as input. A case study of AIF version 1.6 was considered for fault prediction analysis.

Machine learning methods such as Statistical methods (linear regression, logistic regression), and neural network methods such as ANN, FLANN, RBFN (RBFN Basic, RBFN Gradient, RBFN Hybrid), PNN techniques were applied for fault prediction analysis. Here, both full feature data set of CK metrics suite and reduct data set obtained by using feature reduction techniques such as PCA and RST were used as requisite inputs to the prediction models.

It can be concluded from the statistical regression analysis that out of six CK metrics, WMC appears to be more useful in predicting faults. When the reduct data set of PCA and RST were used, LCOM appears to have more impact in predicting faults. Table 3.37 shows that Hybrid approach of RBFN obtained better fault prediction in less number of epochs (14 iterations) when compared with the other three neural network models (ANN, FLANN, and PNN). Also the effectiveness of feature reduction techniques such as PCA and RST can be noticed from Table 3.40. The reduct data set obtained by using RST in case of Hybrid RBFN, obtained better fault prediction accuracy rate (98.54%) when compared with other data sets (full feature set as well as PCA).

Chapter 4

Cost-Based Evaluation Framework for Software Fault Classification

---

Cost-based evaluation framework is necessary to assess the usability of designed fault prediction models. In this chapter, classification of faults using logistic regression and various neural network models as classifiers is discussed in detail. Data classification techniques help in enhancing not only the efficiency of the training process, but also the performance of the predictive model in terms of precision. The proposed approach is applied on a case study discussed in previous chapter viz., Apache Integration Framework version 1.6. Fault prediction is found to be useful where normalized estimated fault removal cost (NEcost) is less than certain threshold value.

4.1 Introduction

Effectiveness of fault-prediction is studied by applying a part of previously known data related to faults and predicting its performance against other part of the fault data.

Several researchers have worked on building prediction models for software fault prediction. But, it was noticed that proving the effectiveness of a fault prediction model needs further study. Table 4.1 lists the proposed criteria considered by various authors in the design of their respective cost evaluation framework.

Table 4.1: Cost evaluation framework for fault classification

Author	**Cost evaluation criteria**
Ostrand (2005)	Performed prediction using defect densities and concluded that this will be able to find more defects in a fixed percentage of code [114].
Jiang (2008)	Introduced cost curve based on Receiver Operating Characteristic [115].
Lessmann (2008)	Identified a common evaluation framework based on ROC Curve (AUC) [116] and used the proposed concept of Demsar [117] to compare the performance of models.
Mende (2009)	Introduced a performance measure (P_{opt}) and compared prediction model with an optimal model. P_{opt} accounted module size to evaluate the performance of a fault-prediction model [118].
Mende (2010)	Proposed two strategies namely AD (effort-aware binary prediction) and DD (effort-aware prediction based on defect density) to include the notion of effort awareness into fault-prediction model [119].
Arisholm (2010)	Proposed a cost performance measure - Cost Effectiveness (CE), a variation of lift charts where the x-axis contains the ratio of lines of code instead of modules [113].

The table emphasizes on the studies carried out by different authors to compare the techniques, and the evaluation criterion considered in choosing an effective fault classification model. This chapter aims to assess the influence of classifier models in predicting faults by using CK metrics as requisite input to the prediction models, to put the results of a fault-prediction technique in

proper perspective. Also an attempt has been made to assess the influence of fault removal cost to know whether performing fault prediction analysis is useful or not.

4.2 Cost-Based Evaluation Framework

In literature it is observed that, the work done on classifying the object-oriented classes as a faulty or not-faulty one has been carried out by numerous authors. This can be viewed as a "two class" classification problem. The objective of this problem is to identify the *dependent variable* (accuracy) using various classifier models based on several *independent variables*. Independent variables can be considered as some sort of metrics or combination of different metrics.

This section describes the construction of a cost evaluation framework, which accounts for realistic cost required to remove a fault and computes the estimated fault removal cost of a specific fault prediction model based on the concept proposed by Wagner [120]. Wagner has designed the cost-based evaluation framework based on certain constraints, as mentioned below:

i. Different phases of testing account for varying fault removal cost.

ii. No testing phase can detect 100 % faults.

iii. It is not practically possible to perform unit testing on all modules, so a limited number of important logical paths should be selected, and testing should be exercised to selectively ensure proper working of the software to be delivered [4].

Fault removal cost approach suggested by Wagner *et al.,* [120] is used to formulate the proposed cost evaluation framework. Since different projects are developed on varying platforms and in varying organization standards, the cost varies. The fault removal costs summarized by Wagner are shown in Table 4.2.

Table 4.2: Removal costs of test techniques (in staff hour per defect)

Type	Min	Max	Mean	Median
Unit	1.5	6	3.46	2.5
System	2.82	20	8.37	6.2
Field	3.9	66.6	27.24	27

The fault identification efficiencies for different testing phases are taken from the study of Jones [121]. The efficiencies of testing phases are summarized in Table 4.3. Wilde *et al* stated that more than fifty percent of modules are usually very small in size, hence performing unit testing on these modules is not fruitful [122].

Table 4.3: Fault identification efficiencies of different test phases

Type	Min	Max	Median
Unit	0.1	0.5	0.25
System	0.25	0.5	0.65

The formulation of Ecost, Tcost and the NEcost of the proposed cost based evaluation framework is described in the following subsections:

4.2.1 Estimated fault removal cost (E_{cost})

The detailed analysis to compute Ecost of the framework is as follows:

- The total number of faulty classes identified by the predictor is equal to the summation of true positive (TP) and false positive (FP) values. Hence, it is necessary to compute testing and verification cost at class level which imply that this cost is equal to the cost of unit testing (C_u). Total unit testing cost of software system is defined using Equation 4.1.

$$Cost_{unit} = (TP + FP) * C_u \tag{4.1}$$

- As it is not possible to identify 100% faults in a specific testing phase, so there is a possibility that some of the correctly predicted faulty classes remain undetected in unit testing. Further, there is a scope that these faulty classes and the faulty classes which were predicted as non-faulty classes (count of false negative (FN)) can be identified by the predictor in later phases of testing such as system testing (C_s) and field testing. The fault removal cost in system testing is computed using Equation 4.2.

$$Cost_{system} = \delta_s * C_s * (FN + (1 - \delta_u) * TP) \tag{4.2}$$

  where, δ_u and δ_s represent fault identification efficiency of unit testing and fault identification efficiency of system testing respectively.

- Remaining faulty classes which were not identified in system testing will be further identified in field testing. The fault removal cost in case of field testing (C_f) is computed using Equation 4.3.

$$Cost_{field} = (1 - \delta_s) * C_f * (FN + (1 - \delta_u) * TP) \tag{4.3}$$

- The estimated overall fault removal cost can be determined by using Equation 4.4.

  ie., Ecost $= Cost_{unit} + Cost_{system} + Cost_{field}$

$$\begin{aligned} Ecost = C_i + C_u * (FP + TP) \\ + \delta_s * C_s * (FN + (1 - \delta_u) * TP) \\ + (1 - \delta_s) * C_f * (FN + (1 - \delta_u) * TP) \end{aligned} \tag{4.4}$$

4.2.2 Estimated testing cost (T_{cost})

The detailed steps to compute Tcost of the framework are as follows:

- In testing, if fault prediction analysis is not carried out, then the tester will perform unit testing on all the classes. So the total cost of unit testing will be computed using Equation 4.5.

$$Cost_{unit} = M_p * C_u * TC \quad (4.5)$$

  where, M_p represents the percentage of classes unit tested.

- Further, there is a possibility that some of the faulty classes that remained undetected in unit testing may be identified in system testing. The total system testing cost is computed using Equation 4.6.

$$Cost_{system} = \delta_s * C_s * (1 - \delta_u) * FC \quad (4.6)$$

- Remaining faulty classes which were not identified in system testing will be further identified in field testing. The fault removal cost in case of field testing is computed using Equation 4.7.

$$Cost_{system} = \delta_s * C_s * (1 - \delta_u) * FC \quad (4.7)$$

- The estimated overall fault removal cost without the use of fault prediction can be determined by using Equation 4.8.

$$\begin{aligned} Tcost = M_p * C_u * TC \\ + \delta_s * C_s * (1 - \delta_u) * FC \\ + (1 - \delta_s) * C_f * (1 - \delta_u) * FC \end{aligned} \quad (4.8)$$

4.2.3 Normalized fault removal cost (NE_{cost})

Normalized fault removal cost and its interpretation is shown in Equation 4.9.

$$NEcost = \frac{Ecost}{Tcost} = \begin{cases} < 1, & Fault\ Prediction \\ & is\ useful \\ \\ => 1, & Perform \\ & Testing \end{cases} \tag{4.9}$$

where, Tcost is the Estimated fault removal cost of the software without using fault prediction. Ecost represents the Estimated fault removal cost of the software when fault prediction is performed. NEcost represents the Normalized Estimated fault removal cost of the software when fault prediction is utilized.

The other notations in this cost evaluation framework are as follows:

i. C_i: Initial setup cost of used fault-prediction technique, C_u: Normalized fault removal cost in unit testing, C_s: Normalized fault removal cost in system testing, C_f: Normalized fault removal cost in field testing.

ii. M_p: percentage of classes unit tested.

iii. FP: Number of false positive, FN: Number of false negative, TP: Number of true positive, TN: Number of true negative, TC: Total number of classes, FC: Total number of faulty classes.

iv. δ_u: Fault identification efficiency of unit testing, δ_s: Fault identification efficiency of system testing.

In this experiment, the values tabulated in Table 4.2 are used in designing cost evaluation framework. δ_u and δ_s show the fault identification efficiency of unit testing and system testing respectively. The values of δ_u, and δ_s are collected from the survey report "Software Quality in 2010" of Caper Jones [121]. M_p shows the fraction of modules unit tested, obtained from the paper

of Wilde [122]. Median values have been chosen in this cost-based analysis. The objective is to provide the benchmarks to approximate the overall fault removal cost. Figure 4.1 shows the flow chart opted for the proposed cost-based evaluation framework for software fault classification.

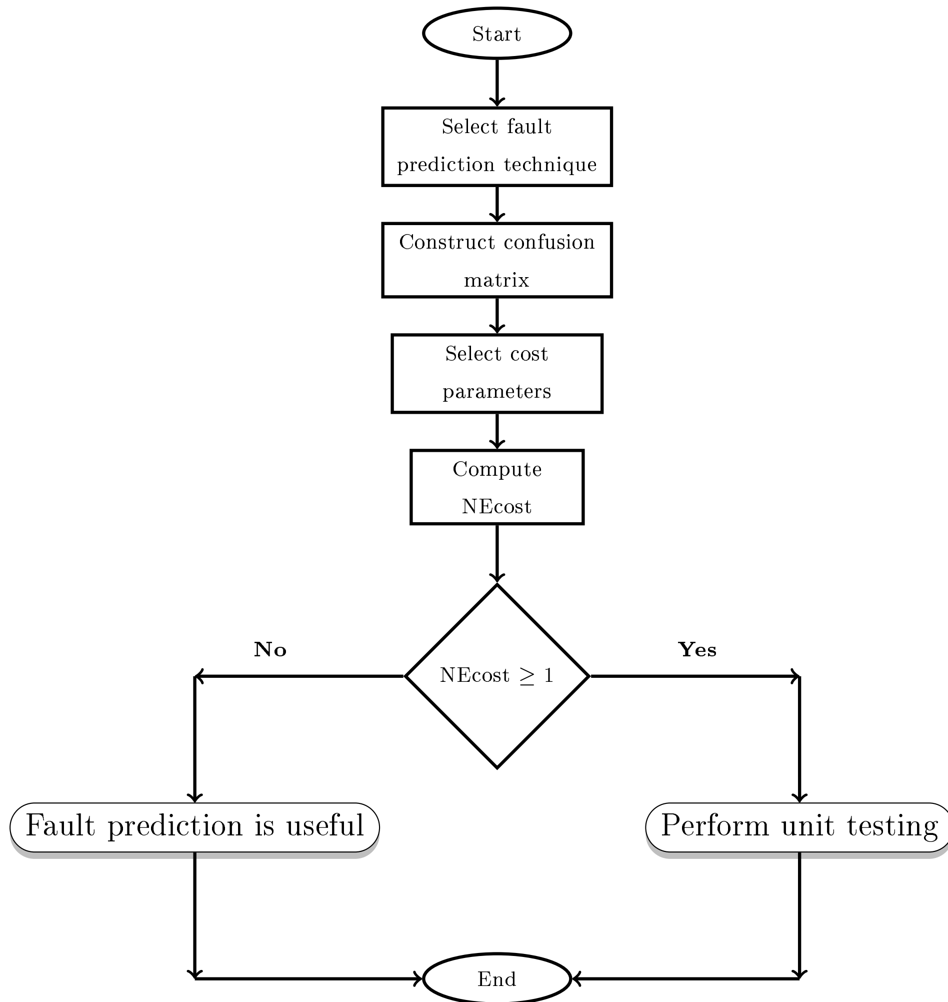

Figure 4.1: Cost-based evaluation framework for software fault classification

The proposed framework clearly states that if a technique accounts for having high false negative and/or high false positive rates, then it results in higher fault removal cost. When this approximated cost exceeds the unit testing cost (Tcost), it is better to test all the modules at unit level instead of using fault prediction technique.

4.3 Performance Evaluation Parameters

This sub-section gives the basic definitions of the performance parameters used in statistical and machine learning methods for fault prediction.

The performance parameters for statistical analysis can be determined based on the confusion matrix [110] as shown in Table 3.3. The confusion matrix contains information about actual and predicted classifications done by a fault-prediction technique.

The following are the performance measures used in classification.

i. False Positive Rate (FPR)

It is measured as the ratio of modules incorrectly classified as faulty class to the entire non-faulty classes.

$$FPR = \frac{FP}{TN + FP} \tag{4.10}$$

ii. False Negative Rate (FNR)

It is measured as the ratio of classes incorrectly classified as non-faulty class to the entire faulty classes.

$$FNR = \frac{FN}{TP + FN} \tag{4.11}$$

ii. True Positive Rate (TPR) or Recall

It is measured as the ratio of classes correctly classified as faulty to the entire faulty classes.

$$TPR = \frac{TP}{TP + FN} \tag{4.12}$$

iv. True Negative Rate (TNR) or Specificity

It measures the ratio of non-faulty modules which are correctly identified

$$TNR = \frac{TN}{TN + FP} \tag{4.13}$$

v. Precision

Precision is defined as the degree to which the repeated measurements under unchanged conditions show the same results [110].

$$Precision = \frac{TP}{FP + TP} \tag{4.14}$$

vi. Accuracy

Accuracy as defined by Yuan *et al.* [111] is the proportion of predicted fault prone modules that are inspected out of all modules.

$$Accuracy = \frac{TN + TP}{TN + FP + FN + TP} \tag{4.15}$$

4.4 Results and Analysis

In this section, the relationship between value of metrics and the fault found in a class is determined. In this approach, the comparative study involves using six CK metrics as input nodes and the output is the achieved fault classification rate for AIF version 1.6.

This section highlights on the design and use of neural network as a classifier and also presents the obtained cost-based analysis results for classification of faults obtained by applying Logistic regression, ANN, RBFN, FLANN and PNN techniques.

4.4.1 Neural network as a classifier

In the design of neural network as a classifier, the target output y' determines the type of classification result of a class as faulty or not faulty. The following conditions are taken into account to predict the accuracy of classification which are mentioned below for four neural network approaches as a

classifier:

$$Classifier = y' \implies \begin{cases} > 0, & Class\ is\ Faulty \\ \\ < 0, & Not\ Faulty \end{cases} \tag{4.16}$$

From Equation 4.16, it can be noticed that "if the target output of the neural network y' is greater than 'zero', then a class is classified as faulty, else it is classified as not-faulty class". Table 4.4 shows the confusion matrix constructed before any neural network model is used as a classifier. This table contains a total number of 965 classes, among which 777 classes have zero bugs and the remaining 188 classes have at least one bug.

Table 4.4: Confusion matrix

	Not-Faulty	Faulty
Not-Faulty	777	0
Faulty	188	0

4.4.1.1 Logistic classifier

The logistic regression method helps to indicate whether a class is faulty or not, but does not convey anything about the possible number of faults in the class.

Table 4.5: Confusion matrix for Logistic classifier

	Not-Faulty	Faulty
Not-Faulty	767	10
Faulty	172	16

In comparison with Table 4.4, after applying logistic regression as a classifier (Table 4.5), a total of 783 (767 + 16) classes are correctly classified as not-faulty with an accuracy of 81.13%.

4.4.1.2 ANN as a classifier

Table 4.6 and Table 4.7 show the classification matrix for Gradient Decent and Levenberg Marquardt learning techniques of ANN.

Table 4.6: Confusion matrix for Gradient Descent

	Not-Faulty	Faulty
Not-Faulty	761	16
Faulty	157	31

In comparison with Table 4.4, Gradient Descent ANN (Table 4.6) is able to classify a total of 792 (761+31) classes as not-faulty with a accuracy of 82.07%.

Table 4.7: Confusion matrix for Levenberg Marquardt

	Not-Faulty	Faulty
Not-Faulty	756	21
Faulty	171	17

In comparison with Table 4.4, LM model (Table 4.7) is able to classify a total of 773 (756+17) classes as not-faulty with an accuracy of 80.10%.

4.4.1.3 RBFN as a classifier

Three layered RBFN is considered, in which six CK metrics are taken as input nodes, nine hidden centers are taken as hidden nodes and output is the fault classification rate. Table 4.8 shows the classification matrix when Basic RBFN is used as a classifier.

From Table 4.4 it is clear that before applying Basic RBFN as classifier, a total number of 777 classes contained zero bugs and 188 classes contained at least one bug. After applying Basic RBFN classifier (Table 4.8), a total of 607 (517 + 90) classes are correctly classified as not-faulty with an accuracy of 62.9%.

Table 4.8: Confusion matrix for Basic RBFN

	Not-Faulty	Faulty
Not-Faulty	517	260
Faulty	98	90

a. Gradient Descent learning method

Equation 3.21 and 3.22 are used for updating center and weight during training phase. After simplifying Equation 3.21, the equation is represented as:

$$C_{ij}(k+1) = C_{ij}(k) - \eta_1(y' - y)Wi\frac{\phi_i}{\sigma^2}(x_j - C_{ij}(k)) \tag{4.17}$$

and the modified Equation 3.22 can be framed as:

$$W_i(k+1) = W_i(k) + \eta_2(y' - y)\phi_i \tag{4.18}$$

where σ is the width of the center and k is the current iteration number.

Table 4.9 shows the classification matrix for Gradient RBFN. Gradient RBFN classified totally 898 classes (776 + 122) as not-faulty with an accuracy rate of 93.05 %.

Table 4.9: Confusion matrix for Gradient RBFN

	Not-Faulty	Faulty
Not-Faulty	776	1
Faulty	66	122

b. Hybrid learning method

In Hybrid learning method, centers are updated using Equation 3.23, while weights are updated using supervised learning methods. In this work, Least Mean Square Error (LMSE) algorithm is used for updating the weights.

Table 4.10 shows the classification matrix for Hybrid RBFN. Hybrid RBFN classified totally 767 classes (747 + 20) as not-faulty with an accuracy of 79.4%.

Table 4.10: Confusion matrix for Hybrid RBFN

	Not-Faulty	Faulty
Not-Faulty	747	30
Faulty	168	20

4.4.1.4 FLANN as a classifier

Table 4.11 shows the obtained classification matrix when FLANN technique is used as a classifier.

Table 4.11: Confusion matrix for FLANN

	Not-Faulty	Faulty
Not-Faulty	742	35
Faulty	160	28

After applying FLANN classifier (Table 4.11), a total of 770 (742+28) classes are correctly classified as not-faulty with an accuracy of 79.79%.

4.4.1.5 PNN as a classifier

Table 4.12 shows the classification matrix obtained by applying PNN technique as a classifier.

Table 4.12: Confusion matrix for PNN

	Not-Faulty	Faulty
Not-Faulty	775	2
Faulty	181	7

In comparison with Table 4.4, it is observed that, after applying PNN as a classifier (Table 4.12), a total of 782 (775 + 7) classes are correctly classified as not-faulty with an accuracy of 81.03%.

The fault removal cost for AIF version 1.6 obtained by applying Logistic regression, ANN, RBFN, FLANN and PNN techniques are tabulated in Table 4.13. In this cost-based evaluation framework, the Estimated fault removal cost of the software without using fault prediction technique 'Tcost' is found to be 3546.8 staff hours per defect.

Table 4.13: Fault removal cost for AIF 1.6 using various classifier models

Classification model	Precision	TP Rate	FP Rate	TN Rate	FN Rate	Accuracy	Ecost	NEcost
Logistic regression	61.54	08.51	01.29	98.71	91.49	81.13	3119.4	0.8795
Gradient Descent	65.96	16.49	02.06	97.94	83.51	82.07	3109.7	0.8762
Levenberg Marquardt	44.74	09.04	02.70	97.30	90.96	80.10	3145.3	0.8868
RBFN Basic	25.71	47.87	33.46	66.54	52.13	62.90	3622.3	1.0213
RBFN Gradient	99.19	64.89	00.13	99.87	35.11	93.50	2922.0	0.8238
RBFN Hybrid	40.00	10.64	03.86	96.14	89.36	79.40	3162.8	0.8917
FLANN	44.44	14.89	04.50	95.50	85.11	79.79	3162.1	0.8915
PNN	77.78	03.72	00.26	99.74	96.28	81.03	3114.3	0.8780

4.4.1.6 Comparison of cost analysis

Data set of AIF version 1.6 from PROMISE repository is chosen to evaluate the impact of fault prediction technique. The fault removal cost (NEcost) computed using the proposed framework is used to evaluate the models.

To illustrate effectiveness of the proposed cost-based evaluation framework, classifier models such as Logistic regression, ANN, RBFN, FLANN and PNN are used for computing misclassification cost. The goal is to demonstrate the cost evaluation framework and suggest whether performing fault prediction using particular prediction model is useful or not rather than identifying the "best" fault-prediction model.

Table 4.13 shows the various parameters related to cost evaluation framework along with NEcost. NEcost is the criterion used in evaluating a classification model to show the usefulness of fault prediction.

From Table 4.13, it can be observed that:

i. The Gradient Descent approach of RBFN classifier obtained the best classification rate of 93.50% when compared with other four models, and

ii. Gradient Descent RBFN incurs less cost involved in testing (with a NEcost ratio of 0.8238).

This indicates that performing fault prediction on the basis of classification cost involved using Hyrbid RBFN method is very much effective in comparison with LR, GD, LM, FLANN and PNN models.

It is observed that, Normalized fault removal cost (NEcost), which is the ratio of Ecost and Tcost (Equation 4.9) determines the fault prediction model's effectiveness in a succinct manner.

The proposed cost-based evaluation framework provides:

1. A binary yes or no scale whether to perform fault prediction analysis or not.

2. A criterion to choose a better fault prediction model based not only on the obtained accuracy rate but also taking NEcost into consideration.

4.5 Summary

In this chapter, an attempt has been made to design a cost based evaluation framework for finding the efficiency of the developed fault prediction model using different neural network models as classifiers. Models such as LR, ANN, RBFN FLANN and PNN were used as classifiers. NEcost of the proposed cost based analysis framework was used as a criterion to evaluate the effectiveness of the respective models. From the obtained results, it is noticed that the Gradient Descent approach of RBFN obtained better results in terms of fault removal cost when compared with LR, GD, LM, Basic and Hyrbid RBFN, FLANN and PNN classifier models.

Chapter 5

Test Data Generation for Object-Oriented Program using Meta-heuristic Search Algorithms

---

An automated approach for test data generation is very crucial as manual process is laborious and time consuming. It is observed that, test data generation for object-oriented programs is difficult because of inheritance, polymorphism, and data abstraction.

Here, Control flow graph for Object-Oriented program is automatically generated, referred to as Extended Control Flow Graph (ECFG). From the generated ECFG, test data are generated for a selected target path. Here test data are generated for the case study i.e., Bank ATM withdrawal task [15] using three meta-heuristic algorithms viz., Clonal selection algorithm, Binary particle swarm optimization, and Artificial bee colony. Experimental results show that Artificial bee colony algorithm generates suitable test data.

5.1 Introduction

Test data generation in program testing, is the process of identifying a set of test data which satisfies the given testing criterion [123]. A test data generator is a tool which helps a tester in generating test data for a given program. Most of the test data generating activities are based on the aspects of either path wise generation [124, 125, 126] or data specific generation [127, 128, 129] or random test data generation [130]. However, these techniques require complex algebraic computations. Hence, artificial intelligence based approaches can be applied for reducing testing efforts.

In this chapter, three meta-heuristic search algorithms viz., Clonal selection algorithm, Binary particle swarm optimization and Artificial bee colony optimization are used for generating test data for Object-Oriented program. All the paths selected in a module need to be executed (at least once), and thus generating a large set of test data (manually) for these set of paths is quite a difficult task. Hence, certain degree of automation may be thought of as an alternative to minimize the use of testing resources. The meta-heuristic search algorithms help in achieving this goal by generating test data required to cover the paths in a module, in an near-optimal way.

5.2 Meta-heuristic Search Algorithms

Meta-heuristic search techniques are high-level frameworks, which utilize heuristics in order to find solutions for combinatorial problems at a reasonable computational cost [56]. Optimization techniques like GA, PSO, CSA, ABC, Simulated annealing (SA), Tabu search (TS) and Ant colony optimization (ACO) algorithms are search based meta-heuristic algorithms applied to problems, where there is difficulty in obtaining the requisite optimal solution within a time bound and involving less space complexity.

Meta-heuristic algorithms have the ability to obtain optimal solution in a very large search space of candidate solutions. These algorithms do not make any assumptions about the problem being optimized. When compared to other techniques, these algorithms can be applied irrespective of whether the problem to be optimized is continuous or discontinuous in nature.

5.2.1 Role of meta-heuristic search based algorithms in software testing

Test data generation is an optimization problem, that satisfies a given test requirement. Search based test data generation helps in such a scenario, and heuristic is applied to solve the problem of test data generation.

Test data can be generated by using either functional or structural testing techniques. Functional testing involves matching the exact functionalities as outlined during the requirements phase. Structural testing involves using techniques such as decision coverage, branch coverage, and path coverage to examine the execution of a software system. In this work, basis path coverage has been chosen for test data generation as it emphasizes the execution of all the feasible paths in the control flow graph. Test data are generated in a manner to cover the selected feasible paths from the control flow graph for a particular program. Hence, path testing can be referred to simply as a constraint-satisfying approach. A target path is executed by covering all the predicate nodes included in that path by using the generated test data.

The process of generating test data to satisfy a given criterion from the large input space of a program can be formulated as a search-based application of an algorithm for test data generation. Hence, the application of meta-heuristic algorithms in test data generation is helpful for obtaining effective solutions.

5.2.2 Need for automated test data generation

The process of automatic generation of test data plays a major role in software testing. In testing, emphasis is given on finding specific input test data that satisfies the given test criterion. In literature it is observed that, common methods such as notion to perform and random methods for test data generation [131]. These two methods are not suitable for large and complex programs because of their complex algebraic computations, and also is time consuming.

Also, manual generation of test data is time consuming, laborious, not-exhaustive and prone to errors. This is due to the presence of large number of predicate nodes in the module. This would lead towards NP-hard problem [132]. In reality, only human effort is not sufficient to generate a suitable amount of test data, which would test the software successfully. The approach that can minimize human effort is by automating the test data generation process. Unfortunately, all the available automated test data generation techniques are rarely used in generating test data because of their inefficiency in achieving adequate coverage by the generated test data. Therefore, some intelligent search based algorithms may be applied to generate optimal test data.

5.3 Extended Control Flow Graph (ECFG)

Procedural oriented testing based on basis path testing, known as McCabe's basis path testing approach has been implemented over the years. However with the increase in need for building hierarchical models, the focus has shifted towards Object-Oriented methodology. The three important concepts of the Object-Oriented methodology i.e., encapsulation, inheritance and polymorphism make the approach of applying procedural testing methods to Object-Oriented methodology a difficult task.

One of the commonly used examples of white-box testing technique is "basis path testing", which ensures that every path of a program needs to be executed

at least once. McCabe cyclomatic complexity metric determines the number of linearly independent paths in a control flow graph (CFG) [133]. The cyclomatic complexity determines the minimum number of test cases required to execute the conditional statements at least once [134].

The CFG constructed for traditional programs is extended for Object-Oriented methodology. The ECFG is a collection of CFG's in a layered manner. The ECFG is a directed graph, where nodes refer to methods rather than statements.

5.3.1 ECFG features

ECFG is a layered graphical model which represents a collection of CFG's of the individual methods of the class or module. ECFG can be formally defined as directed graph - G(V,E), where 'V' refers to methods rather than statements and 'E' represents an edge between the methods.

Essentially, the ECFG is composed of two layers:

i. The first layer represents the methods of individual classes.

ii. The next layer (embedded in the topmost layer) represents the CFG of these methods.

The following are the salient features of ECFG [133]:

i. ECFG is much similar to traditional CFG. It consists of nodes, and edges between a pair of nodes, except that some nodes in the top level may be disconnected.

ii. Nodes in CFG refer to statement(s), whereas in ECFG nodes refer to methods.

iii. Every method has associated graphs (CFG) and their respective cyclomatic complexity (CC) value. Method which is not found in the required class may be a part of its parent class.

iv. Object declaration is similar to variable declaration of procedural language but is not in a sequence (as in CFG), since it refers to constructor method.

v. Edges between nodes are formed, whenever a method calls another method.

5.3.2 Cyclomatic complexity computation for ECFG

The ECFG for Object-Oriented methodology may possibly be connected in six different ways. Figure 5.1 gives a graphical view of how the methods (say m1, m2,...m7) are connected in a main program [133].

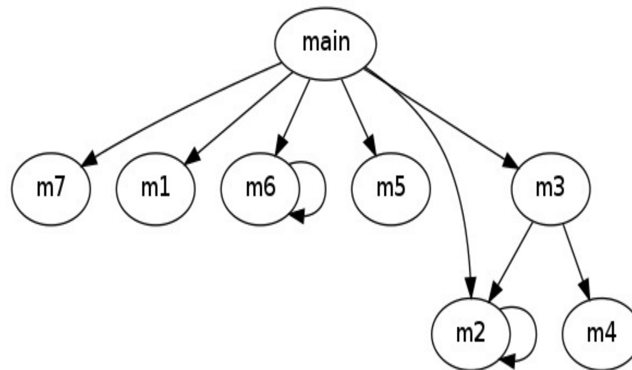

Figure 5.1: Basic ECFG

Case 1: An example is considered; where in two or more graphs are connected in series i.e., methods are executed as a sequence one after another. Extended Cyclomatic Complexity (E-CC) value in this scenario can be computed as: E-CC = max(m1, m2), where m1 and m2 are two methods in series with individual complexity values V(G1) and V(G2) respectively.

$$E - CC = V(Gx) \;\; if \;\; V(Gx) > V(G1), V(G2).....V(Gn) \;\; and \;\; 1 < x < n$$

Example: Let m1 and m2 be two methods in series. Figure 5.2(A), Figure 5.2(B) and Figure 5.2(C) refer to CFGs and ECFG

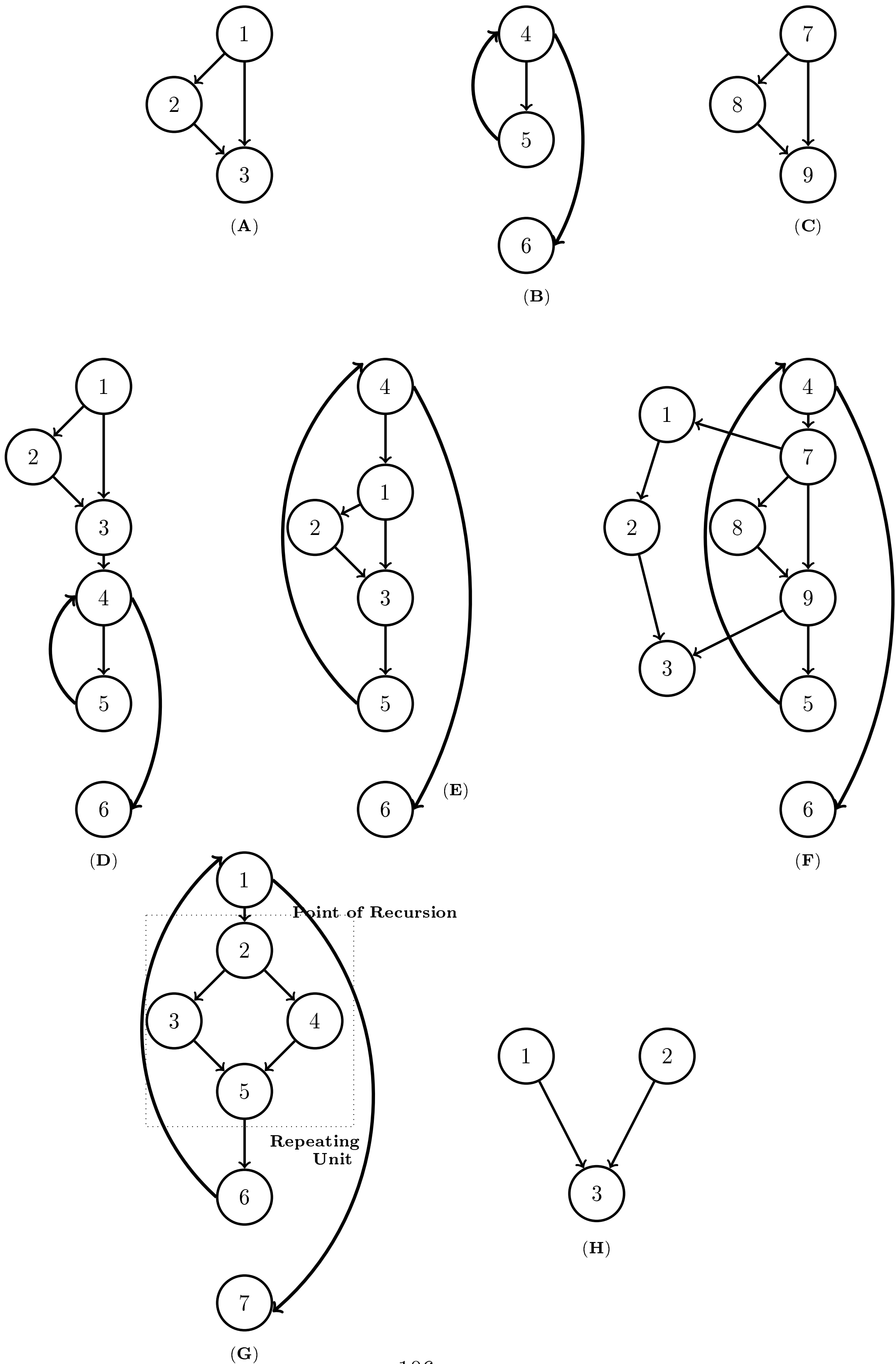

Figure 5.2: Methods association in ECFG

```
m1(int a, int b)
{
1 if(a>b)  -> V(m1) =2
2     printf("A is greater");
3 }

  m2(int a)
  {
  4 while(a!=10)
  5 {printf("a= %d",a);
    a++;}
  6 }
```

Referring to Figure 5.2(C), E-CC = max(2,2) = 2

Case 2: In a large and complex graph having two or more graphs embedded within a graph i.e., when a method calls another method, which in turn calls another and so on. Then E-CC value is equal to sum of V(G1), V(G2), ...V(Gn)-(n-1); where n-1 graphs (2 to n) are embedded within G1. Here V(G1), V(G2),...V(Gn) are the individual complexities of each graph (and in this case G1 embeds G2, G2 embeds G3 and so on).

$$E - CC = V(G1) + V(G2) + V(G3) + ...V(Gn)(n - 1)$$

Example: Consider the same two methods m1 and m2, connected as m2 embedding m1. The method m2 is slightly different with one statement having a call to m1 as given below. The individual CFGs are same as Figure 5.2(A) and Figure 5.2(B). The ECFG is shown in Figure 5.2(D).

```
m2(int a)
{
4 while (a!= 10)
5 { m1(2,5);
 printf("a = %d, a); -> V(m1) = 2
```

```
a++;} -> V(m2) = 2
6 }
```

m2 embeds m1 in such a way that the path 4-5 in m2 is replaced by the paths in m1 as shown in Figure 5.2(D).

E-CC = 2 + 2 - 1 =3.

Case 3: If a graph embeds another graph more than once, for example if G1 embeds G2 thrice, then G2 is taken to be embedded only once. Composite complexity E-CC value will be the same as in Case 2, where all the complexity values refer to unique graphs - G1, G2,......Gn.

Case 4: If in case, many graphs are embedded in the same graph i.e., more than one method calls the same method, then the repeated graph is considered only once but the point of entry should be tested in every context. So the value of E-CC = (V(G1) + V(G2)-1) + V(G3) +...+ V(Gn) + (n-2), where V(G1): complexity of G(1) that is embedded multiple times, V(G2)...V(Gn) are complexity values for graphs which embed G1, and n-2: number of graphs embedding G1 except one.

$$E - CC = (V(G1) + V(G2) - 1) + V(G3) + ... + V(Gn) + (n - 2)$$

Example : Let us consider a method m3, connected as m3 called from m1 and m2.

```
m1(int a, int b)
{
1 if (a>b)
2 printf("A is greater");
  m3(10);
3 }
```

```
  m2(int a)
  {
4 while (a!= 10)
5 { m3(2);
  printf("a = %d",a);
  a++;}
6 }


  m3(int a)
  {
7 if (a = 10)
8 printf("ok");
9 }
```

The CFGs of m1 and m2 are shown in Figure 5.2(A) and Figure 5.2(B) earlier. The CFG of m3 and the ECFG is shown as Figure 5.2(E) and Figure 5.2(F) respectively. V(m1) = 2, V(m2) = 2 and V(m3) = 2. As per ECFG, E-CC = (CC(M1) + CC(M3)- 1) + CC(M2) + 1 = (2 + 2 - 1) + 2 + 1 = 6.

m1 and m2 embeds m3 in such a way that the path 1-3 in m1 and 4-5 in m2 is replaced by the paths in m3 as in Figure 5.2(G).

Case 5: Suppose a graph is recursively repeated i.e., a method is recursively called, the value of E-CC = V(G1) + 2, where V(G1) is the complexity of the method in recursion.

Example: Consider the following code and its ECFG in Figure 5.2H.

```
recur(int i)
1 {
2 if (i/2 == 0)
3 { printf("ok");}
4 else
  { printf("Not ok");
5 }
```

```
6 if (i !=0)
    recur (--i);
  }
7 }
```

E-CC = V(recur) + 2

= 2 + 2 = 4

Case 6: If a graph involves recursive call of more than one method, then this situation is a combination of Case 1, Case 2 and Case 3. Composite complexity value is calculated as:

$E - CC = V(G') + 2$, where G′ is more than one method connected as per case 1 or 2.

5.4 Fitness Function based on Korel's Branch Distance Function

This section gives the design criterion of a fitness function for a real-life application based on Korel's branch distance function.

Fitness function is an essential parameter to solve any optimization problem. A well defined fitness function (objective function) not only consumes less time and few resources to optimize a problem but also increases the affinity in searching a solution.

Meta-heuristic optimization techniques require a numerical formulation of the test goal, from which a 'fitness function' is formed. The purpose of the fitness function is to guide the search into promising, unevaluated areas of a potentially vast input domain, in order to find required test data.

Fitness function for test data generation in case of amount withdrawal in an ATM of a Bank, is derived based on Bogdan Korel's branch function [123]. A path P is considered in the program execution. The goal of the test data

generation problem is to find a program input x on which P will be traversed without loss of generality. Korel assumed that the branch predicates are simple relational expressions (inequalities and equalities). That is, all branch predicates are of the form: E_1 op E_2, where E_1 and E_2 are the arithmetic expressions and 'op' is one of the $\{<, \leq, >, \geq, =, \neq\}$ operators.

In addition, Korel assumed that predicates do not contain AND or OR or any other Boolean operators. Each branch predicates E_1 op E_2 can be transformed to the equivalent predicate of the form: F rel 0, where F and *rel* are given in Table 5.1.

Table 5.1: Equivalent predicate of branch function

Branch Predicate	**Branch Function F**	**rel**
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	$\text{abs}(E_1 - E_2)$	$=$
$E_1 \neq E_2$	$\text{abs}(E_1 - E_2)$	$\leq$

F is a real valued function, referred to as branch function [123], which is:

i. Positive (or zero if rel is $<$); when a branch predicate is false or

ii. Negative (or zero if rel is $=$ or $\leq$); when the branch predicate is true.

It's obvious that F is actually a function of program input x. Symbolic evaluation can be used to find explicit representation of the $F(x)$ values in terms of input variable. However, this process requires a very large and complex algebraic manipulation. For this reason, an alternative approach is used in which the branch function is evaluated. For example, to test a conditional statement "if $a > b$ then" as a branch function F, the F(x) value can be computed for a given input program by evaluating '$a - b$' expression.

5.4.1 Case study: Bank Automatic Teller Machine (ATM)

Figure 5.3 shows the sequence of operations performed in ATM withdrawal task by the customer.

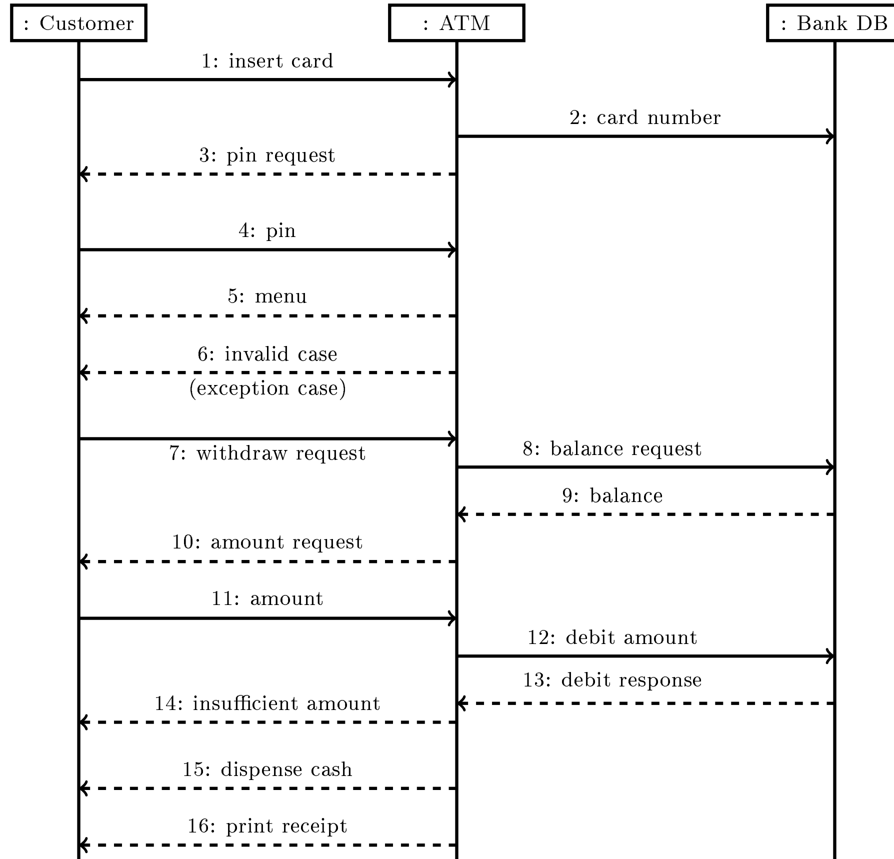

Figure 5.3: Sequence diagram for ATM withdrawal task

Functions associated with withdrawal task in a Bank ATM [15] are tested on the basis of different techniques in order to generate its test data for a single feasible path in ECFG. A feasible path is referred as to a path/code which can be traversed/reached by the generated input (test data). The formal definition of feasible path is as follows: “A path is feasible, if there exists some input that will cause the path to be traversed during execution" [135].

The fitness function is developed on the basis of traversal of predicate nodes. For instance, as shown in Figure 5.4 when the node ‘1’ is visited, the condition

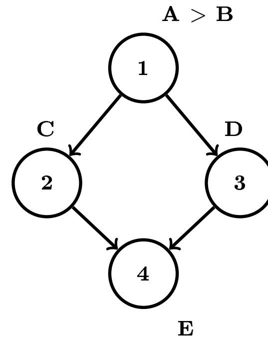

Figure 5.4: Basic CFG

of the predicate node may either be $A > B$ or $B > A$ or even $A = B$. So taking equality condition into consideration, $A = B \Rightarrow A - B = 0$; as test data generation being a minimization problem, the fitness function f is given as: $1/(A - B)$. However, this functional value tends to infinity when $A - B = 0$, so to avoid this sort of numeric overflow, a small delta value ($\delta = 0.05$) has been added to the fitness function. Hence the fitness function in general is given as:

$$f = 1/((abs(A - B) + 0.05)^2) \tag{5.1}$$

where A, and B represent the desired and actual branch predicate values respectively.

Test data are generated for a small segment of code with some realistic constraints for Bank ATM withdrawal such as:

i. The amount the customer enters for withdrawal should be a valid input.

ii. The amount the customer enters for withdrawal should be in multiples of 100.

iii. The amount the customer enters for withdrawal should be less than the net available balance.

iv. An account holder must possess a minimum balance of \$100 after the completion of withdrawal task.

5.5 Test Data Generation for Bank ATM

The various steps for constructing ECFG are followed and then three meta-heuristic algorithms are applied for test data generation.

The steps followed are:

1. A process is being implemented to parse Java class and construct CFG for a given method.

2. CFG is traversed to find CallNodes, to connect edges between nodes of ECFG (in ECFG, methods are represented as nodes). E-CC value is computed for different methods based on the six possible ways as mentioned in Section 5.3.2.

3. A target path is selected, and affinity function derived based on the branch distance is applied to generate test data using three meta-heuristic search algorithms.

The following steps indicate the procedure followed to generate test data using ECFG:

- ECFG is a layered approach, wherein modules are 'overlapped' on each other. The method to be tested is determined based on the Extended Cyclomatic Complexity (E-CC) Value.

- In ECFG, a node refers to methods rather than statements. Hence after selecting predicate nodes, the methods to be tested are identified. Then meta-heuristic search techniques are applied on the predicate nodes to generate test data.

  In this case study, a feasible path of 'withdrawal task' in case of Bank ATM is considered. Here, feasible path is referred to a path/code which can be traversed/reached by the generated input.

The following subsections give a brief note on how a CFG for individual method is constructed. Subsequently, individual CFG's are used in constructing ECFG. On selecting a target path, test data are generated based on the affinity function using CSA, BPSO and ABC algorithms.

5.5.1 Construction of CFG for an individual method

Compiled classes are used for the analysis purpose due to the following reasons:

- If source code is used, there is a need for parsing. Error checking is not necessary for compiled classes.
- For source file, there can be various out going edges for a node, for example: if (a<b && a>0). But for class files there will be at least two out going edges because class file contains instruction like machine language, where a node either evaluates to a Boolean value 'true' or 'false'.

For construction of CFG for a method, ASM, a byte-code engineering library for JAVA is used to work on compiled JAVA classes. Let mainCFG be the main method's CFG that can be constructed by supplying the class file to new MethodFlowGraph object.

The ASM tree API reads a class file into hierarchically arranged objects i.e., ClassNode containing a list of MethodNode for methods of a class and each MethodNode containing list of InstructionNode. By using Analyzer class from tree API, control flow edges between instructions are determined. So, an Analyzer stores edges into MethodFlowGraph object.

A MethodFlowGraph is the actual implementation of CFG for a method. It contains an array of nodes corresponding to each instruction and an adjacency matrix for the graph.

5.5.2 Construction of ECFG

Construction of ECFG starts from CFG of the main method. The CFG for invoked methods are added recursively. Algorithm 2 gives the steps for construction of ECFG, by passing mainCFG.

Algorithm 2 : ECFG Construction

```
addGraphsRecursively(MethodFlowGraph: x)
for each node i in x do
  if (IiscallNode) then
    find the class of called method ;
    search for classnode for calledmethods class in the list
    if (notfound) then
      create new Classnode for calledmethods class and put in the list;
    else if (c = Called methods classnode from list) then
      search for method flow graph in method flow graph's list;
      if (notfound) then
        m = create new method flow graph for the called method;
        add m to method flow graph list and nodes of ECFG;
        add edge from x to m;
      else
        m = method flow graph from the list;
        add edge from x to m ;
      end if
    end if
  end if
end for
```

The algorithm executes as follows:

i. Algorithm traverses the passed method flow graph sequentially, instruction by instruction in search of call nodes.

ii. When a call node is found, it finds the class of the called method invoked, if it is found in the 'list' the class node is taken from the list, otherwise new class node object is formed and inserted in to the list.

iii. “methodsCFG" is searched in the list, if it is previously created then it will be found in the list. If found in the list, an edge is added from called graph to the graph from the list, otherwise new method flow graph is made and inserted in to the list.

5.5.3 Test data generation using meta-heuristic search algorithms

The following sections give a brief description of the algorithmic approach followed for generating test data using CSA, BPSO and ABC algorithms.

5.5.3.1 Clonal selection algorithm

Clonal selection algorithm (CSA) was proposed by Castro and Zuben [12], and is further modified to suit our objective. Algorithm 3 shows the approach followed to generate test data for the selected target path (derived from ECFG) using CSA. Affinity function in CSA is derived based on the concept proposed by Korel [123] as mentioned in Section 5.4. The affinity function utilized in test data generation for Object-Oriented methodology is as follows:

$$f = 1/(abs((net_bal - (wd_amt - min_bal)) + 0.05)) \tag{5.2}$$

CSA execution for generating test data is as follows:

The initial execution begins by randomly generating test data (random population). This initial test data are evaluated based on the affinity function (in CSA, test data are referred to as antibodies). After the evaluation, a target path is selected. On selecting a target path, the initial test data (antibodies) are applied to CSA in order to find the suitable test data that covers the selected target path. Next, the step of cloning is applied. In cloning, two clones of each antibody are generated based on Equation 5.3.

$$q_i = Int(\frac{\beta N}{i}) \tag{5.3}$$

Algorithm 3 Test data generation using CSA

```
Initialize the number of generations (g) = 0;
Initialize the initial population randomly A_o
Evaluate affinity function:
    F = 1/(abs((net_bal − (wd_amt − min_bal)) + 0.05))
if  g < 1000 then
   Print results (test data);
   Exit
else
   Clone A_n to be A_n';
   Hyper-mutate A_n'to A_n'';
   Evaluate and select A_n'';
   Destroy and renew to construct a new population A_n;
   g++;
end if
Evaluate affinity function.
```

where, β is the multiplying factor, and $Int()$ is the function that rounds its argument towards the closest integer.

The clone process follows the rule of superior win. Higher the affinity values of the antibody, the more the antibody gets cloned. Next, the clones are hyper-mutated. Hyper-mutation is a step where the mutation process is executed by eliminating the cross-over operation. In hyper-mutation, every clone is mutated either with a fixed number of bits or with a fixed probability of mutation. After hyper-mutation, 10% of the antibodies which have low affinity value are replaced by newly created random antibodies (N_r = N/2; N_s = $N_r/10$, where N_r - Renewed antibodies and N_s - Worst antibodies). This process of replacing and creating new antibodies helps in achieving diversified results. This entire process continues until the stopping criterion is met.

In CSA, antibodies refer to the input parameter, i.e., the initially generated test data, and the test data satisfying the requirement (suitable test data executing the selected target path) is referred to as Antigen.

5.5.3.2 Binary particle swarm optimization

The following steps outline the format for application of BPSO for test data generation, assuming a binary string representation. Let 'P' be the path (target path) in the program for which test data is to be generated.

Step - 1: Gen = 1000.

Step - 2: Meta-heuristic search is setup.

a. Binary string representation for the randomly generated test data (for the target path) is selected.

b. Particles are evaluated based on the fitness function.

c. The lines of code in the selected target path are instrumented, bit representation is changed using Equation 5.4.

$$P(x_{id}(t) = 1) = f(x_{id}(t), v_{id}(t-1), p_{id}, p_{gd}) \tag{5.4}$$

d. Suitable particles for further search of test data are selected.

Step - 3: Test data are generated.

a. BPSO search is invoked using P_t (Equation 5.4) for fitness computation.

b. Suitable test data are selected based on better fitness value.

c. The test data are regenerated if necessary.

Test data generation begins by representing random input test data into string and then identifying the suitable fitness function. The raw fitness is evaluated using the branch distance function [123]. String fitness is developed as shown in Equation 5.1 (Sub-section-5.4).

When the predicate node is covered by the generated test data, the node is marked as traversed and search is re-run. After all the nodes are traversed, the algorithm selects suitable test data satisfying the given criterion based on fitness value.

5.5.3.3 Artificial bee colony

Algorithm 4 gives the flow for test data generation using ABC, and Table 5.3 gives us the tabulation of various experimental setup (parameters used) for automatic generation of test data using ABC algorithm.

Algorithm 4 Test data generation using ABC

Initialize random population (X_{ij})
Gen = 0
while Gen < 1000 **do**
  Evaluate the fitness value of each bee based on the fitness function (f)
  $f = 1/((\text{abs}(suc_bal(i) - min_bal) + 0.05)^2)$
  Use elitism as the selection operator to select the individuals to enter into the mating pool
  New solutions Y_{ij} in the neighborhood of X_{ij} for the employed bees are produced using the following equation

$$y_{ij} = x_{ij} + \phi_{ij} * (x_{ij} - x_{kj}) \tag{5.5}$$

  where ϕ_{ij} is a random number between 0 and 1 and X_{kj} is a randomly selected solution

  Greedy selection process is applied between Y_{ij} and X_{ij}
  Probability values p_i for the solutions X_i by means of their fitness value are computed using the following equation

$$p_i = \frac{\text{fitness}_i}{\sum_{j=1}^{n} \text{fitness}_j} \tag{5.6}$$

  Produce new solutions from the solutions X_{ij} depending on the probability p_i and evaluate them
  Apply greedy process between new and old test data
  Select bee's with low fitness value, if exists, replace it with a new randomly produced solution
  Memorize the best food source position (test data) achieved so far
  Gen = Gen + 1
end while
Select the colony having the best fitness value as the desired result (effective test data for target path)

Artificial bee colony (ABC) is a swarm-intelligence-based optimization approach. The behavior of the ABC optimization algorithm is employed to generate test data, by finding the global optima in a multidimensional search space. This algorithm combines both local and global search methods to attain global or near-global optima.

The execution of the ABC algorithm is as follows:

1. First, the number of generations are initialized and the fitness value of each bee is computed using the fitness function.

2. Elitism approach is applied to choose the most fit individual among the population. A new population is produced using Equation 5.5.

3. On obtaining the new population, individuals from both the new and the old population are selected based on the greedy selection process. Then the probability value of the newly found solution is computed using Equation 5.6.

4. The greedy selection process is applied again to choose the best fit. The individuals with lower fitness values are discarded and a new population is generated if necessary, otherwise the effective test data is memorized as the solution to the problem. The execution of the algorithm terminates when the stopping criterion is met.

5.6 Results

The following subsections highlight on the results achieved and the interpretations drawn by applying the experimental settings used in CSA, BPSO, and ABC algorithms.

5.6.1 Case study: Bank ATM

ECFG Output

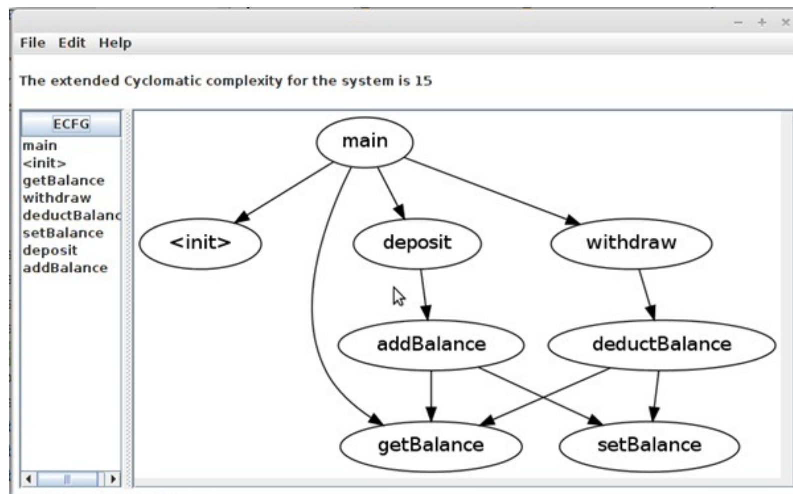

Figure 5.5: ECFG for Bank ATM

Figure 5.5 shows the generated ECFG for Bank ATM case study [15] using Algorithm 2. Figure 5.6 shows the ECFG for ATM withdrawal task and also the E-CC value is found to be 3.

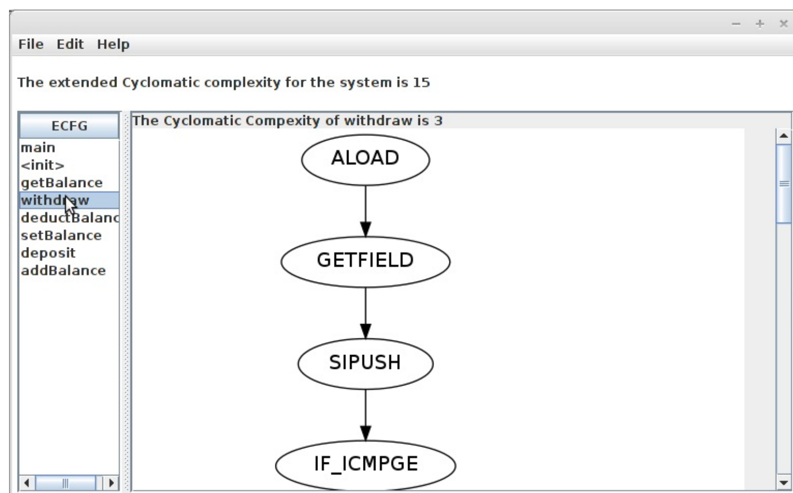

Figure 5.6: ECFG for Bank ATM withdraw task

5.6.2 Experimental settings

Table 5.2 and Table 5.3 show the various parameters and their respective values used in test data generation using CSA, BPSO, and ABC algorithms.

Table 5.2: Parameters used in CSA for test data generation.

Parameter	**Value**
Affinity function	f $= 1/(abs((net_bal - (wd_amt - min_bal)) + 0.05))$
Coding format	Binary
Antigen length	8 bits
Population size	100
Selection	Elitism method
Probability of mutation	0.15
Destroy and renew	N_r = N/2; N_s = Nr/10
Stopping criterion	1000 generations

Table 5.3: Parameters used in BPSO, and ABC for test data generation.

Technique	**Parameters**
BPSO	ϕ_1, ϕ_2: Random numbers from the uniform distribution (0,4) such that $\phi_1 + \phi_2 \leq 4$. w = [0.5+(rnd/2.0)], rnd is random number drawn from uniform distribution (0,1).
ABC	Producing new solutions Y_{ij} from X_{ij} using the Equation 5.5. Probability (p_i) for solutions X_i based on their fitness using Equation 5.6.

5.6.3 Interpretation of results

The approach followed for test data generation for path testing using meta-heuristic search algorithms, consists of four basic steps viz., control flow graph construction, target path selection, test data generation and execution. This subsection highlights on evaluation of the achieved test results.

This section highlights the results achieved using the CSA, BPSO, and ABC optimization algorithms with the experimental settings listed in Table 5.2 and Table 5.3. Test data are generated for a single feasible path in the ECFG for an ATM withdrawal task. The implementation part is hand-coded in MATLAB. A single target path is selected (a bank ATM withdrawal task), since evolutionary meta-heuristic algorithms act on a single characteristic during their execution.

To ensure the execution efficiency of the applied algorithm, the generated output (suitable test data) is divided into three classes of search space. This search space is categorized based on their affinity values.

Table 5.4 shows the affinity value range of test data and the classification of individual chromosome/particle/bee into their respective classes based on affinity value in terms of percentage.

Table 5.4: Percentage of class of test data having maximum fitness values in CSA, BPSO, and ABC respectively.

Fitness value range	**% of test data**		
f(x)(Search space)	**CSA**	**BPSO**	**ABC**
$0 \leq f(x) < 0.3$	13	48	24
$0.3 \leq f(x) < 0.7$	17	05	02
$0.7 \leq f(x) < 1.0$	70	47	74

Table 5.4 shows that more number of chromosomes having higher affinity value ‘f(x)’, lie in the range between $0.7 \leq f(x) < 1.0$. These affinity values are an indication of optimal test data obtained. The table also shows that the highest percentage of test data with the maximum fitness values is achieved by

the ABC algorithm. Out of the 100 individuals in the population, the test data with the maximum fitness values found in the search space $0.7 \leq f(x) < 1.0$ are chosen to check the execution of decision nodes during the testing phase. This also helps a tester to choose suitable solution (test data) based on the affinity values out of the total population size (N) i.e., tester can choose test data from the categorized search space (based on affinity value).

Table 5.5 shows the generated test data for the selected test case (Bank ATM withdrawal task) along with their affinity values for CSA, BPSO and ABC algorithms. From Table 5.5, it can be noticed that ABC generated better test data as it has high affinity values of its population.

Table 5.5: Affinity and the respective test data generated for meta-heuristic techniques

CSA Affinity Values	CSA Test Data	BPSO Affinity Values	BPSO Test Data	ABC Affinity Values	ABC Test Data
0.027	2300	1.41E-05	1400	1.77E-06	1500
0.0294	1900	1.77E-05	1700	1.78E-06	2600
0.0312	2200	2.08E-05	2100	1.89E-06	4500
0.0322	2300	4.85E-05	14000	1.92E-06	2000
0.0333	1300	5.06E-05	1200	1.94E-06	1700
0.0434	1600	6.28E-05	1900	2.01E-05	2100
0.0525	1400	9.30E-04	2700	2.03E-05	1900
0.0525	2700	3.00E-03	5000	2.20E-05	7900
0.0587	1200	9.37E-03	7700	2.38E-05	8800
0.0623	2700	1.88E-02	5400	2.48E-05	12000
0.0712	2000	2.38E-02	6500	4.50E-04	1400
0.0995	2100	3.28E-02	8000	4.68E-04	19000
0.9524	4800	1.31E-01	8500	3.71E-03	17000
0.9524	15000	4.99E-01	17000	3.89E-01	17000
0.9524	16000	9.10E-01	17900	3.92E-01	18500

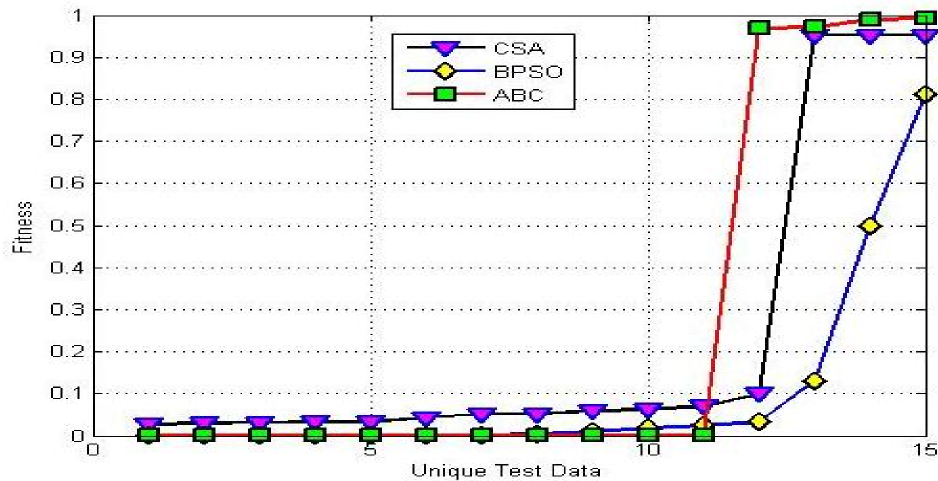

Figure 5.7: Fitness variation for test data

Figure 5.7 depicts the variation of fitness value of unique sets of test data generated when the three meta-heuristic search algorithms are applied. The algorithms are run for 1000 generations. The graph shows that the ABC optimization algorithm provides better fitness values for the generated test data when compared with the BPSO and CSA algorithms.

5.6.4 Code coverage analysis

Table 5.6 gives the analysis result of the obtained code coverage, by covering the nodes traversed in the ECFG. The test data of the respective fitness function values (of CSA, BPSO and ABC) are used separately to achieve code coverage. Results shown in Table 5.6 indicate that the ABC obtained better coverage of nodes in the paths traversed when compared to CSA and BPSO.

Table 5.6: Code coverage analysis for proposed methodology

Meta-heuristic Technique	**No. of Paths Covered Out of 7**	**Code Coverage (%)**
CSA	4	57.14%
BPSO	4	57.14%
ABC	5	71.42%

Table 5.7 shows the obtained code coverage results for Triangle classification problem by Saini *et al* [136].

Table 5.7: Code coverage analysis for existing methodology

Meta-heuristic Technique	**Code Coverage (%)**
GA	56.62%
CSA	58.84%

When Table 5.6 and Table 5.7 are compared, it is found that CSA gives a bit higher code coverage for Triangle classification problem than Bank ATM withdrawal case. This shows the CSA obtains better coverage in this case. It is noted that the performance of these meta-heuristic search techniques depends on the type of application (varying fitness function).

5.7 Summary

In this chapter, an attempt to generate test data automatically for Object-Oriented methodology has been made using three meta-heuristic algorithms. Based on the condition of the predicate node the test data were generated. To generate suitable test data, methods were traversed to cover each node. Test data values are selected based on affinity values of individuals which satisfy the predicate node condition. It was observed that ABC algorithm generates more suitable test data based on the affinity value.

Chapter 6

Conclusions and future scope of work

The conclusive remarks based on the experimental work carried out are included in this chapter along with the scope for future work.

Chapter 1 introduced the overview of the research in the present thesis. It explains the problem at hand, and the proposed solution to handle those by applying different artificial intelligent techniques. This chapter also gave a brief overview of the thesis. This chapter indicated the motive behind the work, and the research objectives formulated based on the motivation. Finally, the chapter summarized the contributions of the thesis, followed by the brief outline of the thesis.

Chapter 2 gave a note on the literature review work, done in the area of fault prediction analysis and test data generation. The survey mainly summarized the work already done by different researchers and practitioners in these areas.

In the first part of the survey, emphasis was laid on the metric suites used, the datasets used for building prediction models, and the methods used to bring out the prediction results. To achieve this, survey related to previous studies for softwares designed by using object-oriented metrics for fault proneness was carried out. This review was helpful to conclude that, among the numerous

metric suites available, CK metrics suite was found to be exclusively used by different authors in their analysis. It was also observed that 'Machine learning methods' were more often used in most of the studies to improve the performance of prediction when compared with statistical approaches. It was further observed that various authors have used datasets available in PROMISE and NASA repositories.

The second part of the survey emphasized on the work done in the area of test data generation, and the respective criteria used by the authors. It is observed that many authors have used optimization algorithms to achieve the desired results. As part of this work, seven meta-heuristic search algorithms were applied for test data generation for traditional methods. It can be concluded that CSA was able to obtain effective test data when compared to the other six algorithms.

Chapter 3 emphasized on the models designed for predicting the fault proneness using different machine learning models.

In the first phase of this chapter, the fault prediction accuracy rate was determined for two statistical and four neural network methods using the full feature set of the CK metric suite. In the second phase of the chapter, the effectiveness of feature reduction techniques such PCA and RST were studied. At the end, the chapter draws a comparative analysis on the obtained fault prediction accuracy rate. It can be concluded that the hybrid approach of RBFN obtained better fault prediction rate when both the full feature and reduced feature data sets were used. Also it is observed that the use of feature reduction techniques slightly enhanced the accuracy rate.

Chapter 4 proposed a cost based evaluation framework, to evaluate the usability of the prediction models designed in Chapter 3. In this work, Classification of faults into faulty or not-faulty classes was carried out using classifiers such as logistic regression and other neural network classifiers such as ANN, FLANN, PNN and RBFN. The usability of the prediction models were evaluated based on the proposed 'Normalized Estimated fault removal cost'

(NEcost), and a trade-off between the obtained classification rate. A prediction model which obtained the least NEcost value was chosen as a suitable model among the set of models designed. Also it implied that a model which obtained an NEcost greater than the threshold value (one) were found to be unsuitable for prediction, and such classes were further unit tested. The chapter concludes that, the Gradient descent approach of RBFN obtained promising results in terms of fault removal cost when compared with LR, GD, LM, FLANN and PNN classifier models.

Chapter 5 proposed an approach to generate test data automatically using three meta-heuristic search techniques. In the literature survey done in this thesis, test data were generated for traditional methods. This chapter concentrated on generating test data for object-oriented methodology. A case study on withdrawal operation in an Bank ATM was considered to employ our approach. As it was not feasible to generate test data for all the tasks, it is necessary to select limited number of important logical paths, for which testing should be exercised. Important paths can be probed for validity, and selectively ensure that the working of the software is correct. In this chapter, test data were generated for only the Bank ATM withdrawal task. As the control flow graph differs from traditional methods, an ECFG was designed for object-oriented methodology. Similarly, the E-CC value was computed based on the designed ECFG. Three meta-heuristic algorithms such as CSA, BPSO, and the ABC algorithms were applied to generate test data. From the implementation work carried out in this chapter, it can be concluded that ABC algorithm was able to obtain suitable test data in comparison with CSA and BPSO algorithms.

The overall conclusions that can be drawn from the work presented in this thesis is that the findings will be beneficial for the researchers, practitioners, and the software professionals because of the following issues being addressed:

1. The designed prediction models can be used to evaluate the quality of the software in the early phases of SDLC, as fault prediction is helpful

in identifying fault prone classes. This will ensure that the testing time and resources are properly utilized.

2. In pipeline to the above key point, after the design phase is complete, the prediction done will be helpful in identifying the highly fault prone classes so that additional testing techniques/effort can be employed on the fault-prone classes during testing phases. Even the design of the software can be reconstructed if a particular class is highly fault prone.

3. A subset of software metrics i.e., CK metrics suite was considered as independent variables, and the fault as dependent variable, where these can be utilized to predict faulty classes.

4. Researchers may use more number of hybrid approaches of machine learning methods rather than statistical methods to classify the classes as faulty or not faulty, so as to obtain better fault prediction accuracy.

5. Generating optimal test data automatically from a huge search space, will be helpful in reducing the testing resources and time.

The contributions as well as the experimental data obtained from this work are as follows:

- The ability of software metrics (CK metric suite) to find out fault prone modules efficiently using machine learning methods over statistical methods was studied (in Chapter-3). The experimental data obtained & the comparison made, indicated that CK metrics suite are better indicators of fault prediction when compared to other models existing in literature. Also the effectiveness of feature reduction methods, which obtain reduced data set were studied to know the performance of these techniques in fault prediction.

- It is a difficult task to deliver a 100% defect free software to the customers. In order to obtain a less defective software, classification of faulty classes

is necessary. This was achieved by performing fault classification (in Chapter-3). Further, a cost based evaluation framework was proposed (in Chapter-4) to identify faulty and non-faulty classes. From the obtained results it was noticed that, performing fault prediction analysis is necessary or not based on the cost incurred in fault identification.

- Testing ensures that the software is defect free i.e, it is carried out with an intention to find errors. In Chapter-5, test data were generated for an object-oriented application by using meta-heuristic search techniques. The results obtained, showed that ABC algorithm generated suitable test data when compared to CSA and BPSO. Further, even code coverage analysis carried out indicates that ABC algorithm achieved better coverage that CSA and BPSO.

6.1 Future scope of work

No one can unwrap the future. The future is a state based on the series of events that have taken place since the initial state. In the long run, the effort put in are of concern. Triggered by this thought process, all the work reported in this thesis work will lead to extension that will enhance their impact in the specific area of work.

The research work carried out, is an empirical study to find the effect of object-oriented metrics on fault proneness. The results provide the necessary guidance for future research on the impact of object-oriented metrics on fault proneness. Some areas, which this thesis may help to extend in future are:

1. To couple various neural network models with meta-heuristic algorithms such as GA, PSO, ABC etc. to achieve better classification rate.

2. To generate test data using affinity function for multiple paths in ECFG.

3. To perform code coverage analysis based on the generated test data for multiple paths.

Bibliography

[1] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in software quality-concept and definitions of software quality," Rome Air Development Center, Tech. Rep. RADC-TR-77-369, November 1977.

[2] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. McLeod, and M. Merritt, *Characteristics of Software Quality.* North-Holland Publihser Co.,, 1978.

[3] R. G. Dromey, "A model for software product quality," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 146–162, February 1995.

[4] R. S. Pressman, *Software Engineering - A practitioner's approach.* McGraw Hill Higher Education, June 2000.

[5] M. Wakil, A. Bastawisi, M. Boshra, and A. Fahmy, "Object-oriented design quality models: A survey and comparison," in *Proceedings of 2nd International Conference on Informatics and Systems*, March 2004, pp. 1–11.

[6] R. B. Grady and D. L. Caswell, *Software metrics: Establishing a company-wide program.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1987.

[7] S. Parnami, K. S. Sharma, and S. V. Chande, "A survey on generation of test cases and test data using artificial intelligence techniques," *UACEE International Journal of Artificial Intelligence and Neural Networks*, vol. 2, no. 1, pp. 16–18, April 2012.

[8] M. Warren and P. Walter, "A logical calculus of ideas immanent in nervous

activity," *Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, December 1943.

[9] B. Christian and R. Andrea, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, September 2003.

[10] T. Back, U. Hammel, and H. P. Schwefel, "Evolutionary computation: Comments on the history and current state," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 3–17, April 1997.

[11] J. Brownlee, *Clever Algorithms: Nature-Inspired Programming Recipes.* Lulu Enterprises Inc., 2011.

[12] L. N. De Castro and F. J. Von Zuben, "Learning and optimization using the clonal selection principle," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 3, pp. 239–251, June 2002.

[13] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, "Self-nonself discrminination in a compuer," in *Proceedings of IEEE Symposium on Research in Security and Privacy.* IEEE, Los Alamitos, CA, 1994, pp. 202–212.

[14] J. Timmis, M. Neal, and H. John, "An artificial immune system for data analysis," *Journal of BioSystems*, vol. 55, no. 1, pp. 143–150, February 2000.

[15] M. Blaha and J. Rumbaugh, *Object-oriented modeling and design with UML.* Pearson Education, 2005.

[16] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, December 1976.

[17] M. H. Halstead, *Elements of Software Science.* New York, USA: Elsevier Science, 1977.

[18] W. Li and S. Henry, "Maintenance metrics for the object-oriented paradigm," in *Proceedings of First International Software Metrics Symposium*, 1993, pp. 52–60.

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

[20] F. B. E. Abreu and R. Carapuca, "Object-oriented software engineering: Measuring and controlling the development process," in *Proceedings of the 4th International Conference on Software Quality*, October 1994, pp. 1–8.

[21] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics.* NJ, Englewood: Prentice-Hall, 1994.

[22] R. Martin, "Object-oriented design quality metrics - an analysis of dependencies," in *Proceedings Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, October 1994.

[23] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi, "A software complexity model of object-oriented systems," *Decision Support Systems*, vol. 13, pp. 241–262, March 1995.

[24] W. Melo and F. B. E. Abreu, "Evaluating the impact of object-oriented design on software quality," in *Proceedings of the 3rd International Software Metrics Symposium*, 1996, pp. 90–99.

[25] L. C. Briand, P. Devanbu, and W. Melo, "An investigation into coupling measures for c++," in *Proceedings of International Conference on Software Engineering Association for Computing Machinery*, 1997, pp. 412–421.

[26] L. Etzkorn, J. Bansiya, and C. Davis, "Design and code complexity metrics for object-oriented classes," in *Object-Oriented Programming*, March 1999, pp. 35–40.

[27] M. Cartwright and M. Sheppered, "An empirical investigation of an object oriented software system," *IEEE Transactions on Software Engineeering*, vol. 26, no. 8, pp. 786–796, August 2000.

[28] R. Burrows, F. Ferrar, O. Lemos, A. Garcia, and F. Ta, "The impact of coupling on the fault-proneness of aspect-oriented programs: An empirical study," in *Proceedings of 21st International Symposium on Software Reliability Engineering*, San Jose, 2010, pp. 329–338.

[29] K. K. Aggarwal, S. Yogesh, K. Arvinder, and R. Malhotra, "Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study," *Software Process: Improvement and Practice*, vol. 14, no. 1, pp. 39–62, August 2009.

[30] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," *IEEE Transactions on Software Engineeering*, vol. 33, no. 6, pp. 402–419, June 2007.

[31] H. Leung and Y. Zhou, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771–789, October 2006.

[32] F. Wu, "Empirical validatin of object-oriented metrics on NASA for fault prediction," in *Proceedings of International Conference on Advances in Information Technology and Education.* Springer, 2011, pp. 168–175.

[33] G. Pai and J. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on Software Engineeering*, vol. 33, no. 10, pp. 675–686, October 2007.

[34] Y. Singh, A. Kaur, and R. Malhotra, "Empirical validation of object-oriented metrics for predicting fault proneness models," *Software Quality Journal*, vol. 18, no. 1, pp. 3–35, March 2010.

[35] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented sys-

tems," *The Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, May 2000.

[36] E. Emam K, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *The Journal of Systems and Software*, vol. 56, pp. 63–75, August 2001.

[37] 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, October 2005.

[38] N. Nagappan, L. Williams, V. Mladen, and O. Jason, "Early estimation of software quality using in-process testing metrics: a controlled case study," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, July 2005.

[39] S. Kanmani, U. V. Rhymend, V. Sankaranarayanan, and P. Thambidurai, "Object-oriented software fault prediction using neural networks," *Journal of Information and Software Technology*, vol. 49, no. 5, pp. 483–492, May 2007.

[40] P. Tomaszewski, J. Hakansson, L. Lundberg, and H. Grahn, "Statistical models vs. expert estimation for fault prediction in modified code - an industrial case study," *The Journal of Systems and Software*, vol. 80, pp. 1227–1238, August 2007.

[41] P. Tomaszewski and H. Grahn, "Improving fault detection in modified code - a study from the telecommunication industry," *Journal of Computer Science and Technology*, vol. 22, no. 3, pp. 397–409, May 2007.

[42] R. Shatnawi and W. Li, "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process," *The Journal of Systems and Software*, vol. 81, no. 11, pp. 1868–1882, November 2008.

[43] Y. Singh, A. Kaur, and R. Malhotra, "Software fault proneness prediction using support vector machines," in *Proceedings of the World Congress on Engineering*, July 1–3, London, U. K, 2009.

[44] A. Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Third International Symposium on Empirical Software Engineering and Measurement*, Lake Buena Vista, FL, 2009, pp. 460–463.

[45] Y. Zhou, B. Xu, and H. Leung, "On the ability of complexity metrics to predict fault-prone classes in object oriented systems," *The Journal of Systems and Software*, vol. 83, no. 4, pp. 660–674, 2010.

[46] M. Ruchika and S. Yogesh, "On the applicability of machine learning techniques for object-oriented software fault prediction," *Software Engineering: An International Journal (SEIJ)*, vol. 1, no. 1, pp. 24–37, September 2011.

[47] M. Bharavi and K. K. Shukla, "Defect prediction for object oriented software using support vector based fuzzy classification model," *International Journal of Computer Applications*, vol. 60, no. 15, pp. 8–16, December 2012.

[48] M. Ruchika and J. Ankita, "Fault prediction using statistical and machine learning methods for improving software quality," *Journal of Information Processing Systems*, vol. 8, no. 2, pp. 241–262, June 2012.

[49] H. Kapila and S. Satwinder, "Analysis of CK metrics to predict software fault-proneness using bayesian inference," *International Journal of Computer Applications*, vol. 74, no. 2, pp. 1–4, July 2013.

[50] J. Holand, *Adaptation in Nature and Artificial Systems.* Cambridge: MIT Press, 1975.

[51] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of International Conference on Neural Networks.* IEEE, 1995, pp. 1942–1948.

[52] K. James and C. E. Russell, "A discrete binary version of the particle swarm algorithm," in *Proceedings of IEEE international conference on systems, man, and cybernetics, Orlando, Florida*, 1997, pp. 4104–4108.

[53] V. Cerny, "A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm," *Journal of Optimization Theory and Applications*, vol. 45, no. 1, pp. 41–51, January 1985.

[54] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085–1110, December 2001.

[55] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, March 2010.

[56] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, June 2004.

[57] T. Seeley, *The Wisdom of the Hive.* Harvard University Press, Cambridge, MA, 1995.

[58] M. P. Kevin, "Bacterial foraging optimization," *International Journal of Swarm Intelligence Research*, vol. 1, no. 1, pp. 1–16, June 2010.

[59] S. Srikrishna and P. Seeni, "Bacterial foraging algorithm based parameter estimation of three winding transformer," *Energy and Power Engineering*, vol. 3, no. 2, pp. 135–143, May 2011.

[60] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 263–282, December 1999.

[61] J. C. Lin and P. L. Yeh, "Automatic test data generation for path testing using GAs," *Information Sciences*, vol. 131, no. 1, pp. 47–64, January 2001.

[62] N. Mansour and M. Salame, "Data generation for path testing," *Software Quality Journal*, vol. 12, no. 2, pp. 121–136, June 2004.

[63] M. A. Ahmed and I. Hermadi, "GA-based multiple paths test data generator," *Computers and Operations Research*, vol. 35, no. 10, pp. 3107–3124, October 2008.

[64] C. Chen, X. Xu, Y. Chen, X. Li, and D. Guo, "A new method of test data generation for branch coverage in software testing based on EPDG and genetic algorithm," in *Proceedings of 3rd International Conference on Anti-counterfeiting, security and identification in communication*, 2009, pp. 307–310.

[65] P. R. Srivastava and T. H. Kim, "Application of genetic algorithm in software testing," *International Journal of software Engineering and its Applications*, vol. 3, no. 4, pp. 87–96, October 2009.

[66] I. Alsmadi, F. Alkhateeb, E. A. Maghayreh, S. Samarah, and I. A. Doush, "Effective generation of test cases using genetic algorithms and optimization theory," *Journal of Communication and Computer*, vol. 7, no. 11, pp. 72–82, November 2010.

[67] A. Rauf, S. Anwar, M. A. Jaffer, and A. A. Shahid, "Automated GUI test coverage analysis using GA," in *Proceedings of 7th International Conference on Information Technology: New Generations (ITNG)*. IEEE, 2010, pp. 1057–1062.

[68] H. Tahbildar and B. Kalita, "Heuristic approach of automated test data generation for program having array of different dimensions and loops with variable number of iteration," *International Journal of Software Engineering and Applications (IJSEA)*, vol. 1, no. 4, pp. 75–93, October 2010.

[69] C. Sharma, S. Sabharwal, and R. Sibal, "A survey on software testing techniques using genetic algorithm," *International Journal of Computer Sciences Issues*, vol. 10, no. 1, pp. 381–393, January 2013.

[70] S. Swain and D. Mohapatra, "Genetic algorithm-based approach for adequate test data generation," in *Proceedings of International Conference on Advanced Computing, Networking, and Informatics*, 2013, pp. 453–462.

[71] A. Nunez, M. G. Merayo, R. M. Hierons, and M. Nunez, "Using genetic algorithms to generate test sequences for complex timed systems," *Journal of Soft Computing*, vol. 17, no. 2, pp. 301–315, February 2013.

[72] W. Andreas, W. Stefan, and W. Joachim, "Applying particle swarm optimization to software testing," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, New York, USA*, vol. 1. IEEE, 2007, pp. 1121–1128.

[73] A. Li and Y. Zhang, "Automatic generating all-path test data of a program based on PSO," in *Proceedings of World Congress on Software Engineering*, vol. 4. IEEE, 2009, pp. 189–193.

[74] C. Huanhuan, C. Li, Z. Bian, and K. Halei, "An efficient automated test data generation method," in *Proceedings of International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, vol. 1. IEEE, 2010, pp. 453–456.

[75] S. Zhang, Y. Zhang, H. Zhou, and Q. He, "Automatic path test data generation based on GA-PSO," in *Proceedings of IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS)*, vol. 1. IEEE, 2010, pp. 142–146.

[76] C. Mao, X. Yu, and J. Chen, "Swarm intelligence-based test data generation for structural testing," in *Proceedings of IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS)*. IEEE, 2012, pp. 623–628.

[77] R. Ding, X. Feng, S. Li, and H. Dong, "Automatic generation of software test data based on hybrid particle swarm genetic algorithm," in *Proceedings of Symposium on Electrical & Electronics Engineering (EEESYM)*. IEEE, 2012, pp. 670–673.

[78] S. A. Khan and A. Nadeem, "Automated test data generation for coupling based integration testing of object oriented programs using particle swarm op-

timization," in *Proceedings of the Seventh International Conference on Genetic and Evolutionary Computing, ICGEC*, 2013, pp. 115–124.

[79] X. Xu, Y. Chen, X. Li, and D. Guo, "A path-oriented test data generation approach for automatic software testing," in *Proceedings of 2nd International Conference on Anti-counterfeiting, Security and Identification (ASID).* IEEE, 2008, pp. 63–66.

[80] K. Liaskos and M. Roper, "Hybridizing evolutionary testing with artificial immune systems and local search," in *Proceedings of International Conference on Software Testing, Verification and Validation.* IEEE, 2008, pp. 211–220.

[81] A. Pachauri and Gursaran, "Use of clonal selection algorithm as software test data generation technique," in *Proceedings of Second International Conference on Advanced Computing & Communication Technologies (ACCT).* IEEE, 2012, pp. 1–5.

[82] A. Khushboo, P. Ankur, and Gursaran, "Towards software test data generation using binary particle swarm optimization," in *Proceedings of XXXII National Systems Conference (NSC), 17th-19th December, Roorkee, India*, vol. 1, 2008, pp. 339–343.

[83] P. Ankur and Gursaran, "Comparative evaluation of a maximization and minimization approach for test data generation with genetic algorithm and binary particle swarm optimization," *International journal of software engineering and applications (IJSEA)*, vol. 3, no. 1, pp. 207–218, January 2012.

[84] G. Haichang, F. Boqin, and Z. Li, "A kind of SA-GA hybrid meta-heuristic algorithm for the automatic test data generation," in *Proceedings of International Conference Neural Networks and Brain*, 2005, pp. 111–114.

[85] L. Bao, L. Zhi-Shu, Z. Jing-Yu, and S. Ji-Rong, "An automated test case generation approach by genetic simulated annealing algorithms," in *Proceedings of Third International Conference on Natural Computation (ICNC)*, 2007, pp. 106–111.

[86] X. Tan, C. Longxin, and X. Xiumei, "Test data generation using annealing immune genetic algorithm," in *Proceedings of 5th International Joint Conference on INC, IMS and IDC*, 2009, pp. 344–348.

[87] B. Zhang and C. Wang, "Automatic generation of test data for path testing by adaptive genetic simulated annealing algorithm," in *Proceedings of International Conference on Computer Science and Automation Engineering(CSAE)*. IEEE, China, 2011, pp. 38–42.

[88] L. Gentiana Ioana, C. Octavian Augustin, and L. Vacariu, "Automatic test data generation for software path testing using evolutionary algorithms," in *Proceedings of Third International Conference on Emerging Intelligent Data and Web Technologies (EIDWT)*, Bucharest, Romania, 2012, pp. 1–8.

[89] D. J. Mala and V. Mohan, "ABC tester - artificial bee colony based software test suite optimization approach," *International Journal of Software Engineering, Sprinter Global Publication*, vol. 2, no. 2, pp. 1–33, July 2009.

[90] D. Surender Singh, C. Jitender Kumar, and K. Shakti, "Application of artificial bee colony algorithm to software testing," in *Proceedings of 21st Australian Software Engineering Conference , Auckland, New Zealand*, vol. 1, 2010, pp. 149–154.

[91] K. Arvinder and G. Shivangi, "A bee colony optimization algorithm for code coverage test suite prioritization," *International journal of engineering science and technology (IJEST)*, vol. 3, no. 4, pp. 2786–2797, April 2011.

[92] A. Srikanth, N. J. Kulkarni, N. K. Venkat, P. Singh, and P. R. Srivastava, "Test case optimization using artificial bee colony algorithm," in *Proceedings of Advances in Computing and Communications*, vol. 192, March 2011, pp. 570–579.

[93] L. Soma Sekhara Babu, M. L. Hari Prasad Raju, M. Uday Kiran, C. Swaraj, and P. R. Srivastav, "Automated generation of independent paths and test suite optimization using artificial bee colony," in *Proceedings of International*

Conference on Communication Technology and System Design, 2012, pp. 191–200.

[94] R. K. Sandhu, A. Puri, and H. S. Gill, "A novel approach for software testing based on artificial bee colony technique," in *Proceedings of International Conference on Sustainable Manufacturing and Operations Management*, 26th-28th June 2013, pp. 406–410.

[95] S. Dalal and R. S. Chhillar, "A novel technique for generation of test cases based on bee conlny optimization and modified genetic algorithm," *International Journal of Computer Applications*, vol. 68, no. 19, pp. 12–16, April 2013.

[96] J. Dobbins, "IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software," Institute of Electrical and Electronics Engineers, New York, NY, USA, Tech. Rep. IEEE Std 982.2-1988, June 1989.

[97] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, October 1996.

[98] T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan. (2012, June) The promise repository of empirical software engineering data. [Online]. Available: http://promisedata.googlecode.com

[99] K. J. Yogendra and K. B. Santosh, "Min-max normalization based data perturbation method for privacy protection," *International Journal of Computer and Communication Technology*, vol. 2, no. 8, pp. 45–50, October 2001.

[100] R. Battiti, "First and second-order methods for learning between steepest descent and newtons method," *Neural Computation*, vol. 4, no. 2, pp. 141–166, March 1992.

[101] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quarterly Journal of Applied Mathematics*, vol. 2, no. 2, pp. 164–168, July 1944.

[102] D. W. Marquardt, "An algorithm for the least-squares estimation of nonlinear parameters," *SIAM Journal of Applied Mathematics*, vol. 11, no. 2, pp. 431–441, June 1963.

[103] D. S. Broomhead and L. David, "Multivariable functional interpolation and adaptive networks," *Complex Systems*, vol. 2, no. 3, pp. 321–355, 1988.

[104] J. Moody and J. Darken C, "Fast learning in networks of locally-tunes processing units," *Neural Computation*, vol. 1, no. 2, pp. 281–294, Summer 1989.

[105] Y. H. Pao, *Adaptive pattern recognition and neural networks.* Addison-Wesley, 1989.

[106] D. F. Specht, "Probabilistic neural networks," *Neural Networks*, vol. 3, no. 1, pp. 109–118, 1990.

[107] P. Karl, "On lines and planes of closest fit to systems of points in space," *Philosophical magazine*, vol. 2, no. 11, pp. 559–572, 1901.

[108] Z. Pawlak, "Rough sets," *International Journal of Computer and Information Sciences*, vol. 11, no. 5, pp. 341–356, October 1982.

[109] R. Sowinski, *Intelligent Decision Support: Handbook of Applications and Advances of the Rough Sets Theory.* Kluwer Academic Publishers, 1992.

[110] C. Cagatay, "Performance evaluation metrics for software fault prediction studies," *Acta Polytechnica Hungarica*, vol. 9, no. 4, pp. 193–206, 2012.

[111] X. Yuan, Khoshgoftaar, E. B. Allen, and K. Ganesan, "Application of fuzzy clustering to software quality prediction," in *Proceedings of 3rd Symposium on ASSEST.* IEEE, 2000, pp. 85–91.

[112] G. Denaro, M. Pezzem, and S. Morasca, "Towards industrially relevant fault proneness models," *International Journal of Software Engineering and Knowledge Engineering*, vol. 13, no. 4, pp. 395–417, August 2003.

[113] E. Arisholm, L. Briand, and E. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *Journal of System and Software*, vol. 83, no. 1, pp. 2–17, January 2010.

[114] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, April 2005.

[115] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 561–595, October 2008.

[116] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, August 2008.

[117] J. Demsar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, no. 1, pp. 1–30, January 2006.

[118] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering (PROMISE)*, New York, USA, 2009, pp. 7:1–7:10.

[119] T. Mende and K. R, "Effort-aware defect prediction models," in *Proceedings of 14th IEEE European Conference on Software Maintenance and Re-engineering (CSMR)*, March 2010, pp. 107–116.

[120] S. Wagner, "A literature survey of the quality economics of defect-detection techniques," in *Proceedings of the International Symposium on Empirical Software Engineering (ISESE)*. ACM/IEEE, 2006, pp. 194–203.

[121] C. Jones, *Software quality in 2010: a survey of the state of the art.* Founder and Chief Scientist Emeritus, December 2010.

[122] R. Huitt and N. Wilde, "Maintenance support for object-oriented programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, pp. 1038–1044, December 1992.

[123] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, August 1990.

[124] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select - a formal system for testing and debugging programs by symbolic execution," vol. 10, no. 6, pp. 234–245, June 1975.

[125] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 215–222, September 1976.

[126] W. E. Howden, "Symbolic testing and the dissect symbolic evaluation system," *IEEE Transactions on Software Engineering*, vol. 4, no. 4, pp. 266–278, July 1977.

[127] J. A. Bauer and A. B. Finger, "Test plan generation using formal grammars," in *Proceedings of the 4th International Conference on Software Engineering.* IEEE, 1979, pp. 425–432.

[128] W. Jessop, J. R. Kane, S. Roy, and J. Scanlon, "Atlas - an automated software testing system," in *Proceedings of the 2nd International Conference on Software Engineering.* IEEE Computer Society Press, 1976, pp. 629–635.

[129] N. R. Lyons, "An automatic data generating system for data base simulation and testing," *ACM SIGSIM Simulation Digest*, vol. 8, no. 4, pp. 8–11, June 1977.

[130] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases," *IBM Systems Journal*, vol. 22, no. 3, pp. 229–245, 1983.

[131] W. Xibo and S. Na, "Automatic test data generation for path testing using genetic algorithms," in *Proceedings of Third International Conference on Mea-*

suring Technology and Mechatronics Automation (ICMTMA), vol. 1. IEEE, 2011, pp. 596–599.

[132] M. Grindal, J. Offutt, and J. Mellin, "On the testing maturity of software producing organizations," in *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006.* IEEE, 2006, pp. 171–180.

[133] K. Ananya and B. Swapan, "Static analysis of object-oriented systems using extended control flow graph," in *Proceedings of TENCON.* IEEE, 2004, pp. 310–313.

[134] M. J. C Ghezzi and D. Mandrioli, *Fundamentals of Software Engineering.* Prentice Hall, India, 1998.

[135] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman, "Test data generation and feasible path analysis," in *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis.* ACM, 1994, pp. 95–107.

[136] S. Poonam and S. Tyagi, "Test data generation for basis path testing using genetic algorithm and clonal selection algorithm," *International Journal of Science and Research*, vol. 3, no. 6, pp. 995–998, June 2014.

Dissemination

Journals

1. **Yeresime Suresh**, Lov Kumar and Santanu Ku. Rath, "Statistical and Machine Learning Methods for Software Fault Prediction using CK Metric Suite: A Comparative Analysis", *ISRN Software Engineering*, Vol. 2014, pp. 1-15, January 2014, Hindawi Publishers.

2. **Yeresime Suresh**, Lov Kumar and Santanu Ku. Rath, "Framework to Assess the Effectiveness of Software Fault Classification", *International Journal of Information Processing*, vol. 8, no. 2, pp. 1-12, June 2014, IK Publishers.

3. **Yeresime Suresh**, and Santanu Ku. Rath, "Evolutionary Algorithms for Object-Oriented Test Data Generation", *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 4, pp. 1-6, July 2014.

4. **Yeresime Suresh**, and Santanu Ku. Rath, "Application of Meta-heuristic Algorithms for Automated Software Test Data Generation", *International Journal of Computational Intelligence Studies*, Inderscience. [Accepted for Publication on: 25th April 2014]

Conferences

1. **Yeresime Suresh**, Meghansh Sharma, Shariq Islam, and Santanu Ku. Rath, “Test Data Generation for Object-Oriented Methodology using Clonal Selection Algorithm", *in Proceedings of CONSEG: 7th International Conference on Software Engineering*, pp. 27-33, 15th-17th November 2013, Pune, India. [Published by: Computer Society of India (CSI)]

2. **Yeresime Suresh**, Lov Kumar and Santanu Ku. Rath,“ A Cost Based Evaluation Framework for Software Fault Classification using Chidamber and Kemerer Metric Suite," *7th IEEE International Conference on Contemporary Computing (IC3)*, 7th-9th August 2014, JIIT Noida (Accepted).

Appendix A

Bank ATM pseudocode

```
package test;

import java.util.Scanner;

public class ATM {

  private int balance;
   int getBalance(){
    return balance;
  }

    void withdraw(int amount){
    if(balance <1000)
      System.out.println("insufficient funds to withdraw");
    else if(balance-amount<1000)
      System.out.println("balance after the withdrawl should not
   be less than 1000");
    else
      deductBalance(amount);
  }
   void deposit(int amount){
    addBalance(amount);
```

```
}
 void deductBalance(int amount){
  int bal;
  bal=getBalance();
  bal-=amount;
  setBalance(bal);

}
 void addBalance(int amount){
  int bal=getBalance();
  bal+=amount;
  setBalance(amount);
}
 void setBalance(int amount){
  balance=amount;
}

/**
 * @param args
 */

public static void main(String[] args) {
  // TODO Auto-generated method stub
      int choice=0;
      ATM a=new ATM();
      while(choice!=-1){
  System.out.println("Enter your choice");
  System.out.println("1.Check Balance");
  System.out.println("2.Withdraw Amount");
  System.out.println("3.Deposit Money");
  System.out.println("Enter -1 to exit");
  Scanner s=new Scanner(System.in);
  choice=s.nextInt();
  if(choice >3 || choice <1)
    continue;
```

```
    else if(choice==1)
      System.out.println(a.getBalance());
    else if(choice==2){
      System.out.println("Enter Amount to withdraw");
      int amount=s.nextInt();
      a.withdraw(amount);
      }
    else if(choice==3){
      System.out.println("Enter Amount to Deposit");
      int amount=s.nextInt();
      a.deposit(amount);
      }


        }
  }

}
```

Listing A.1: Input Source Code for Bank ATM

Appendix B

Code Coverage Analysis

```
Code Coverage Analysis:-
clear
clc
    min = 1000;
    max = 20000;
tot_max = 45000;
pin_c=0; mod_c=0; max_c=0; more_c=0; less_c=0; tmax_c=0; wid_c=0;
ch='n';

fileID=fopen('exp.txt','r');
data=fscanf(fileID,'%d',3);
pin=data(1);
tot_amt=data(2);
bal=data(3);
fclose(fileID);

upin = input('Enter 4 Digit Pin Number : ');
if pin~=upin
    fprintf('Wrong PIN Number!!!!!\n Insert the card again\n');
    pin_c=1;
else
```

```
    ch='y';
end
while((ch=='y')||(ch=='Y'))
    amt=input('Enter amount to be withdrawn:');


    fileID=fopen('exp.txt','r');
    data=fscanf(fileID,'%d',3);
    pin=data(1);
    tot_amt=data(2);
    bal=data(3);
    fclose(fileID);

    if (mod(amt, 100) ~= 0)
        mod_c=1;
        fprintf('Entered amount is not in multiple of 100\n');
    elseif amt>max
        max_c=1;
        fprintf('Entered amount is greater than maximum withdrawal
     amount which is:%d\n', max);
    elseif amt>bal
        more_c=1;
        fprintf('Entered amount is greater than current balance
    which is:%d\n', bal);
    elseif bal-amt<min
        less_c=1;
        fprintf('Entered amount leads balance to less than minimum
     balance which is:%d\n', min);
    elseif (tot_amt + amt) > tot_max
        tmax_c=1;
        fprintf('Entered amount leads todays transaction balance
    to maximum limit:%d\n', tot_max);
        fprintf('Your todays total withdrawl amount:%d\n', tot_amt
    );
    else
```

```
        wid_c=1;
        bal = bal-amt;
        tot_amt = tot_amt+amt;
        fileID=fopen('exp.txt','w');
        A=[pin; tot_amt; bal];
        for i = 1 : 3
            fprintf(fileID,'%d\n', A(i));
        end
        fclose(fileID);
        fprintf('New balance:%d\n', bal);
    end
        ch=input('Want to continue?(Y/N)','s');
end
fprintf('Total Paths=7\n');
total_path = pin_c + mod_c + max_c + more_c + less_c + tmax_c +
    wid_c;
fprintf('No. of Paths Covered=%d\n', total_path);
coverage=(total_path*100)/7;
fprintf('Path Coverage=%0.2f%%\n', coverage);
```

The above code is the portion of code coverage implementation. The test data achieved through the use of Clonal selection algorithm, Binary particle swarm optimization and Artificial bee colony algorithm are passed as input to the code coverage module, to achieve better coverage during the testing process.