DSpace Institution

DSpace Repository **http://dspace.org**

Computer Science thesis

2021-02-02

Software Defect Prediction Using Source Code Metric and Semantic Features: A Deep Learning Approach

Esrael, Geremew

http://ir.bdu.edu.et/handle/123456789/12385

Downloaded from DSpace Repository, DSpace Institution's institutional repository

BAHIR DAR UNIVERSITY

BAHIR DAR INSTITUTE OF TECHNOLOGY

SCHOOL OF RESEARCH AND POSTGRADUATE STUDIES

FACULTY OF COMPUTING

Software Defect Prediction Using Source Code Metric and Semantic Features: A Deep Learning Approach

MSc. Thesis Final

By

Esrael Geremew

Program: Regular

Main Advisor: Mokonnen Wagaw (Ph.D.)

February 2, 2021

BAHIRDAR, ETHIOPIA

BAHIR DAR UNIVERSITY

BAHIR DAR INSTITUTE OF TECHNOLOGY

SCHOOL OF RESEARCH AND POSTGRADUATE STUDIES

FACULTY OF COMPUTING

Software Defect Prediction Using Source Code Metric and Semantic Features: A Deep Learning Approach

MSc. Thesis Final

By

Esrael Geremew

A Thesis submitted to the school of Research and Graduate Studies of Bahir Dar Institute of Technology, Bahir Dar University in partial fulfillment of the requirements for the degree of Master of Science in the Software Engineering in the Computing Faculty

Program: MSc. in Software Engineering

Main Advisor: Mokonnen Wagaw (Ph.D.)

February 2, 2021

Bahir Dar, Ethiopia

BAHIR DAR UNIVERSITY

BAHIR DAR INSTITUTE OF TECHNOLOGY

SCHOOL OF RESEARCH AND GRADUATE STUDIES

COMPUTING FACULTY

Approval of Thesis for defense

I hereby certify that I have supervised, read, and evaluated this thesis titled 'Software Defect Prediction Using Source Code Metric and Semantic Features: A Deep Learning Approach' prepared by Esrael Geremew under my guidance. I recommend the thesis to be submitted for oral defense.

Mokonnen Wagaw (Ph.D.)	______________	February 2, 2021
Advisor's Name	Signature	Date

DECLARATION

I, the undersigned, declare that the thesis comprises my own work. In compliance with internationally accepted practices, I have acknowledged and refereed all materials used in this work. I understand that non-adherence to the principles of academic honesty and integrity, misrepresentation/ fabrication of any idea/data/fact/source will constitute sufficient ground for disciplinary action by the University and can also evoke penal action from the sources which have not been properly cited or acknowledged.

Name of the student: Esrael Geremew Signature: ____________

Date of submission: February 2, 2021

Place: Bahir Dar

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

Advisor Name: Mokonnen Wagaw (Ph.D.)

Advisor's Signature: ________________

BAHIR DAR UNIVERSITY
BAHIR DAR INSTITUTE OF TECHNOLOGY

SCHOOL OF RESEARCH AND GRADUATE STUDIES
COMPUTING FACULTY
Approval of thesis for defense result

I hereby confirm that the changes required by the examiners have been carried out and incorporated in the final thesis.

Name of Student: Esrael Geremew Signature: ________ Date: February 2, 2021

As members of the board of examiners, we examined this thesis entitled "**Software Defect Prediction Using Source Code Metric and Semantic Features: A Deep Learning Approach**" by Esrael Geremew. We hereby certify that the thesis is accepted for fulfilling the requirements for the award of the degree of Masters of Science in Software Engineering".

Board of Examiners

Name of Advisor	Signature	Date
Mokonnen Wagaw (PhD)	________	February 2, 2021
Name of External examiner	Signature	Date
Elfelious G. Belay (PhD)	________	April 28, 2021
Name of Internal Examiner	Signature	Date
Esubalew Alemneh (PhD)	________	01/06/2021
Name of Chairperson	Signature	Date
Mekuanint A. (PhD)	________	01/06/2021
Name of Chair Holder	Signature	Date
Gebeyeh B. (Dr.)	________	________
Name of Faculty Dean	Signature	Date
Belete B.	________	________

© 2021

ESRAEL GEREMEW

ALL RIGHTS RESERVE

Acknowledgements

I would like to offer my sincere gratitude to the following people who have helped, supported and encouraged me during this unforgettable journey of Thesis work.

- To my supervisor, Mokennen Wagga (Ph.D.), I thank you for advising and challenging me to bring this thesis work towards completion. I am also grateful for all support and guidance that you have provided during my MSc. study. Your supervision and advice on both research as well as on my career have been invaluable.
- To Dr. Esubalew Alemneh thank you for providing insightful thoughts and constructive feedback to guide my research. It is with your motivational thoughts and feedback that this work came into existence.
- To my informants, their names cannot be disclosed, but I want to acknowledge and appreciate their help and transparency during my study and research. Their information has helped me complete this thesis work.
- And lastly, but by no means least, to my family whose love and guidance are with me in whatever I pursue. Words cannot express how grateful I am to my dear parents, and my beloved wife—Haymanot G.—for your inspiration and selfless support during the last two years. I would not have come this far without you.

Abbreviations

Acronym	Definition
AST	Abstract Syntax Tree
Bi-LSTM	Bi-directional Long Sort Term Memory
CDD	Combined Defect Data
CK	Chidamber and Kemerer features
CMSF	Code Metric and Semantic Features
CNN	Convolutional Neural network
CPU	Central Processing Unit
CS	Cost Sensitive
DFT	Depth Frist Transversal
DL	Deep Learning
FCN	Fully Connected Network
HFC	Hand-crafted Features Combination
ML	Machine Learning
MLP	Multi-Layered Perceptron
OOM	Object-Oriented Metrics
PROMISE	Open-source repository of software defect dataset
PSC	PROMISE Source Code
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SDP	Software Defect Prediction
SGD	Stochastic Gradient Descent
SPSC	Simplified PSC
THF	Transformed Hand-crafted features

Table of Contents

ACKNOWLEDGEMENTS .... **IV**

ABSTRACT .... **XI**

1 INTRODUCTION **1**

1.1 BACKGROUND 1

1.1.1 Software defect prediction 2

1.1.2 Deep Learning-based software defect prediction methods 6

1.2 PROBLEM STATEMENT 11

1.3 OBJECTIVE OF THE STUDY 13

1.3.1 General Objective 13

1.3.2 Specific objectives 13

1.4 RESEARCH QUESTIONS 14

1.5 THESIS CONTRIBUTION 14

1.6 ASSUMPTIONS, AND RESEARCH SCOPE 15

1.7 DEFINITION OF TERMS 17

1.8 SUMMARY 18

2 REVIEW OF LITERATURE **19**

2.1 DEFECT PREDICTION USING SOURCE CODE METRIC FEATURES 20

2.2 DEFECT PREDICTION USING SEMANTIC FEATURES 20

2.3 DEFECT PREDICTION USING HYBRID FEATURES 22

3 METHODOLOGY **25**

3.1 APPROACH OVERVIEW AND DATASET COLLECTION 27

3.1.1 Problem formulation 27

3.1.2 Workflow 27

3.1.3 Raw Dataset Collection 28

3.2 COMBINED DEFECT DATA MODELLING FRAMEWORK 31

3.2.1 File-level Defect Data Engineering 31

3.2.2 The Obtained Datasets 41

3.2.3 Experimentation Framework 42

3.3 MODEL BUILDING 42
3.3.1 Raw representations 43
3.3.2 The Deep Neural Networks for Representation Learning 46
3.3.3 Model Evaluation 53
4 EXPERIMENTS, AND RESULTS 55
4.1 FILE-LEVEL DEFECT DATASET QUALITY EVALUATION 56
4.1.1 Experiment Setting 56
4.1.2 Experimental Procedure 57
4.1.3 Results and Analysis 60
4.2 MATCHED SOFTWARE DEFECT DATASET (CDD) QUALITY ANALYSIS (**RQ1**) 64
4.3 EVALUATIONS OF FILE-LEVEL SOFTWARE DEFECT PREDICTION APPROACHES 66
4.3.1 Baseline approaches 66
4.3.2 Baseline Models 67
4.3.3 Obtained Results and Analysis 68
5 DISCUSSIONS, LIMITATIONS, AND CONCLUSION 75
5.1 DISCUSSIONS 75
5.1.1 Individual Information Effectiveness (**RQ2.**) 75
5.1.2 Combined Information Effectiveness (**RQ3**) 78
5.1.3 Prediction Performance Sensitivity Analysis (**RQ4**) 80
5.1.4 Threats to validity 82
5.1.5 Final Remark 83
5.2 CONCLUSION 84
5.2.1 Summary of thesis findings 84
5.2.2 Future works 85
REFERENCES 86
APPENDIX C: FULL EXPERIMENTAL RESULTS 90

List of Tables

Table 1.1: Object-Oriented software metrics (Zhou Xu et al., 2019) 4
Table 2.2: Summary of the recent, relevant and related works 22
Table 3.1: Definition of static code metrics (Cong Pan et al., 2019; Shaojian Qiu et al., 2019).. 30
Table 3.2: Sample Data for Defect data with code metric attributes 34
Table 3.3: Verified Sample software projects with and without Defect Data. 35
Table 3.4: Sample data for combined defect dataset 41
Table 3.5: Statistical result for the extracted java, PROMISE and matched file instances 41
Table 3.6: Representative AST nodes (Ziyi Cai et al., 2017; Miltiadis Allmanis et al., 2018) 44
Table 4.1: Python toolkit and experimental datasets for combined defect data preparation 57
Table 4.2: Experiments, Projects, and Dataset used for Software Engineering related setting. ... 60
Table 4.3: Results for software systems with unactionable files. 62
Table 4.4: Experimental results for sequence length of mapped and filtered ASTs..................... 63
Table 4.5: Experimental data setting of file-level defect prediction.. 66
Table 4.6: Experimental Parameter setting for the language model 67
Table 4.7: Experimental results for prediction models trained in source code metrics 68
Table 4.8: Prediction results using matched source code semantic features 70
Table 4.9: Prediction results using combined information from source codes............................ 72
Table 4.10: Results for defect prediction using unified features representation.......................... 73
Table 5.1: Defect prediction results using source code metric, and/or semantic features 75
Table 5.2: Comparison of defect prediction using source code metric and semantic features 77
Table 5.3: Evaluation of defect prediction performance using d/t features combination 79

List of Algorithms

Algorithm 1: Pseudo code for the extraction of Matched Defect Data.. 33

Algorithm 2: Pseudo code for verifying the existence of defect dataset for a source project. 35

Algorithm 3: Pseudo code for extraction java source code files 36

Algorithm 4: Pseudo code for parsing source code file in to AST representation 37

Algorithm 5: Pseudo code for encoding AST string tokens to numeric tokens............................ 38

Algorithm 6: Pseudo code for combined defect data extraction.. 40

Algorithm 7: Pseudo code for pre-training the LHFR model with combined defect dataset 51

List of Figures

Figure 1.1: A typical software defect prediction process (Cong Pan et al., 2019) 2
Figure 1.2 overview of multilayer perceptron with one hidden layer (Guanjun Lin, 2020)........... 7
Figure 1.3 LSTM and overview of RNN with LSTM 10
Figure 3.1: The process of software defect dataset preparation (i.e. CDDM) 25
Figure 3.2: Overview of deep neural networks for DL-generated features combination 26
Figure 3.3: The processes of SDP using code metric and semantic features............................... 27
Figure 3.4: Overview of the proposed SDP using source code metric and semantic features 28
Figure 3.5 Overview of file-level defect data modelling (CHDD) framework 31
Figure 3.6: Sample data with java source code files... 36
Figure 3.7: Sample data for sequence data. 37
Figure 3.8: Sample data for numeric sequence data. 39
Figure 3.9: Sample data for Numeric sequence data with unified sequence length. 40
Figure 3.10 The architecture of LHFR for combined features extraction 52
Figure 4.2: Bar Chart of Raw Labeled Files and Extracted Source Code Files............................ 61
Figure 4.3: Bar chart of cost-sensitive bug Rates 63
Figure 4.4: Radar charts of optimal SeqLent evaluation... 65
Figure 4.5: Visualization of AST Node types (words) representation as semantic features 71
Figure 5.1: Stacked lines comparing the effectiveness of Matched and PROMISE code metrics 76
Figure 5.2: Prediction performance under different defect classification sachem........................ 80
Figure 5.3: Radar charts for prediction indicators of LHFR model with 3 types of features 81

Abstract

Software Defect Prediction (SDP) aims to observe defective modules to modify the cheap allocation of testing resources, which is associate with economically important activity in software quality assurance. Many contributions have been made in this area through the features representation—from source code metric and/or contextual data—by Deep Learning (DL) models; however, the prediction performance they present with high false positive and unactionable alerts. Specifically, Obtaining Combined Defect Data (CDD), lack of proper features combining methods and learning effective features representation from CDD are some of the major challenges encountered in supervised defect prediction domain.

Combined features have the potential to improve the performance of individual features-based prediction models and thereby reduce false-positive rates. Thus, the goal of this research effort was to improve the performance of individual features-based defect prediction model by proposing and testing novel frameworks named CDDM and LHFR by Combined Defect Data Modelling from source code metrics and context representation and Learning Hybrid Feature Representation from CDD for SDP, respectively. Specifically, we use a deep neural network with a new hybrid network that consists of a Multi-Layered Perceptron (MLP) to learning a more discriminative features representation of the hand-crafted features and Bi-directional Long Short Term Memory (Bi-LSTM) to learn semantic features. To evaluate the effectiveness of the proposed frameworks, we conduct extensive experiments on a benchmark dataset with 12 software defect datasets (each with four types of features), using five traditional indicators. Comparing to the random forest model built solely using the individual features set, our LHFR approach improves the average accuracy, precision, recall, F1-score, and AUC by 1.6%, 0.71%, 1.6%, 2.27%, and 0.01% respectively.

This work contributes a file-level combined software defect dataset based upon 12 open-source java systems. It also enhanced an existing hand-crafted data set generation framework to include additional software context-based predictive features. However, the main contribution is a feature combination methodology that may be used to discover effective features combination and representations that increase the defect prediction performance.

Key Words: - Deep Learning-generated Features Combination, Combined Defect Data, Matched Defect Data, Metric Features, and Semantic Features.

Chapter One

1 Introduction

1.1 Background

Software applications are almost certain to produce failures due to defects/bugs introduced during the development or maintenance phases (Haonan Tong et al., 2018). A good number of these errors are discovered by the testing team and few of them are discovered over time in the operational phase but it has been reported that the cost of fixing a bug in later stages of the software development cycle can be very expensive when compared to the development phase, Therefore one must take this into account and try to find and fix bugs as early as possible (Zhou Xu et al., 2019). The simplest way to find bugs is by software testing is defined as 'the process of executing a program with the intent of finding errors' (Alyahya, 2020). There are different types of testing such as unit testing, function testing, system testing, and integration testing. It has been observed that most of the time the testing activities consume anything in between 45% to 75% of the total development time (Alegre, 2017).

The increasing complexity of modern software has elevated the importance of software reliability and building highly reliable software requires a considerable amount of testing and debugging (Cong Pan et al., 2019). However, since both budget and time efforts are limited, and must be prioritized for efficiency. As a result, SDP techniques that predict the occurrence of bugs/faults/errors i.e. defects in a software module, have been widely used to assist developers in prioritizing their testing and debugging efforts (Le Hoang Son et al., 2019). SDP is the process of building the ML model as a classifier model which classifies the label of a software module (i.e. package, file, functions, and so on) with, or without defective properties/characteristics as defective or non-defective respectively; and predictive model which predicates the level/probability of a given instance to be defective (Jayanthi. R et al., 2017). The prediction results can assist software development practitioners, for example, the software developers in prioritizing their testing and debugging efforts (Miltiadis Allmanis et al., 2018). The application of defect prediction in the processes of software development) contributed to the improvement of software product's quality (Aston Zhang et al., 2019).

From the perspective of defect prediction granularity, SDP can include method-level, class-level, file-level, package-level, and change-level defect prediction. This study is focused on file-level

defect prediction due to: the existence of enough amount of file-level defect datasets i.e. labeled file instances for building an accurate ML model as a predictive model and the performance of ML models depends on the amount of training dataset (Alegre, 2017; Hoa Khanh Dam et al., 2018).

1.1.1 Software defect prediction

Software defect prediction is an important research problem in the field of software engineering with the following four main phases as depicted in Figure 1.1 that shows a typical defect prediction process (Tianchi Zhou et al., 2019):

1. Extract program modules/files/classes such as instances of defect dataset by mining software historical repositories and labeled as defect-proneness or not (Tianchi Zhou et al., 2019).
2. Extract features (Tianchi Zhou et al., 2019) that are related to software defects by analyzing software code or the development process, then these features (e.g., Halstead features, McCabe features, and CK features (Jayanthi.R et al., 2017), and semantic features from ASTs (Guisheng Fan et al., 2019)) are used to measure the defect-proneness of each instance. The non-defect and defect prone modules are represented by green and red, respectively.
3. Construct defect prediction model by training the ML algorithms (Le Hoang Son et al., 2019) for example, Naive Bayes, Support vector machine, Random forest, and Logistic regression, using the set of training instances with their corresponding features.
4. Use the defect prediction (i.e. trained) model to predict the unlabeled program instances.

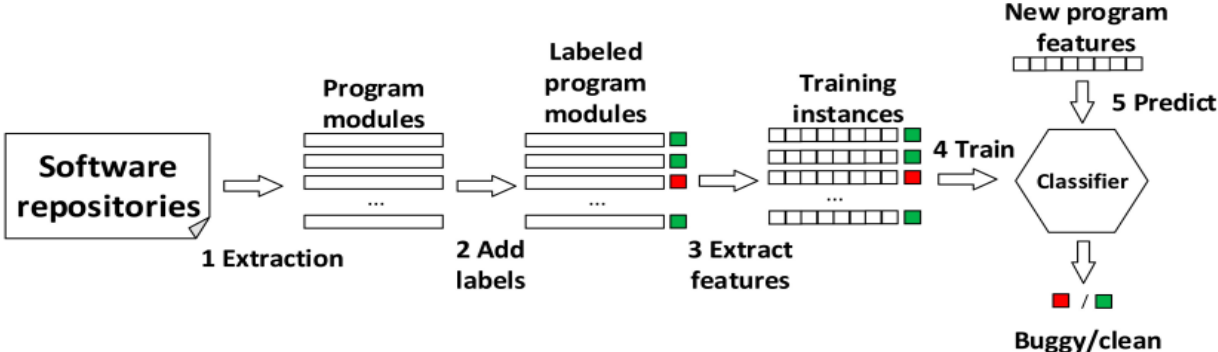


Figure 1.1: A typical software defect prediction process *(Cong Pan et al., 2019)*

SDP studies have taken one of two research directions (Jayanthi. R et al., 2017): one is creating new features or using different combinations of traditional hand-crafted features for better defect data characterization, and the other is utilizing existing ML models and making improvements to existing models for better classification of buggy code. Alongside the first direction, researchers

have designed various hand-crafted features to capture defect characteristics (Dario Di Nucci et al., 2018), for example, Halstead features based on operator and operand counts, McCabe features based on dependencies, and other comprehensive feature sets including Chidamber and Kemerer (CK) features object-oriented design metrics (OOM), and code change features. As for the second direction, many machine learning models have been designed to target software defect prediction, including the decision tree (Tianchi Zhou et al., 2019), random forest (Guanjun Lin, 2020), logistic regression (Cong Pan et al., 2019), and deep learning models (Ming Wen et al., 2017).

Unlike natural languages, software programs (Jian Li et al., 2018) have a well-defined syntax, which can be further organized into Abstract Syntax Trees (AST). ASTs have been widely used for characterizing source code contextual information. However program semantics (Ziyi Cai et al., 2017) is also buried deep in ASTs, it is ignored by traditional hand-crafted features. Researchers have shown the usefulness of ASTs in software engineering tasks— defect prediction (Miltiadis Allmanis et al., 2018; Guisheng Fan et al., 2019), and code completion (Jian Li et al., 2018). In recent, some approaches (Nivetha. R, Kavitha.S, 2019; Shi Meilong et al., 2020) have begun to extract structural and semantic features from ASTs of source code for software defect prediction task and reported improvement of performance over typical defect prediction using traditional hand-crafted features

DL methods have emerged as a powerful technique for automated feature extraction, since DL architecture can effectively capture highly complicated nonlinear features (Rudolf Ferenc et al., 2020). To make use of DL's powerful feature extraction capability, the state-of-the-art method leveraging, convolutional neural network (CNN) (Jian Li et al., 2018), and recurrent neural network (RNN) (Hongliang Liang et al., 2019) was proposed to learn semantics from source code ASTs, which have been shown to outperform traditional feature-based approaches in software defect prediction (Cong Pan et al., 2019). The defect prediction models have been build using different types of attributes/features of software program modules/files/classes, which are detailed in section below.

1.1.1.1 Software metric-based attributes/features

Static code metrics are metrics that can be extracted directly from the source code modules (Jayanthi. R et al., 2017; Le Hoang Son et al., 2019), such as the total number of lines of code (LOC), Halstead metrics, Henry-Kafura metrics, and Cyclomatic complexity metrics. One such

static code metric is the Source Lines of Code (SLOC) which represents a related family of metrics focusing on counting lines in a source code file. Among others, the SLOC family includes the total number of blank lines (BLOC), the total number of lines in a file (LOC), the total number of commented lines of code (CLOC), and the total number of the logical lines of source code (SLOC-L) which is the total number of executable lines of code. The Cyclomatic Complexity (CCN) also known as the McCabe metric or the McCabe Cyclomatic Complexity (Jayanthi.R et al., 2017) is a measure of the complexity of the decision structure in the control flow graph of the program module's code, and equal to the number of linearly independent paths in the control flow graph. The object-oriented software metrics relevant for defect prediction that are defined by (Zhou Xu et al., 2019), and summarized as Table *1.1*.

Table 1.1: Object-Oriented software metrics *(Zhou Xu et al., 2019)*

Object-Oriented Metrics (OOM)	Definition of Metric's quantitative measurements
Weighted Method per Class (WMC)	is the number of total methods in the class
Depth of Inheritance Tree (DIT)	which measures the inheritance levels from the top of the hierarchy of objects for each class
Number Of Children (NOC)	measures the number of the class's immediate descendants
Coupling Between Objects (CBO)	number of classes '*coupled*' to given class, The coupling can be achieved through method calls, field accesses, inheritance, method arguments, return types and exceptions
Response For a Class (RFC)	is the number of distinct methods and constructors invoked by a class
Lack of Cohesion Of Methods (LCOM)	if ' m ' is the total number of *methods* in a class and ' a ' is the total number of *attributes* in the class, then $LCOM$ is computed as follows: $\mathrm{LCOM} = 1 - (\mathrm{sum(am)}/\mathrm{a} * \mathrm{m})$ where ' am' is the number of methods accessing a particular attribute and ' $sum(am)$' is the total sum of ' am'' over all the instances in the class
Number of Public Methods (NPM)	which is a metric that counts all the methods in a class that are declared as public

Data Access Metric (DAM)	measures the ratio of the number of private attributes to the total number of attributes declared in the class
Measure Of Aggregation (MOA)	measures the extent of the part whole relationship (aggregation, composition), realized by using attributes, and the metric is a count of the number of class fields whose types are user defined classes
Measure of Functional Abstraction (MFA)	is the ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods of the class
Cohesion Among Methods of Class (CAM)	computes the relatedness among methods of a class based upon the parameter list of the methods, and the metric is computed using the summation of a number of different types of method parameters in every method, divided by a multiplication of number of different method parameter types in whole class and number of methods

The software metrics i.e. static source code attributes including McCabe and Code Churn metrics are numerical indicators of the quality of software products from different points of viewand the McCabe metrics are software complexity metrics (Zhou Xu et al., 2019). The defect prediction models using these metrics build on the assumption that complex code is difficult for a practitioner to comprehend and consequently hard to maintain and test (Jayanthi. R et al., 2017). Although these metrics have a long history in measuring software quality and are used as indicators (properties of defective instances) for defect classification, they are not direct indicators of defect prediction. For example, these metrics do not provide contextual information of program modules for defect analysis (Cong Pan et al., 2019). Hence, many studies that exclusively relied on software metrics as features to build ML models reported poor results in defect prediction (Mustafa Hammad et al.).From the perspective of how a practitioner performs code auditing, factors such as the complexity of the code can be taken into consideration for further checking, but these factors will not be the decisive ones for a practitioner to determine whether a piece of code is buggy. Hence, other feature sets that can better depict the characteristics of vulnerable code snippets are needed to train more effective ML classifiers to facilitate software defect prediction.

1.1.1.2 Program context-based features

The prior section focused on emphasizing the applicability and benefits of static code metrics for software defect prediction. However, certain characteristics of source code modules associated with structural information and semantic information are not captured by traditional hand-crafted features, and challenge building accurate defect prediction models (Jian Li et al., 2018). Artificial feature-based defect prediction models have been developed using program context information like semantic and synthetic features from AST representations of program modules by leveraging deep learning models (Miltiadis Allmanis et al., 2018; Simone Bonechi et al., 2019).

1.1.2 Deep Learning-based software defect prediction methods

In contrast to more conventional machine learning and feature engineering algorithms like the DL methods has the advantage of potentially providing a solution to address the data analysis and learning problems found in massive volumes of input data (Maryam M Najafabadi, 2016; Le Hoang Son et al., 2019). This makes it a valuable learning tool for software program data analytics, which involves data analysis from very large collections of raw data that is generally supervised and unsupervised data. Supervised learning concerns learning from examples with the known outcomes for each of the training samples (Haonan Tong et al., 2018), while unsupervised learning tries to learn from data without a known outcome (Min Zhang, 2020).

The hierarchical learning and extraction of different levels of complex, data abstractions in DL provide a certain degree of simplification for defect prediction tasks (Hoa Khanh Dam et al., 2018), especially for analyzing massive volumes of data (Rudolf Ferenc et al., 2020), information retrieval (Zhong Zhang and Donghong Li, 2018), and discriminative tasks (Bin Liu et al., 2019; Nivetha. R, Kavitha.S, 2019; Hongliang Liang et al., 2019) such a classification and prediction. Supervised learning is at the same time called classification. It is the classification problem that this thesis focuses on since based on the DL-extracted features, the ML algorithms predict whether a file is a buggy or not. There is a variety of DL models for deep feature extraction as well as classification which includes MLP (Bin Liu et al., 2019), RNN (Guisheng Fan et al., 2019), encoder-decoder (auto-encoders) (Haonan Tong et al., 2018), CNN (Cong Pan et al., 2019), with different classes of Linear Classifiers (Le Hoang Son et al., 2019) including Logical Regression, Naive Bayes Classifier, Perceptron and Support Vector Machine.

The Full Connected Network (FCN)

The FCN (Brownlee, 2018) is a neural network architecture that contains: one input layer with neurons equal to the dimensionality of the input feature vector, the hidden layer(s) that receives input from the previous layer and produce output for the next layer, and a neuron in this layer takes a weighted sum of its input values from the neurons of the preceding layer. A non-linear activation function (Guanjun Lin, 2020; Kang, 2017) within the neuron is then applied to the weighted sum to produce an output that will be propagated to neurons of the subsequent layer as presented in Figure1. 2. This type of neural network is also called MLP (Ming Wen et al., 2017).

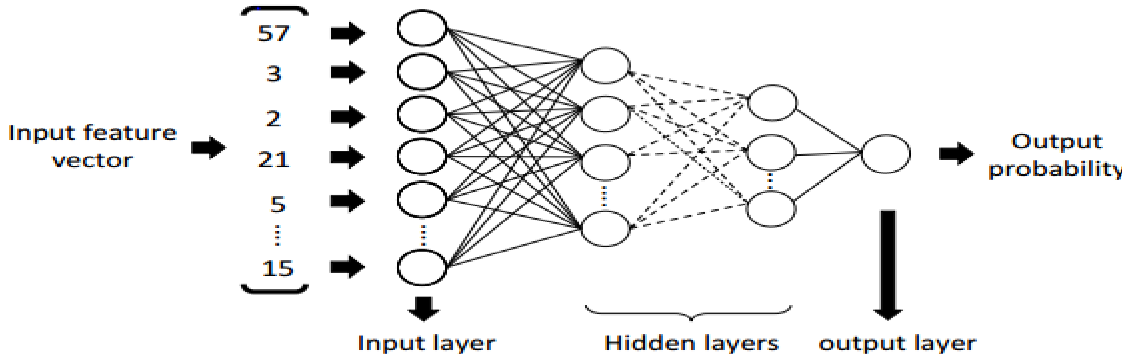


Figure 1.2 overview of multilayer perceptron with one hidden layer (Guanjun Lin, 2020).

Suppose that the input layer has *n* neurons and the output of the i^{-th} neuron in the subsequent hidden layer can be calculated by the following formula:

$$yi = f(\sum_{i=1}^{n} Wni.xi + b), \qquad (1.1)$$

Where xi and yi are the input and output of the neuron Wni is the weight matrix between the n^{-th} neuron in the previous layer and the current neuron, and b is a bias term. Particularly, the $f(\cdot)$ is a non-linear activation function (e.g. the Sigmoid function, Hyperbolic Tangent, Rectified Linear Unit) which brings the non-linearity to the output (Bin Liu et al., 2019). With a cascade of neurons in each layer, the application of non-linear activation functions of each neuron will result in high non-linearity of the original input (Aston Zhang et al., 2019).

During the network training phase, a loss function $L(\widetilde{\gamma}, \gamma)$ is used to minimize between the actual label of γ and the generated output$\widetilde{\gamma}$. The modern networks (Brownlee, 2018; Aston Zhang et al., 2019; Guanjun Lin, 2020) use the backpropagation rule to propagate the loss backward to adjust the weights of each neuron and the weight-adjusting process is accomplished by iteratively

computing the gradients—stochastic gradient descent (SGD) using mini-batches is the common method for training neural networks and applied by most reviewed studies—that splits the training samples into small batches and the gradient is calculated and weights are updated after processing the samples in a small batch.

Compared with conventional ML algorithms such as random forest (Ashima Kukkar et al., 2019; Tianchi Zhou et al., 2019), and support vector machine (Jayanthi.R et al., 2017), the FCN (Aston Zhang et al., 2019; Bin Liu et al., 2019) can fit the patterns that are highly non-linear and abstract. The FCN has the potential of learning richer models than conventional ML algorithms given a large dataset. This potential has motivated researchers to use it for modeling the vulnerable code patterns which are latent and complex (Brownlee, 2018; Zhou Xu et al., 2019). Another advantage of the FCN is that it is "*input structure agnostic*" (Guanjun Lin, 2020), which means that the network can take any forms of input data—e.g. textual and numeric data (Zhou Xu et al., 2019), sequences (Zhou Xu et al., 2019; Ming Wen et al., 2017)—it also offers researchers the flexibility to handcraft various types of features for the network to learn from. Hence, FCN has been successfully and widely adopted in existing learning task-based studies focused on deep feature extraction from complex input features set such as click-through prediction (Bin Liu et al., 2019), defect prediction (Zhou Xu et al., 2019).

The Recurrent neural networks (RNN)

The RNN model has been commonly used and adopted in sequence learning tasks-related studies like machine translation speech recognition (Maryam M Najafabadi et al., 2015), sentence sentiment analysis (Song Wang et al., 2016), and software engineering tasks such as code completion (Jian Li et al., 2018), program synthesis (Yili et al., 2019), program analysis (Okutan, Ahmet, 2018). The RNN network neurons (i.e. units) (Hongliang Liang et al., 2019) have edges (the recurrent edges) that connect adjacent time steps to form cycles/sequences. For a node with the recurrent edge at a given time step t, its output yt not only depends on the current input xt, but also on the hidden node value ht-1 which is from the previous time step (Guisheng Fan et al., 2019; Aston Zhang et al., 2019). This feature enables the RNN to retain information in the past for the current prediction, which makes the RNN suitable for handling sequentially dependent data (Maryam M Najafabadi et al., 2015). Nevertheless, training RNNs can be challenging due to the problems of vanishing and exploding gradients encountered when the loss is back-propagated across many time steps (Ming Wen et al., 2017).

The LSTM network

To overcome the difficulties during the RNN training (Guisheng Fan et al., 2019; Aston Zhang et al., 2019; Guanjun Lin, 2020), the LSTM network which is a variant of RNN was introduced, equipping with a gate mechanism and memory cells, capable of processing long-term sequential interactions for input sequences. Unlike the traditional RNN (Ming Wen et al., 2017) as depicted in Figure 1.3 (a), the LSTM variant introduces a new structure called a memory cell as shown in Figure 1.3 (b). An LSTM memory cell comprises four major components (Guanjun Lin, 2020): a self-recurrent neuron, an input gate, a forget gate, and an output gate. Each cell stores its own state, which is denoted as CSt. This design determines what information in the sequence to store, forget, and output, which allows keeping the long-term temporal contextual information for the sequence. In this way, salient temporal patterns can be remembered over arbitrarily long sequences. According to an up-to-date design of the LSTM structure (Nivetha.R, Kavitha.S,, 2019), an input xt to an LSTM node at the current time step t can be formulated by the following transition equations:

$$it = \sigma W^{i}ixt + U^{i}iht - 1 + b^{i};$$

$$it = \sigma W^{i}xt + U^{i}ht - 1 + b^{i};$$

$$ft = \sigma W^{f}xt + U^{f}ht - 1 + b^{f};$$

$$ot = \sigma W^{o}xt + U^{o}ht - 1 + b^{o});$$

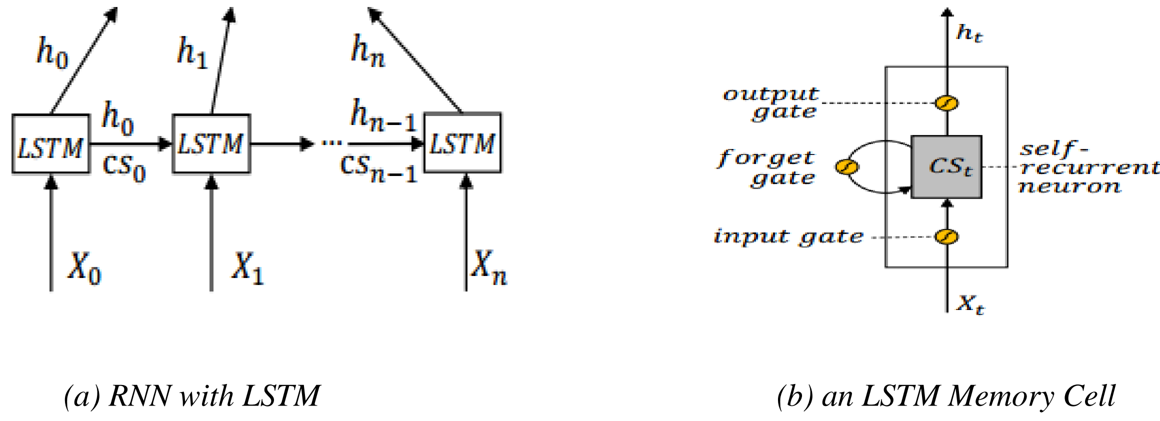
$$ut = \tanh(W^{u}xt + U^{u}ht - 1 + b^{u});$$

$$CSt = it \odot ut + ft \odot CSt - 1;$$

$$ht = ot \odot tanh(CSt); \qquad \textbf{(1.2)}$$

Where it,ft , and ot denote input, forget and output gates, respectively. There are four input weightsW^{u},W^{i}, W^{f} and W^{o}, corresponding to the unit input, the input gate, forget and output gates, respectively. There are also four recurrent weightsU^{u},U^{i}, U^{f} and U^{o}, and four bias terms b^{u}, b^{i}, b^{f} andb^{o}, respectively. The'CSt 'is a built-in memory cell of an LSTM node, which can maintain its internal state over multiple time steps; ht is a hidden state and ut is an input node. The weight matrix W is the weight between the input and the current hidden layer, and U is the weight between the current and previous hidden layer. The symbol $\boldsymbol{\sigma}$, and tanh represent the non-

linear sigmoid function and Hyperbolic Tangent function, respectively. The Θ signifies element-wise multiplication.



(a) RNN with LSTM *(b) an LSTM Memory Cell*

Figure 1.3 LSTM and overview of RNN with LSTM

The gate mechanism *(Guanjun Lin, 2020)* and therefore the memory cell work corporately throughout the coaching section to by selection memorize/forget the knowledge in order that the knowledge obtained within the preceding sequence may be optionally unbroken for access once process the succeeding sequences that permits the network to find out long-range dependencies (Nivetha. R, Kavitha.S, 2019). Consistent with (Hongliang Liang et al., 2019), the interior state of the memory cell is controlled by 3 gates of Associate in Nursing LSTM node. The gates output worth between zero and one and therefore the value of zero denote ‘completely take away this" whereas the worth of one represents ‘completely keep this". From the above equation within the in particular transition *equation 1.2,* one will see that the interior state maintained by the memory cell depends on the element-wise multiplication between it that is that the worth of input gate and therefore the worth of the input node being ut, and the element-wise multiplication between foot that is that the worth of forgetting gate and therefore the previous internal state ct-1. Therefore, a zero worth of the input gate suggests that fully forget the present time step input, and equally, a zero worth of the forget gate suggests that fully forget the previous internal state.

The bi-directional from of RNN

An RNN network can capture the dependencies of a i^{-th} token xi given the preceding tokens such as *xi*-4: *xi*-1. For many natural language processing tasks (Maryam M Najafabadi et al. , 2015), checking the information from previous tokens of a sequence is unable to perform an accurate prediction, the subsequent tokens such as *xi*+1: *xi*+4 of token *xi* can also be useful. Hence, to

capture the long-term dependencies of surrounding tokens of token xi, the bi-directional RNN (e.g. the Bi-LSTM (Nivetha.R, Kavitha.S,, 2019)) structure is designed to serve this purpose. A Bi-RNN network consists of two different RNN networks. Given an input sequence $x1$: xn, the two RNN networks will be generated. One of the RNN networks is fed to the input sequence: $x1$: xn, and the other takes the reverse input sequence: xn: $x1$. The forward and backward states assemble a complete state representation to help capture then token dependencies of a sequence from both directions. Given the current time step t, let $\overrightarrow{ht}$ and $\overleftarrow{ht}$ be the value of the hidden layers in the forward and reverse directions, respectively, the output of a node in the Bi-RNN network can be calculated by:

$$ht' = \mathcal{Z}\left(\overrightarrow{ht},\ \overleftarrow{ht}\right), \qquad (1.3)$$

where function $\mathcal{Z}$ can be a concatenating, summation, averaging or multiplication function.

Being able to capture code contextual information is crucial for bug detection (Miltiadis Allmanis et al., 2018), since the structure of source code is logical and semantic, being coherent and closely connected. Hence, the occurrence of a vulnerable code snippet can be usually associated with either preceding or succeeding code, or even with both. The LSTM network is adopted in this study, particularly the Bi-LSTM which is capable to learn a long-term dependency of both previous and subsequent context and facilitate the detection of defective code patterns.

1.2 Problem Statement

The increasing complexity of modern software has elevated the importance of software reliability but building a reliable software requires a considerable amount of testing and debugging (Cong Pan et al., 2019). Since, both budget and time are limited, these efforts must be prioritized for efficiency (Alyahya, 2020). Additionally, with the extreme growth in cloud (i.e. distributary connected) applications need, increased exposure of modern software applications to the internet and the impact of successful data breaches (Goeschel, 2019), and software failure due to the execution of un-reliable i.e. defective/vulnerable programs, improving the reliability of software being produced is imperative.

The classification and detection of software defects called Defect Prediction early in the development lifecycle is less costly to correct than post development phases (Homona Jacob & Geetha Raju, 2017; Yili et al., 2019; Alyahya, 2020). Defect prediction models can assist software practitioners—e.g. the developers, testing and quality assurance team—in prioritizing their testing

and debugging efforts (Alegre, 2017; Le Hoang Son et al., 2019). A reliable DP model is required to distinguish defective modules from non-defective ones, and can alert quality vulnerabilities of a software application; however, they present developers and analysts with a high rate of false positives and unactionable alerts (Jayanthi.R et al., 2017; Hoa Khanh Dam et al., 2018; Zhou Xu et al., 2019).

Additionally, ambiguity in prioritization schemas make the task of determining which alerts to address first confusing for developers and analysts (Goeschel, 2019; Guisheng Fan et al., 2019; Le Hoang Son et al., 2019). Valuable time and effort is wasted when analyzing irrelevant alerts (Hoa Khanh Dam et al., 2020; Guanjun Lin, 2020). This problem may lead to the loss of confidence in the scanning and prediction models, possibly resulting in the prediction models less practical in daily activity of software developers and analysts (Zhou Xu et al., 2019). The discontinued use of these unreliable models may increase the likelihood of software defects/bugs being released into production; resulting in the compromise of one or several deep learning capabilities such as feature extraction, transformation, and combination.

The classification and assessment of defects/faults/bugs in a source code is commonly called Software Defect Prediction. The introduction of bugs into a software application can resulting in the source code for unexpected execution—e.g. failure to compile or unintended behavior—or the instability or un-usability of the application (Goeschel, 2019). Thus, early efforts in the SDP domain were focused on designing new or source code metric features combination to distinguish buggy modules form non-buggy ones (Song Wang et al., 2016). Source code metric features (i.e. hand-crafted features) can enhance the reliability of DP models by specifically capturing the characteristics/properties of defective module. DL models were expanded to extracted discriminative features representation of source code metric (Rudolf Ferenc et al., 2020), referred as code metrics-based defect prediction. However, source code metrics-based defect prediction methods presents challenges including high false positive rates (Dario Di Nucci et al., 2018; Hoa Khanh Dam et al., 2018; Zhou Xu et al., 2019), missing contextual information of source code—e.g. semantic features—for improving DP model performance (Michele Tufano et al., 2018).

Citing these problems, several research efforts have been made to extract semantic features from the source code context representations such as ASTs, and control flow graphs. The source code context-related information's i.e. semantic features (Hongliang Liang et al., 2019), and syntactic

features as contextual information (Cong Pan et al., 2019) have been used to build/train a reliable defect prediction model, referred as semantic features-based defect prediction method. These efforts result in the overall reduction of false positive rates, but presented challenges including high false positive rate and low prediction accuracy due to the misclassification of infrequent/rare code to buggy ones.

Overall, source code metric and semantic features have different characteristics and are usually adopted to implement a particular attribute of source codes, e.g. line of code, and structure and semantics respectively (Le Hoang Son et al., 2019); either of the attributes may become the same from the buggy and non-buggy files, leading to misclassification of files. Consequently, the defect predictor can be tailored to source code metrics or semantic features (Dario Di Nucci et al., 2018).

Recently, DL models were expanded to extract and combined source code metric and semantic features for building a highly accurate defect prediction model (Cong Pan et al., 2019), called as hybrid features-based SDP. DL-based features combination schemes for the prediction of defects have shown success (Jian Li et al., 2018; Shaojian Qiu et al., 2019; Nivetha.R, Kavitha.S, 2019); however, the work is limited due to: the lack of software defect dataset with both source code metric and semantic information—called *Combined Defect Data* (CDD) for SDP tasks, Lack of proper hand-crafted features combining methods, h*igh dimension* of combined features, and lack of method for effective combined features representation learning—e.g. a model—for learning features structure and dimension agnostic representation. In-depth investigation into hybrid feature representation learning for improved defect classification will greatly add to the literature in the supervised DP domain.

1.3 Objective of the Study

1.3.1 General Objective

The main goal of this research effort was to improve the software defect prediction model by proposing and testing a hybrid features representation leveraging deep learning models.

1.3.2 Specific objectives

- ✓ Obtaining Combined Defect Data (CDD) at file-level context.
- ✓ Extracting hand-crafted features from source code metrics and context representations as hand-crafted features combination (HFC).
- ✓ Extracting semantic features (SF) from AST representations by language model

- ✓ Learning hybrid features representation (HybridF) from HF and SF via the proposed deep learning model
- ✓ Building and evaluating defect prediction model using HybridF.

1.4 Research Questions

- ❖ RQ_1: How much source code files of java software systems are utilized for supervised defect prediction tasks?
- ❖ RQ_2: How the proposed defect prediction approach improves defect prediction performance compared to baseline approaches?
- ❖ RQ_3: How does LHFR perform, compared to the state-of-the-art defect prediction methods using combined features?
- ❖ RQ_4: How is the prediction performance of SDP-CMSF under different parameter settings?

1.5 Thesis Contribution

To alleviate the problems mentioned in Section 1.2, we conduct a series of research to develop a new software defect prediction method in the scenario where there are limited matched and labeled defect data available. Specifically, the major contributions of this thesis summarized as follows:

- ❖ We propose a Bi-LSTM network for file-level defect prediction, which offers a software context-based feature extraction capability, facilitating the discriminative features representation of defective source code. The proposed network is capable of effectively extracting deep source code context representations i.e. Abstract Syntax Tree (AST) that capture the structural and semantic information of functions and reflects the buggy programming patterns. The extracted representations support the word embedding across AST node types, allowing the unlabeled sequence data from similar source software projects to be utilized for learning the buggy patterns to contribute to the learning of limited labeled and infrequent code in the testing project. To illustrate the effectiveness of our proposed framework, we compared the defect prediction performance of our method with other baseline methods, and the results are listed in Chapter 4 Table *4.7*, and Table *4.8*.
- ❖ To further improve the defect prediction performance when there is insufficient training data, we extended our framework to utilize multiple software defects relevant information. This means that the software metric and context information or hybrid features representation learning is more general. In contrast to learning the knowledge from single

software information which is insufficient to predicated defects in the target, learning from multiple defect-relevant information allows the hybrid features representation learning to be performed from a heterogeneous input (i.e. features structure and dimension agonistic representation). To achieve this, we apply multiple networks as our framework components including Bi-LSTM and MLP, each of which will handle data sets that are within-domain and contain data of different structure, domination, and formats. For each network, the outcome of learning is a group of unified representations as to the effective feature sets which can be combined—through an FCN component as the hybrid features set—to boost the prediction performance. The proposed framework can be easily extended to capture multiple data inputs and is able to automatically learn a set of unified feature representations for each data input, these representations can subsequently be used as features and be combined for training classifiers. Empirical studies (please see Chapter 4, *Table 4.9Table 4.10*) showed that by leveraging two different data inputs, the defect detection performance was significantly improved.

- ❖ To enhance the reliability of the defect prediction model in learning combined features of source and capturing the cost associated with the misclassification of buggy files, we trained our defect prediction models by following cost-sensitive defect classification sachem. Since the cost of misclassifying a defective one is not equal to the reverse one and the performance of defect prediction is highly affected by the class imbalance problem presented in our dataset. With respect to the performance of combined features-based defect prediction models under different class weighting sachems as presented in Figure *5.2* showed that by leveraging a cost-sensitive defect classification, the prediction performance was slightly improved in terms of precision, F1_score, and AUC indictors.

1.6 Assumptions, and Research Scope

Even though significant work has been conducted in the area of using source code metric and semantic features for SDP by adopting different deep learning models, to-date there are no authoritative models that can be used to predict fault proneness in large software systems (Ghanathey, 2018). To the specific, software engineers and computer scientists are still experimenting with different deep learning models as well as source code metric and semantic features in a quest to identify the best feature extraction model to use along with a standardized set of features and pre-processed data, so that these can be used to train such defect classifier in order

to yield predictive models. This thesis falls in the area of experimental software engineering and aims to shed light on the problem of identifying such a collection of source code and semantic features that can be used for the generation of predictive models. The thesis, objective is to provide experimental data points based on source code metric and semantic features for software practitioners to use towards developing more accurate defect prediction models, and focus on the analysis of open source systems. The scope of the thesis can be summered as follow:

1. Investigate and report on the effectiveness of file-level defect prediction using different pieces of information such as source code metric, contextual and combined features. To the best of our knowledge the proposed approach is the first to learn from the matched source code files and extract defect data specialized for defect prediction tasks in the following two manners: (I) learn the contextual information i.e. the syntactic and semantic information automatically from the AST representation of the matched source codes, and then (II) the static code metric attributes and extracted deep semantic features responds to a source code file are combined and utilized as combined features for training accurate defect prediction model with static code metric for better software defect classification.
2. Investigate different deep neural networks for feature representation learning and their effectiveness on the performance of the defect prediction model.
3. Investigate four key research questions (RQ1 - RQ4) that can shed light on the effectiveness of the predictive model on different operational scenarios.

As a summary, the scope of this thesis are three-fold: 1) conduct an empirical study on the effectiveness of the obtained CDD for within-project defect prediction task; 2) propose an approach to utilize the various information retained by applying DL models; and 3) propose a combined DL model to learn the most effective features representation for each target system.

The thesis is comprised of two main frameworks: (1) CDDM for modeling file-level defect data which is detailed in section 3.2, and (2) LHFR for DL-based feature representation learning and extraction (detailed in section 3.4.2). Additionally, the investigations and reports (i.e. the analysis, discussion, and explanation of the obtained results) on different approaches of defect prediction: Software metric, contextual, and combined features-based defect prediction approaches (for detail descriptions, please refer to section 2.1.1). However, this thesis does not address whether the models have similar performance to different levels of defects (e.g. high level, medium level, and

low-level defects that depend on defect severity). Furthermore, the thesis does not take into account lower-level source code dependencies or other low-level source code-related information (information extracted from the control flow graph).

1.7 Definition of Terms

In this study, some terminologies regarding software defect prediction are defined as follows:

Abstract Syntax Tree (AST): a representation of source code file as a tree structure.

Base Metric Features (BMF): a predictive features set (i.e. defect data) of all defect instances or samples of PROMISE repository.

Combined Defect Data Modelling: the analytical processes by which a *combined defect dataset* is extracted from both matched instances and files.

Combined Defect Data set (CDD): a data set created with both *metric* and *sequence* data of a *matched* defect instances and files, respectively.

Combined or Hybrid features-based defect prediction: an approach of utilizing both source code metric and semantic features as predictive features for supervised defect prediction tasks.

Deep Learning (DL): a machine learning method—e.g. neural network building a model based upon learning deep patterns in input data—based on the concepts of the human brain.

Defect classification: the binary classification of defect labels**.**

Defect Data: refers to the predictive features set also referred as independent features; whereas **Defect Information** refers to the dependent variables—e.g. binary classes—in supervised defect prediction domain.

Defect Prediction: the defect probability of given sample/class.

DL-generated features: an artificial features extracted with deep learning model automatically.

F-Measure: the harmonic mean of precision and recall measurement.

Hand-crafted Features Combination (HCF): a set of twenty one hand-crafted predictive features created with twenty source code metrics and one *novel* contextual feature i.e. RTF.

Learning Hybrid Features Representation (LHFR): the DL-based feature extraction process by which a common features representation is created from both metric data and sequence data.

Matched File: a java source code file path mapped with, PROMISE defect instance called *Matched Defect Instance*.

Matched Metric Features (MF): the static code metric attributes of matched defect instances.

Metric Data set: a data set created with source code metric attributes of defect instances.

Representative Tokens per File (*RTF*): Number of tokens in AST representation of a java file.

Semantic Features (SF) Extraction: the process by which same features representation is extracted from relatively similar AST tokens (patterns) of the sequence data.

Sequence Data set: a data set created with AST representations of source codes.

Software metric features-based defect prediction: the method of defect prediction by using source code metric features of target software systems.

Software Metric: a quantitative measure used to assess the progress of software; whereas contextual features refer to syntactic and semantic features of a software program.

Software practitioners: the actors of software development process. For example, the developers testers and quality assurance teams.

Software semantic features-based defect prediction: the process of supervised learning for software defect prediction by using source code semantic features alone.

Testing set: a set of instances used to validate the trained model.

Training set: a set of instances used to train a defect prediction model.

1.8 Summary

This Chapter has outlined a brief history of the software defect prediction using source code metric and/or semantic features and its current problems, discussed motivating factors for continued research, and posited the goal of this proposed research effort. Research questions that guided this research were presented. Barriers, limitations, and assumptions were identified. The rest of this paper is organized as follows: Chapter 2 presents a review of literature; Chapter 3 outlines the research methodology that was followed; Chapter 4, outlines the experiments and results analysis; Finally, Chapter 5 presents the discussion, limitation and conclusion of the thesis.

Chapter Two

2 Review of Literature

To date, many studies have been proposed to apply DL approaches for software defect prediction tasks. The underlying assumption is that buggy code snippets contain latent buggy patterns and characteristics which are discoverable by DL models. Among these studies, deep neural networks were also adopted for the extraction of source code metric, and semantic features and had achieved more promising results compared with that achieved using conventional ML models trained in source code metric features set. Furthermore, the deep neural networks which are capable of automated feature learning and transformation can significantly reduce the effort of feature engineering tasks (Bin Liu et al., 2019; Hoa Khanh Dam et al., 2018; Guanjun Lin, 2020). Using a networks such as multi-layered perceptron and neural memory networks, combination of hand-crafted source code metrics and source code semantics can be directly learned and processed for identifying potentially buggy code files (Zhao, et al., 2019; Yili et al., 2019).

We categorize the work applying DL models for defect prediction task into three categories based on different types of predictive features and we provide some background knowledge of each type of predictive features and review studies on each category to reveal how the distinct predictive features benefited the learning of defective code patterns/characteristics. In this chapter, we focus on reviewing the research efforts which utilized both source code metric and semantic features for predicating the defects from a new perspective, pointing out that defect prediction methods based on effective representation of source code metric and/or semantic features by DL models, can be a new research trend which has brought promising results.

This chapter is organized as follows: in Section 2.1 presents the previous efforts on SDP which utilized the DL-generated discriminant features representation of source code metrics for building a discriminative defect prediction models. In Section 2.2, we discuss SDP studies which adopted deep neural networks for the extraction of predictive features—e.g. semantic and/or structural features—from source code context representation. Section 2.3, focuses on the review of the studies that utilized the hybrid/combined features representation of both source code metric and semantic features by leveraging DL models for supervised defect prediction task. By the end of each sub-section, we discuss how the obtained features representation is toiler for the prediction of defects. We also identify the limitations of studies in each category.

2.1 Defect Prediction Using Source Code Metric Features

The source code metric features was chosen by many pioneer researchers as predictive features for defect prediction based on static hand-crafted code metric attributes. This category of works treated static source code metric attributes as representative predictors for learning defective characteristics/properties.

Summary of recent works

Yang et al. (2018) proposed a deep learning model for just-in-time defect prediction, which predicts defect-prone changes. They selected 14 basic change measures regarding code change, and leveraged dynamic baseman network to build a set of expressive features from these basic measures. At last, they used machine learning models for classification. Experiments show that their methods could discover 32.22% more bugs than the state-of-the-art model.

Haonan Tong et al. (2018) proposed a deep learning model and two-stage ensemble learning for defect prediction. They leveraged stacked de-noising auto-encoders to generate effective features from traditional hand-crafted features in the NASA MDP dataset, and used ensemble classifiers for defect prediction. The results showed that deep representations of software metrics are promising for software defect prediction.

Zhou Xu et al. (2019) proposed a deep learning model with hybrid loss function for defect prediction. The researchers described in their study that, a triplet loss is used to learn discriminate feature representations from collection of traditional hand-crafted features in the PROMISE and AEEEM datasets and weighted cross entropy loss is leveraged in order to address class imbalance problem of software defect dataset. The results showed that the merit of distance learning, and cost-sensitive metrics learning for building high accurate defect prediction model.

However, these of the obtained expressive features are still exposed to human-level feature extraction cost and error.

2.2 Defect Prediction Using Semantic Features

In this types of software defect prediction, artificial features such as semantic and synthetic features have been utilized for defect prediction task by leveraging different deep learning models (Cong Pan et al., 2019) (Jian Li et al., 2018) (Guisheng Fan et al., 2019) (Rudolf Ferenc et al., 2020). In 2016, Wang et al. leveraged DBN for software defect prediction. They used selected

AST sequences taken from source codes as input to the DBN model, which generate new expressive features and used machine learning models for classification. Their WPDP and cross-version defect prediction experiments showed that their model outperformed the state-of-the-art machine learning models.

Summary of recent works

Then in 2018, Li et al. had proposed a CNN-based defect prediction model, which leveraged word embedding for encoding AST token vectors into integer vectors, a random oversampling technique in order to handle the class imbalance issue of defect data and CNN model for defect prediction. Although they employed merge operator in Keras in order to combine traditional hand-crafted features and semantic features and used logistic regression instead of various machine learning models for classification, their results outperformed the DBN models of Wang et al. (2016). They also proved that adding traditional features to deep features could further enhance model performance.

In 2018, two papers leveraging RNN for SDP were published. The first paper (Hoa Khanh Dam et al., 2018) used a type of RNN model, the long-short term memory (LSTM) model, to predict defects, which takes AST sequences as the input. The second paper (Michele Tufano et al., 2018) leveraged tree-based LSTM models to predict defects, which takes AST as the input. However, their results were not as good as the results for Li's model (Zhong Zhang and Donghong Li, 2018).

Guisheng Fan et al. in 2019 designed SDP model, which learns syntactic and semantic features automatically according to the context of source code. They, made use of javalange to parse source code into AST tokens, dictionary mapping, and word embedding to encode AST token vectors into integer vectors which are used as input for the Bi-directional RNN model. Although they employed an attention mechanism to extract critical features and build a highly accurate defect prediction model. The results improved the F1-measure by 14% and AUC by 7% compared with state-of-the-art methods.

Cong Panet al. (2019), proposed an improved CNN-based software defect prediction. They followed similar producers like (Cong Pan et al., 2019) while encoding input vectors of CNN model. The embedding layer is built based on skip-gram models, instead of word embedding (Zhong Zhang and Donghong Li, 2018). Although they adopted a dropout mechanism in between dense layers, order to handle over-fitting, and improve the generalizability of the defect prediction

model. The experimental outcomes showed that their method, discover 32.22% more bugs than the state-of-the-art model and improved F-measure, G-measure, and MCC by 6%, 5% and 2%, respectively.

Shaojian Qiu et al. (2019) proposed a CNN-based model for cross-project defect prediction. AST token vectors of source and target projects, are encoded into integral vectors, and inputted into variant CNN models for semantic feature generating. A matching layer, and classic distribution adoption method is adopted in between the source and target model for an effective transferable deep feature and traditional hand-crafted feature extraction. Python's concrete method is adopted to combine both features. The average outcome achieved was 0.532 in terms of F-measure.

Overall, still there is a dreadful requirement of software metric features modelling in order to predict the buggy files accurately.

2.3 Defect prediction using hybrid features

In this types of defect prediction, traditional hand-crafted features and deep features including semantic and synthetic features have been utilized leveraging different deep learning models and machine learning algorithms for software defect prediction task. The most related works can be summarized as follow:

Table 2.1: Summary of the recent, relevant and related works

Adopted methods/tools	Gap/Limitation
By leveraging a **CNN model** for learning source code semantic features, (*Jian Li et al., 2018; Shaojian Qiu et al., 2019*) had proposed a combined features-based SDP method. Both of them used ***javalang tool*** in order to parse(encode) source code into AST representation, but a ***Marge operator*** in Jian Li et al. (2018) and ***concrete method*** in Shaojian Qiu et al. (2019) is employed to combine hand-crafted code metric features and CNN-generated semantic features for batter defect prediction performance. The obtained results shows that, the false positive rate was decreased.	While the false positive rate was decreased, the prediction accuracy was not improved.

(Shi Meilong et al., 2020) had proposed a SDP model using both structural and semantic features form source code representation by adopting CNN. The utilized tools/methods and obtained results of this study are summarized as: the '*DependencyFinder'* API had used to parse the compiled source files (.zip or .jar extension) and extract their relationships using a tool developed by ourselves, and further features embedding learning is done by using the Node2vec method. Finally, the authors had concatenated the semantic feature vectors with structural feature vectors via Merge operator in Keras. The obtained results **improved the precision measure.**	While these studies had limited to provide experimental report regarding other performance indicators or measures—e.g. the recall, f1-masure and others—the reported contribution of source code pattern i.e. disorder of feature measure with software metric sets in an average of 4.81% absolute and 11.57% relative improvement not only verified the visibility of our study, but motivated us to validate the feasibility of our approach through reviewing the analysis of defect prediction results using source code semantic features when compared to metric sets.
(Nivetha. R, Kavitha.S, 2019) had proposed a combined features-based Bi-RNN model to learn the common patterns (or) regularities of source code for batter defect prediction performance The utilized tools/methods and obtained results of this study are summarized as: The ***Dynamic Link Library*** were used to find the nodes of the python-based software components; While building the proposed model; **LSTM algorithm** is adopted—that enables to learn deep semantic characteristics—at the same time, source codes are extracted from software repository and the components are measured using **software metric sets.** Finally, the obtained semantic features and software metric sets are **combined** to measure the code and find the defect. The obtained results **improved the precision measure.**	
(Guanjun Lin, 2020) had proposed a software context-based SDP methods by leveraging RNN model. The	However, the authors had identified that, the source code metrics-based

adopted tools/methods in this study and reported results analysis can be summarized as: **Depth First Transversal** (DFT) and **Word2Vec model** had employed to preserve the syntactical structure and semantic properties of source code, respectively; A **Bi-RNN** with LSTM cells were adopted to capture structural information and long dependencies among codes. Based on their analysis of defect prediction models trained in source code metric sets and semantic features, they reported that, the software contextual information (i.e. semantic features set) is significant for defect/fault/bug detection.	defect prediction model had outperformed the proposed semantic features-based model; suggested a SDP approach that combines both source code metric and semantic features for defect prediction tasks.

Chapter Three

3 Methodology

In this chapter, we present a DL approach for SDP targeted on multiple sources of defect-relevant information in a given software defect dataset. The aforementioned SDP method aims to utilize source code metric and context-related information's via adopting deep neural networks for effective and automatic learning of defective code property (pattern) using static code attributes and AST representations from the labeled training dataset. The study is conducted along three dimensions, which are detailed and depicted as follow:

The first dimension has to do with the selection of appropriate software systems for SDP tasks and the extraction of code metric suites and source code representations to use in the subsequent dimensions, and which are depicted as Figure 3.1 **Error! Reference source not found.**(a) and (b), respectively.

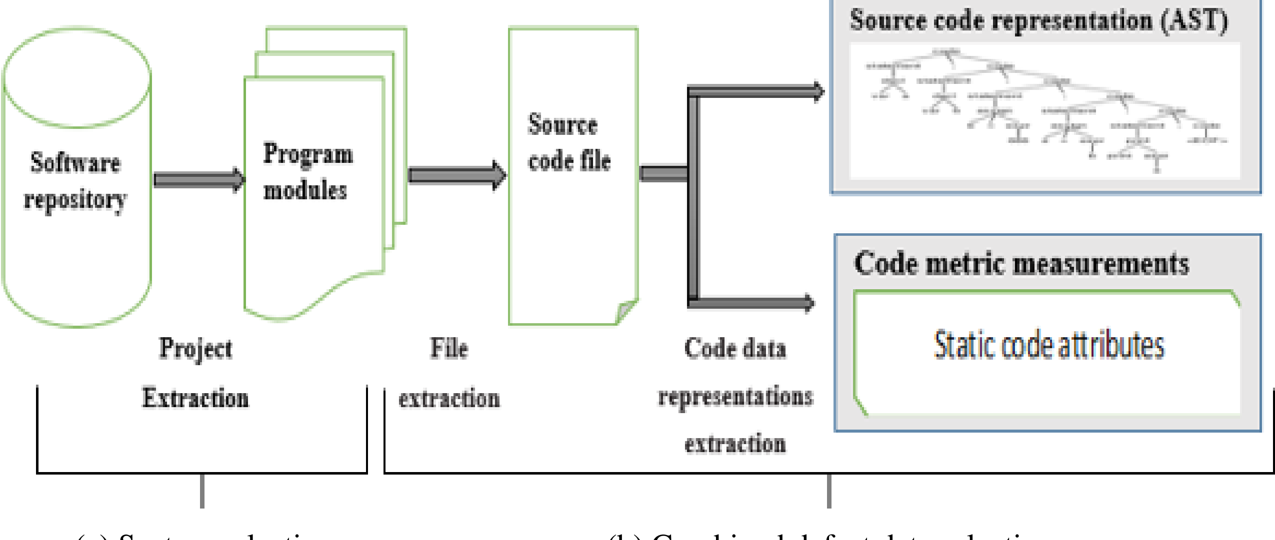


Figure 3.1: The process of software defect dataset preparation (i.e. CDDM)

The second dimension is depicted in Figure *3.2*, which is related to the selection of appropriate data modeling methods i.e. deep learning methods (Zhou Xu et al., 2019). In the case of this study, variant deep neural models are adopted in order to learn high-level features and unified representation using the selected historical labeled software defect data in the preceding dimension (Nivetha. R and Kavitha.S, 2019), and the extracted combined features are used as input for building highly accurate defect predictor in the subsequent dimension.

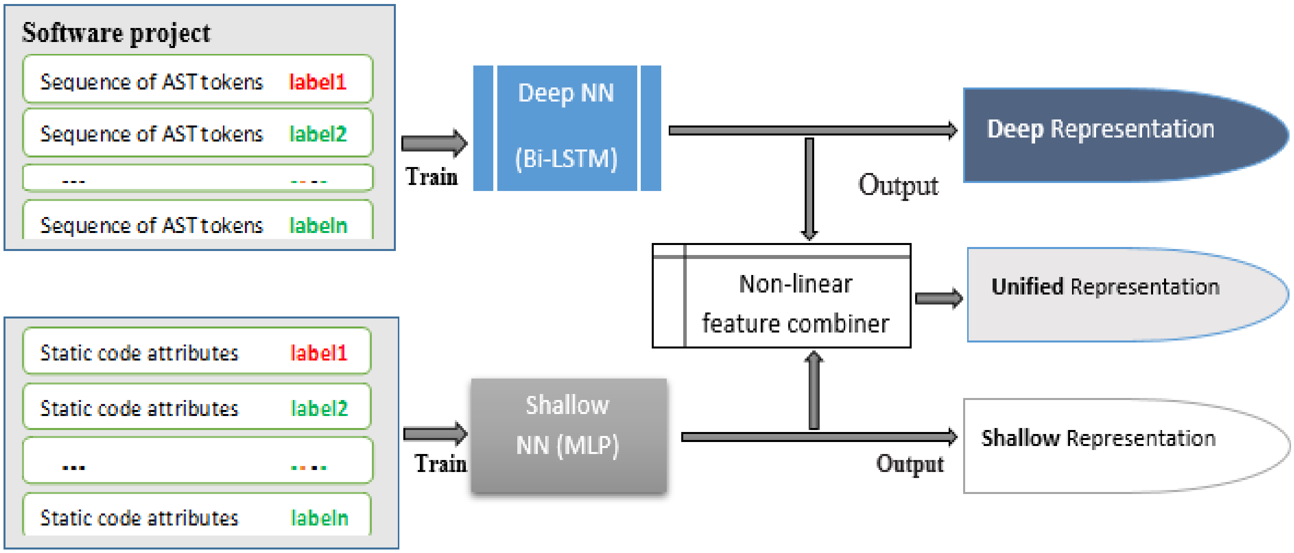
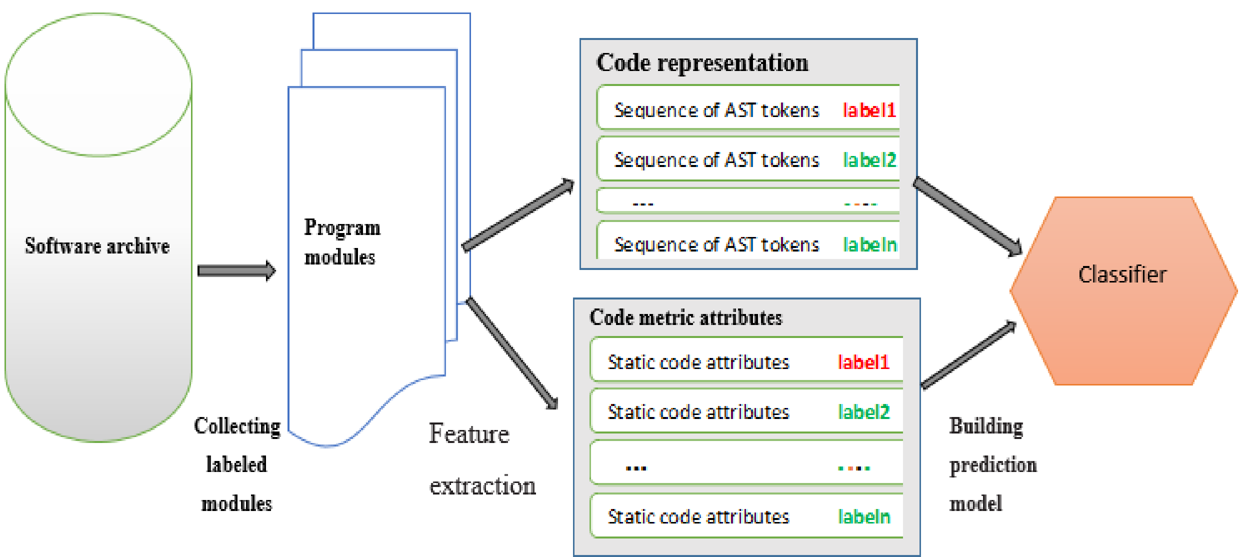
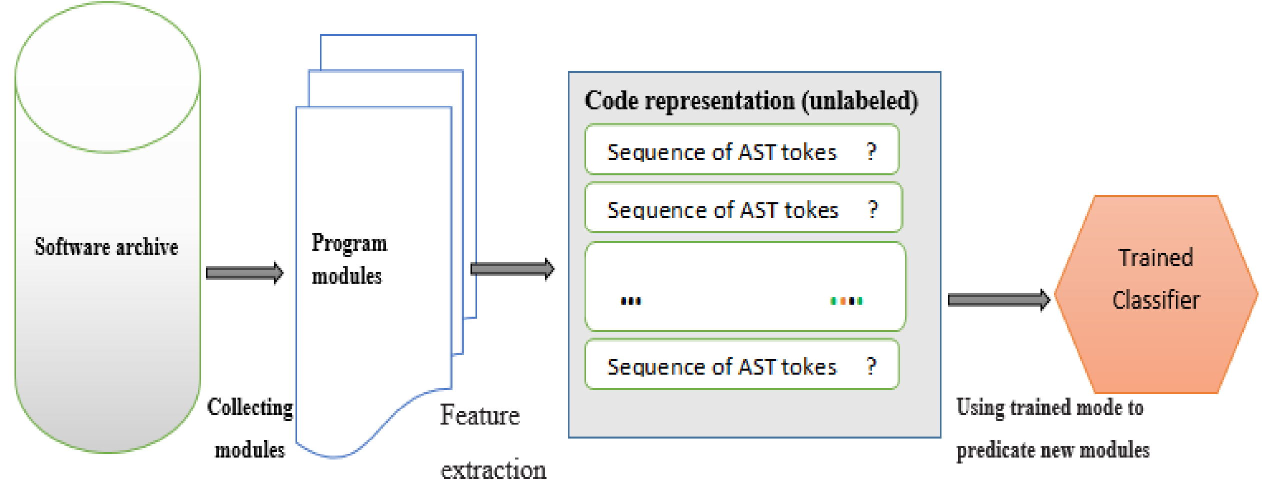


Figure 3.2: Overview of deep neural networks for DL-generated features combination

And the last dimension but not the least one deals with, building a conventional ML model as an efficient predictive model using DL-generated representations (Cong Pan et al., 2019), and evaluations of defect prediction models, respectively and depicted as the following Figure *3.3* Figure *3.2* (a) and (b).



(a) The processes of building defect prediction model using training dataset



(b) Evaluation processes of defect prediction model using testing dataset

Figure 3.3: The processes of SDP using code metric and semantic features

3.1 Approach overview and Dataset collection

This section provides an overview of the proposed DL approach for SDP tasks, by describing a workflow of how source code metric and semantic features are utilized for the defect prediction tasks. Following this, the source code file-level data representations and their relevance for defect prediction tasks, and the process of data collection will be introduced.

3.1.1 Problem formulation

The proposed method takes a list of source code files from a given software system (project) as input and outputs a file ranking list based on the likelihood of the input file is defective. Let ***f*** = {*file1, file2, file3 … filen*} be all program modules in a given software system. The proposed defect prediction method aims to find a file-level software defect predictor **Đ** : ***f*** **ε** [0,1], where 0 means a given file is definitely non-defective, and 1 means a given file is definitely defective, such that Đ(*filei*) measures the probability of *filei* containing defective code(s), and also it suffices to treat Đ(*filei*) as the probability score or likelihood of a *filei* to be defective, called as defect proneness (Mustafa Hammad et al.), so the proposed DL model can classify defect, and provide top risk level files.

3.1.2 Workflow

In this section, we elaborate on the proposed DL approach of using code metric and semantic features for defect prediction tasks, which is abbreviated as SDP-CMSF. In this study, CMSF refers to Code Metric and Semantic Features. Figure *3.3* depicts the overall framework of SDP-CMSF, which consists of three main parts: the first part focuses on the extraction of software defect

data using traditional hand-crafted information's from static code metric suites, and will be detailed in Section 3.3.1.1. The second part focuses on the extraction of source code context-related features such as structural and semantic features using AST representations of program modules. The last but not the least one refers to the utilization of source code metric- and semantic-related features (information) obtained in the first two steps to obtain a unified representation for building software defect predictor and will be detailed in Section 3.3.2.

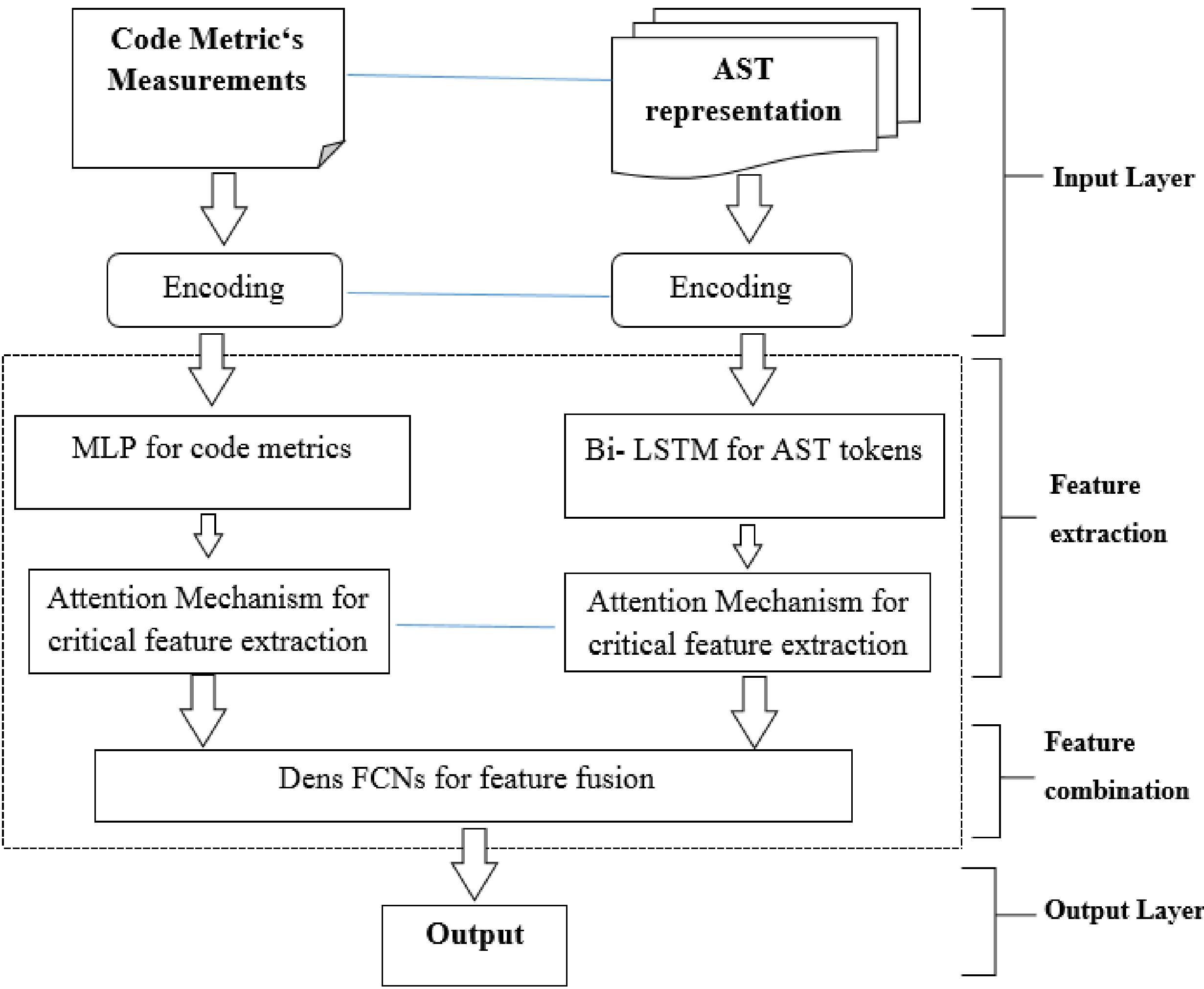


Figure 3.4: Overview of the proposed SDP using source code metric and semantic features

3.1.3 Raw Dataset Collection

The software defect datasets which are used in this thesis are available from the public PROMISE repository, which is an open-source dataset and have been widely used in SDP literature (Jayanthi. R et al., 2017; Guisheng Fan et al., 2019; Cong Pan et al., 2019; Nivetha. R, Kavitha.S, 2019). The datasets used in the experiments of the study are subsets of two original datasets which were, the

PROMISE Source Code (PSC) and the Simplified PSC (SPSC) datasets2, and collected and provided by (Cong Pan et al., 2019). The authors have discussed in their study that, the bug information and the logs are collected and analyzed from the version control system in order to validate whether a commit is a bug fix. The providers (Cong Pan et al., 2019; Shaojian Qiu et al., 2019) of the datasets having discussed that the SPSC dataset is a smaller dataset and that is used in neural network-based studies, while the PSC dataset is a larger dataset designed by hand for defect prediction of a wider, and the dataset included the version numbers, the class name of each file and defect label for each source files, as well as 20 traditional features such as weighted methods per class (WMC), depth of inheritance tree (DIT), and a number of children (NOC).

This study aimed to use both code metrics and AST representations as input, so we have downloaded the corresponding labeled code metrics and versions of the projects from open source repositories. To verify the validity of the study, we selected 12 open-source projects as our evaluation datasets. The source codes and corresponding PROMISE data for all projects are public and have been widely used in SDP research (Jayanthi. R et al., 2017; Zhou Xu et al., 2019; Yili et al., 2019). In our experiments, we extracted deep features from the java source codes (Shaojian Qiu et al., 2019), and adopted the static code metrics and data labels from the PROMISE repository which is collected by (Cong Pan et al., 2019; Shaojian Qiu et al., 2019).

In order to facilitate feature extraction from the source code metrics and AST representations, the PSC dataset providers had set five guidelines for software defect dataset design: (1) the dataset had to come from existing high-quality defect prediction repositories, (2) the dataset had to be based on open source projects i.e. the projects included in the dataset had to contain at least two versions for cross-version comparison, and (4) the link between the open-source project versions and labeled csv files had to be verified to make comparisons with deep features and traditional features easier (Cong Pan et al., 2019; Shaojian Qiu et al., 2019). Table 3.1 shows the static code metrics contained in the PROMISE repository (highlighted in Section 2.1.1.1.). Appendix C. 1 shows the essential information of selected PSC projects (Shaojian Qiu et al., 2019; Cong Pan et al., 2019), including project name, project version, number of files, number of defects and defect rate (i.e. the percentage of defective instances).

Table 3.1: Definition of static code metrics (*Cong Pan et al., 2019; Shaojian Qiu et al., 2019)*

Metrics	**Description**
dit	The maximum distance from a given class to the root of an inheritance tree
Noc	Number of children of a given class in an inheritance tree
Cbo	Number of classes that are coupled to a given class
Rfc	Number of distinct methods invoked by code in a given class
Lcom	Number of method pairs in a class that do not share access to any class attributes
lcom3	Another type of the lcom metric proposed by Henderson–Sellers
Npm	Number of public methods in a given class
Loc	Number of lines of code in a given class
Dam	Ratio of total private/protected attributes to the total number of attributes in a given class
Moa	Number of attributes in a given class that are of user-defined types
Wmc	Number of methods in class
Mfa	Number of methods inherited by a given class divided by the total number of methods that can be accessed by the member methods of the given class
Cam	The ratio of the sum of the number of different parameter types of every method in a given class to the product of the number of methods in the given class and the number of different method parameter types in the whole class
Ic	Number of parent classes that a given class is coupled to
Cbm	Numbers of new or overwritten methods that all inherited methods in a given class are coupled to
Amc	The average size of methods in a given class
Ca	Afferent coupling, which measures the number of classes that depend on a given class
Ce	Efferent coupling, which measures the number of classes that a given class depends on
max_cc	The maximum McCabe's Cyclomatic complexity (**CC**) score of methods in a given class
avg_cc	The arithmetic mean of McCabe's Cyclomatic complexity scores of methods in a given class

3.2 Combined Defect Data Modelling Framework

In order to collect defect data from PROMISE[1] with source code metric information and/or context-related information (e.g. syntactic and semantic information), a Python Module is developed and used in order to query software projects for matched files/instances and extract defect data. Figure *3.5* depicts the overall framework of defect data collection/extraction processes (i.e. python module/tools), which consists of two main parts. The first one focuses on file-level defect data extraction, including code metric- and/or AST representation-based data, and will be detailed in Section 3.2.1. The second one is detailed in section 3.3.1, which is focused on defect data reconciling.

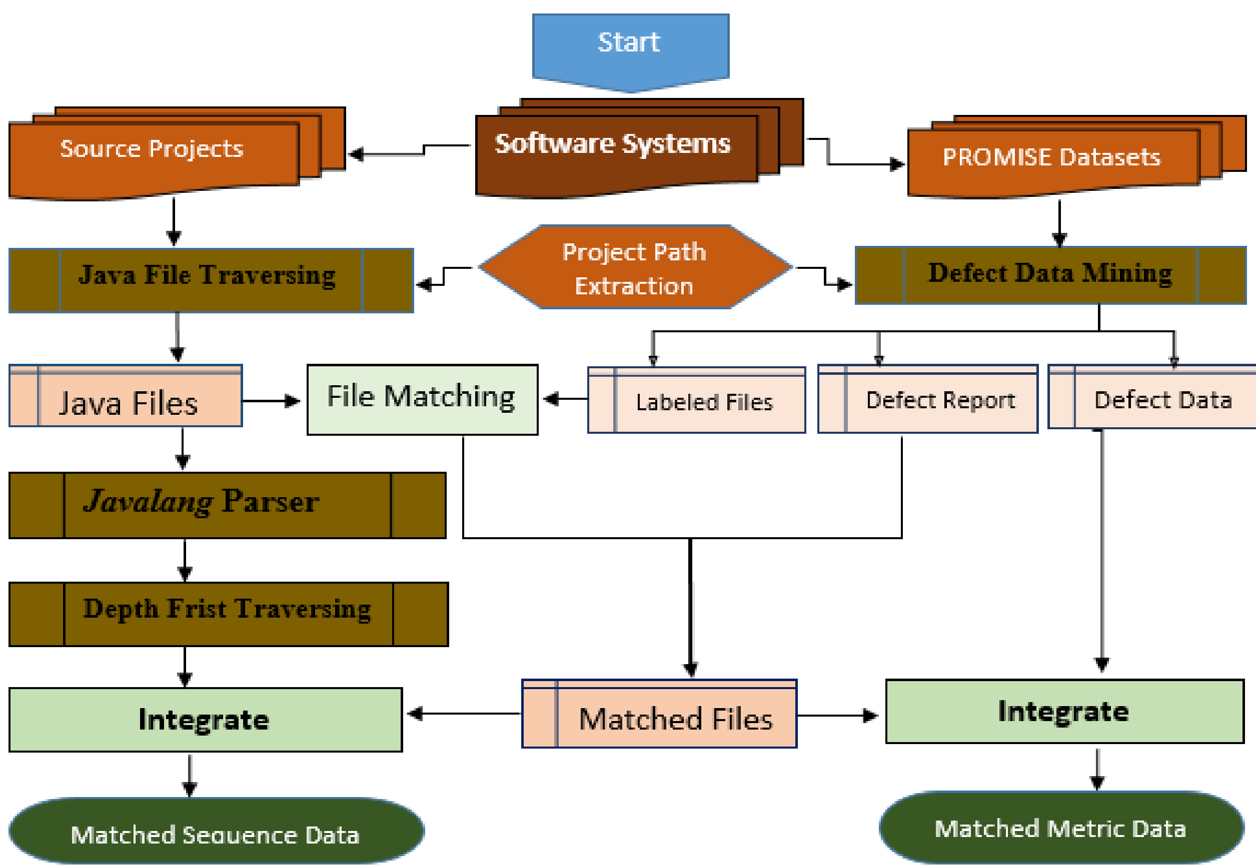


Figure 3.5 Overview of file-level defect data modelling (CHDD) framework

3.2.1 File-level Defect Data Engineering

In order to extract defect data from the source code module containing both software metric and context-related information, a file-level defect data pre-processing module i.e. python program is

[1] PROMISE Repository: - the original website is http://openscience. us/repo; however, the website is currently unavailable, and a backup repository can be found at https://github.com/opensciences/opensciences.github.io

implemented and it contains three main tasks (submodules). The first one is detailed in section 3.2.1.1 and all about this sub-module is to load defect dataset and keep information's like labeled file name, bug report, and static code metric attributes for every sample. The second sub-module is detailed in section 3.2.1.2, and sequence data (i.e. AST representation) extraction is all about this task. All of this sub-module is to transverse every file path of the project that was matched for a '.java' file extension and keep it as 'source code file name', and parsing each java source code file (path) i.e. source code file name' to AST representation. The third task is detailed in section 3.2.1.3, and all about this task is to combine code metric data and sequence data that was matched for a file name. This sub-module ensures the obtained labeled file name matches with the source code file name and combines the obtained defect data (i.e. bug report and static code metric attributes) with its AST representation.

3.2.1.1 Source Code Metric-based Defect Data Extraction

Different types of metric suites have been commonly used in SDP studies (Cesar Couto et al., 2016; Ming Wen et al., 2017; Tianchi Zhou et al., 2019; Zhou Xu et al., 2019), for performing software defect prediction tasks. DL methods have been commonly adopted in existing software metric-based defect prediction methods, in order to learn discriminant feature representation from code metric attributes and/or source code context representations for building a highly accurate defect prediction model (for a detailed explanation, please refer to section 2.2.1). We observe that there are some important code metrics used to predict tasks and they are often considered as traditional metric sets (Zhong Zhang & Donghong Li, 2018; Nivetha. R, Kavitha.S, 2019).

As shown in (Guanjun Lin, 2020), features in static code metrics have proven to beat AST representations in defect classification tasks through capturing robust indicative characteristics of defective modules. Therefore, in this thesis, we choose static code metrics for extracting discriminative features in order to discriminate defective files from non-defective ones. The vector of static code metrics with hand-crafted attributes can be leveraged by the proposed deep neural network (i.e. MLP) for obtaining discriminative feature representations capable of capturing properties of defective files. Below is the pseudo-code that is used to fetch every code metric attributes and bug report from a csv formatted defect dataset that was matched with a file name.

Algorithm 1: Pseudo code for the extraction of Matched Defect Data

```
Input: Project Datasets Pl = {Pp1, Pp2, ..., Ppm}; # Pp refers to Project path
Output: Mapped Files  MF = {Lf1, Lf2, ..., Lfn}; # Lf refers to Labeled file
          : Mapped Dataset  Md = {Mp1, Mp2, ..., Mpn}; # Mp refers to Mapped project
Procedure DefectDataExtraction
   (1) Initialize STRING LabeledFileName;
   (2) Initialize LIST CodeMetricData, ;
   (3) Initialize INTEGER BugReport;
   (4) Initialize  DICTIONARY DefectData, LabledFiles;
   (5) for PorojectPath in Pl do
   (6)       With Open (PorojectPath.csv) as ProjectDataset do
   (7)             for raw in csv.reder(ProjectDataset) do
   (8)                 GET raw[2], raw[3:23], raw[23];
   (9)                 STORE  raw[2]+'.Java' to LabeledFileName
   (10)                STORE   raw[3:23] to CodeMetricData
   (11)                STORE   raw[23] to BugReport
   (12)                CREATE MAP  between BugReport, CodeMetricData
   (13)                STORE   BugReport, CodeMetricData to DefectData
   (14)                CREATE MAP  between LabeledFileName, DefectData
   (15)                STORE   LabeledFileName, DefectData to LabledFiles
   (16)                ADD    LabledFiles to MF
   (17)            end for
   (18)       ADD  MF to Md
   (19) return Md
end Procedure
```

Note: this Module is used to extract file name, bug report and code metric data from matched defect samples

Table 3.2: Sample Data for Defect data with code metric attributes

1	Projct	version	File Name	wmc	dit	noc	cbo	ca	avg_cc	bug
2	ant	1.3	org\apache\tools\ant\taskdefs\ExecuteOn	11	4	2	14	2	1.2727	0
21	ant	1.3	org\apache\tools\ant\BuildLogger	4	1	0	3	2	1	0
102	ant	1.3	org\apache\tools\ant\taskdefs\Replace	19	4	0	8	2	1.2632	1
116	ant	1.3	org\apache\tools\ant\Target	19	1	0	18	16	1.4211	1
122	ant	1.3	org\apache\tools\ant\taskdefs\Javadoc	65	3	0	21	6	1.5231	0
124	ant	1.3	org\apache\tools\ant\IntrospectionHelper	14	1	0	24	20	2.7857	2
125	ant	1.3	org\apache\tools\ant\NoBannerLogger	4	2	0	3	0	1.75	0
126	ant	1.3	org\apache\tools\mail\MailMessage	27	1	0	3	1	1.4444	0
127	**Total**	__	125	__	__	__	__	__	__	__

3.2.1.2 Source Code Context-based Defect Data Extraction

We observe that software defects are often reflected in the syntactical structure and semantic properties of source code (Hongliang Liang et al., 2019; Cong Pan et al., 2019; Shi Meilong et al., 2020), particularly at the java source code file-level. To capture these features and code properties, we follow the early work of (Guisheng Fan et al., 2019).

Since the proposed method uses source codes as input, so suitable code representation was deemed to be beneficial for parsing source code. Code representations (Cong Pan et al., 2019) include character-level, token-level, AST-node-level, tree-level, graph-level, path-level, among others. According to the former research (Cong Pan et al., 2019), AST-node-level code representation is the suitable representation that can reflect the structural and semantic information of programs, also it has proven to beat character-level, token-level, and representations of higher granularities in program classification tasks. Therefore, in this thesis, we choose ASTs for extracting the latent programming patterns indicative of potential bugs. To achieve this, an AST needs to be tokenized/serialized for converting to a vector while preserving its structure and semantics. The vector that holds the structural and semantic information can then be leveraged by the proposed deep neural network (Guanjun Lin, 2020) for obtaining deep feature representations capable of distinguishing buggy files from non-buggy ones.

Below is the pseudo-code that is used to fetch all the AST representations of java source code files from a given project. The program is divided into three parts namely project verification, java source code file extraction, parsing source code file to AST, traversing and selection representative

string tokens of AST called sequence data extraction, and finally exporting the extracted data to a csv formatted file.

In the first part i.e. project verification, we downloaded projects with source code files and defect datasets from a public repository. Then, we specify the data preprocessing module to ensure every software source projects to the existence of its defect dataset before extracting its details (highlighted in Algorithm 2).

Algorithm 2: Pseudo code for verifying the existence of defect dataset for a source project.

Input: Labeled Projects $Pl = \{Pp1, Pp2, \dots, Ppm\}$; # Pp refers to Project path

: Source Projects $Ps = \{p1, p2, \dots, pn\}$; # p refers to Source Project's path

Output: Verified Source Projects VP $= \{p1, p2, \dots, pj\}$; # $j \leq m, n$

Procedure $SourceProjectVerification$

(1) **for** i in Ps **do**

(2) **for** j in Pl ***do***

(3) **If** $Ps[i]$ endswith $Pl[j]$ **then**

(4) **ADD** Ps[i] to VP

(5) **end If**

(6) **end for**

(7) **end for**

(8) return VP;

end Procedure

Table 3.3: Verified Sample software projects with and without Defect Data.

Project	Source Path	Defect Data Path	Configured
ant-1.3	F:\Datasets\SDP_Datasets\SDP_M2S\source file\Source Projects(All)\ant-1.3	F:\Datasets\Labled Data\ant-1.3.csv	Yes
ant-1.5	F:\Datasets\SDP_Datasets\SDP_M2S\source file\Source Projects(All)\ant-1.5	F:\Datasets\Labled Data\ant-1.5.csv	Yes
forrest-0.7	F:\Datasets\SDP_Datasets\SDP_M2S\source file\Source Projects(All)\camel-1.6	**Not Found**	**Not**
xerces-1.3	F:\Datasets\SDP_Datasets\SDP_M2S\source file\Source Projects(All)\xerces-1.3	F:\Datasets\Labled Data\xerces-1.3.csv	Yes
Verified samples to Total (%)	**41/42 =98%**	**41/41 =100%**	**98.81%**

As shown in Algorithm 3 the second part of this module/program, for every verified source project, we look at java source code files that were in a source project and procure the list of java file paths (i.e. defect instance). All of this data, that is, java source code file paths are stored in a list object.

Algorithm 3: Pseudo code for extraction java source code files

Input: Verified Source Projects VP $= \{p1, p2, \dots, pm\}$;

Output: Source Code File Paths CP $= \{cp1, cp2, \dots, cpn\}$; # $cp\ refers\ to\ code\ pathe, n = j$

Procedure $SourceCodeFileExtraction$

(1) Initialize LIST ProjectSourceFiles;

(2) **for** i, i → m **do**

(3) **for** eachDir, DirName in os.walk(VP[i])**do**

(4) **for** sourceCodePath Path(eachDir).glob('.java) ***do***

(5) **ADD** sourceCodePath to ProjectSourceFiles

(6) **end for**

(7) **end for**

(8) **ADD** ProjectSourceFiles to CP

(9) **end for**

(10) return CP

end Procedure

1	Project	Source Code Path
2	ant-1.3	F:\Datasets\Data_Sample\ant-1.3\src\antidote\org\apache\tools\ant\gui\About.java
3	ant-1.3	F:\Datasets\Data_Sample\ant-1.3\src\antidote\org\apache\tools\ant\gui\Antidote.java
4	ant-1.3	F:\Datasets\Data_Sample\ant-1.3\src\antidote\org\apache\tools\ant\gui\Args.java
5	ant-1.3	F:\Datasets\Data_Sample\ant-1.3\src\antidote\org\apache\tools\ant\gui\Main.java
310	ant-1.3	F:\Datasets\Data_Sample\ant-1.3\src\main\org\apache\tools\ant\types\ZipFileSet.java
328	ant-1.3	F:\Datasets\Data_Sample\ant-1.3\src\main\org\apache\tools\tar\TarEntry.java
362	ant-1.3	F:\Datasets\Data_Sample\ant-1.3\src\testcases\org\apache\tools\ant\util\regexp\JakartaRegexpMatcherTest.java
363	**Total**	**361**

Figure 3.6: Sample data with java source code files

In the third part of this program as depicted in Algorithm 4, every matched java source code file of a software project is opened as read the character streams through adopting a character encoding mechanism called '*cp437*', and stored to a file object. Though employing the python-based java

source code parser called javalang, the stored characters stream of a file is parsed into AST representation, and then the representative node types called string tokens are selected. The obtained AST representations are transverse into a sequence of string tokens via employing DFT. All of these data sets of the mapped source code files are stored as a dictionary of the file name to a sequence of AST tokens. Finally, the obtained sequence data is exported to a csv formatted file.

Algorithm 4: Pseudo code for parsing source code file in to AST representation

Input: Source Code File Paths CP $= \{cp1, cp2, \ldots, cpn\}$;

: Representative String Nodes $S = \{ s1, s2, \ldots, sn\}$;

Output: Sequence of AST's String Vectors V $= \{v1, v2, \ldots, vn\}$

Procedure $SequenceDataExtraction$

(1) Initialize $\text{AST}_{\text{transversed}}$;

(2) $Initialize\ DICTIONRY\ V$

(3) **for** i → n **do**

(4) ASTi constructing AST from cpi;

(5) Traversing $node$ in ASTi by DFT;

(6) **If** $node$ in S **then**

(7) APPEND node$i$ into $\text{AST}_{\text{transversed}}$;

(8) **end If**

(9) ADD $\text{AST}_{\text{transversed}}$ $with\ index$ MAP cpi to V

(10) EXPORT $\text{AST}_{\text{transversed}}$ to csv file

(11) return V

end Procedure

1	Project	Source File Path	Sequence Data
7	ant-1.3	F:\Datasets\Data_	['PackageDeclaration', 'ClassDeclaration', 'ConstructorDeclaration', 'ReferenceType']
345	ant-1.3	F:\Datasets\Data_	['PackageDeclaration', 'ClassDeclaration', 'ConstructorDeclaration', 'FormalParameter',
363	ant-1.3	F:\Datasets\Data_	['PackageDeclaration', 'ClassDeclaration', 'MethodDeclaration', 'ReferenceType', 'Retu
364	Total	--	361

Figure 3.7: Sample data for sequence data.

3.2.1.3 Source Code Metric- and Context-based Defect Data Extraction

The proposed SDP method uses software metric- and context-related information's as input, in addition to extracting separate metric data and sequence data, our data preprocessing model will integrate both them and extract a unified defect data with metric data and sequence data for a unique source code files. This sub-program i.e. part of our data preprocessing module includes three main tasks namely ASTs encoding (tokenization), padding and truncating raw sequence data, and extracting defect data with metric and sequence data, and are detailed in the following sections.

ASTs encoding (Tokenization) which is presented in Algorithm 5, since deep learning methods (Miltiadis Allmanis et al., 2018) takes numeric vectors as inputs, unique AST string node types are encoded into unique integer numbers. To achieve this, in this part, a mapping is built to link each unique textual element of sequence data (string tokens) to unique integer numbers. These integers act as 'tokens" that uniquely identify each textual element (Guanjun Lin, 2020). Assuming that the number of string tokens is n and each token corresponds to a unique integer, then the mapping range is from 1 to n. Firstly, we select the 29 representative AST node types as string tokens (please refer to Table 5 for more details). After that, we establish an indexed dictionary starting from 1 to the string tokens, in which the unique string token is mapped unique integer number i.e. numeric token. Finally, the textual elements of the sequence data are replaced with numeric tokens, and their sequence remains intact (Guanjun Lin, 2020).

Algorithm 5: Pseudo code for encoding AST string tokens to numeric tokens.

Input: Representative String Tokens $R = \{s1, s2, \dots, sn\}$;

: File's Sequence Data with String Tokens $V = \{f1, f2, \dots, fm\}$; # f $refers$ to $file$ $name$

Output: Numeric Sequence Data $SeqData_{num} = \{t1, t2, \dots, tz\}$; # $z = m$

Procedure *ASTencoding*

(1) Initialize *DICTIONRY Str2IntMap*;

(2) *Initialize Index to* 1;

(3) **for** token in R **do**

(4) MAP token to *Index*;

(5) ADD token , *Index to Str2IntMap*;

(6) *INCREMENT Index* by 1;

(7) **end for**

(8) **for** fileName in V **do**

(9) Initilize LIST $\text{SeqData}_{\text{num}}$;

(10) **for** StrToken in V[fileName] **do**

(11) ADD $Str2IntMap$[StrToken] to $\text{SeqData}_{\text{num}}$;

(12) **end for**

(13) return $\text{SeqData}_{\text{num}}$

end Procedure

1	Project	File Path(Name)	String Sequence Data	Numeric Sequence Data
7	ant-1.3	ACSDefaultElem	['PackageDeclaration', 'ClassDeclaration', 'ConstructorDecla	['5', '7', '9', '14']
17	ant-1.3	ACSTreeNodeEle	['PackageDeclaration', 'ClassDeclaration', 'ReferenceType']	['5', '7', '14']
35	ant-1.3	NoOpCmd.java	['PackageDeclaration', 'ClassDeclaration', 'ConstructorDecla	['5', '7', '9', '12', '14', '26', '3', '8', '14']
53	ant-1.3	DynamicCustomi	['PackageDeclaration', 'ClassDeclaration', 'StatementExpres	['5', '7', '26', '1', '14', '14', '26', '1', '14',
65	ant-1.3	BusFilter.java	['PackageDeclaration', 'InterfaceDeclaration', 'MethodDecla	['5', '6', '8', '15', '12', '14']
121	ant-1.3	Constants.java	['PackageDeclaration', 'InterfaceDeclaration']	['5', '6']
219	ant-1.3	ClassFileIterator.	['PackageDeclaration', 'InterfaceDeclaration', 'MethodDecla	['5', '6', '8', '14']
363	ant-1.3	JakartaRegexpMa	['PackageDeclaration', 'ClassDeclaration', 'MethodDeclaratic	['5', '7', '8', '14', '21', '14', '9', '12', '14',
364	**Total**	==	**361**	**361**

Figure 3.8: Sample data for numeric sequence data.

The second one is extracting sequence data with unified sequence length through the Padding and Truncation method. Padding and truncation (Guanjun Lin, 2020) are standard practices to handle vectors of various lengths and provide a unified length of input vectors: in order to avoid vectors being too sparse, the appropriate vector length should be selected, and for a vector whose length is longer than the specified length, the extra part is truncated. For a vector whose length is less than the specified length, it is padded with 0 because 0 does not have any meaning since we map tokens starting from 1. Since the token with a higher frequency is mapped into smaller integers, the token with the lowest frequency is mapped into the maximum integer. In addition to the structural information, the code semantics (Nivetha. R, Kavitha.S, 2019) also needs to be preserved through word embedding for building a highly accurate defect prediction model.

1	Project	File Path(Name)	Numeric Sequence Data
6	ant-1.3	ACSBeanDescriptor.java	['5', '7', '9', '12', '14', '26', '
7	ant-1.3	ACSDefaultElement.javi	['5', '7', '9', '14']
17	ant-1.3	ACSTreeNodeElement.j	['5', '7', '14']

1	Numeric Sequence Data(Unified)
6	['5', '7', '9', '12', '14', '26', '1', '1', '26', '1', '1', '1', '1', '26', '1
7	['5', '7', '9', '14', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',
17	['5', '7', '14', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0',

Figure 3.9: Sample data for Numeric sequence data with unified sequence length.

The third one is concerned with integrating the metric and sequence data for a matched file samples. The source code metric-related information (e.g. static code metric attributes) and the sequence data obtained by the data preprocessing module (python program) are then combined to form reconciled and integrated defect data. Below is the pseudo-code that is used to extract defect data with metric and sequence data.

Algorithm 6: Pseudo code for combined defect data extraction

Input: Mapped Files $MF = \{Lf1, Lf2, \ldots, Lfn\}$; # Lf refers to Labeled file

:Sequence of AST's String Vectors $V = \{v1, v2, \ldots, vn\}$

Output: Matched Defect Dataset $MD = \{d1, d2, \ldots, dn\}$; # Mp refers to Mapped project

Procedure *SourceCodeFileExtraction*

(1) Initialize DICTIONARY MD;

(2) **for** SourceFileName in V **do**

(3) **for** LabledFileName in MF **do**

(4) **if** SourceFileName.endswith(LabledFileName]) ***then***

(5) **GET** V[SourceFileName], MF[LabledFileName];

(6) **ADD** AST reprsentaion, Metric data with index of LabledFileName to

(7) **end if**

(8) **end for**

(10) return MD

end Procedure

Note: - Defect Data for Mapped File represents both metric data (i.e. collections of static code metric attributes) and sequence data (i.e. sequence of AST tokens with numeric representations).

Table 3.4: Sample data for combined defect dataset

1	Project	File Name	Metric Data	Sequence Data	Bug	Defective
2	ant-1.3	org\apache\tools\ant'	['11', '4', '2', '14', '42', '29', '2', '12'	['5', '7', '14', '10', '14', '15', '10', '14', '10',	0	0
24	ant-1.3	org\apache\tools\ant'	['22', '4', '1', '15', '104', '185', '2', '1	['5', '7', '14', '10', '14', '10', '15', '10', '14',	3	1
45	ant-1.3	org\apache\tools\ant'	['6', '2', '0', '3', '11', '13', '2', '1', '6',	['5', '7', '14', '10', '8', '12', '14', '26', '3', '3',	0	0
65	ant-1.3	org\apache\tools\ant'	['11', '2', '0', '7', '15', '13', '5', '3', '1	['5', '7', '14', '10', '14', '10', '14', '10', '14',	0	0
116	ant-1.3	org\apache\tools\ant'	['4', '2', '0', '3', '16', '0', '0', '3', '4', '	['5', '7', '14', '10', '8', '12', '14', '26', '3', '1',	0	0
117	ant-1.3	org\apache\tools\mai	['27', '1', '0', '3', '63', '297', '1', '2',	['5', '7', '14', '10', '14', '10', '14', '10', '10',	0	0
118	**Total**	__	__	__	**116**	**116**

Legend: - The numeric values in Metric, and Sequence Data indicates static code metric attributes[2], and AST's node types of a given source code, respectively.

3.2.2 The Obtained Datasets

The resulting dataset of the file-level defect data modeling process for a given project is a set of file-level defect datasets with static code metric attributes as metric data and a sequence of AST tokens representations as contextual data. The targeted software systems altogether included 14, 355 labeled source code files, and resulted in 23,989 java source code files for AST representations extraction. Unmatched source code files and labeled file samples (i.e. source code files missed for its AST representation or bug report) were excluded during file-level data modeling, so the number of samples for our defect prediction experiments were brined to 13, 885 file-level defect instances.

Table 3.5: Statistical result for the extracted java, PROMISE and matched file instances

#Project	#Java source codes	#PROMISE defect instance	#Matched defect instances	Detail statistics result
42	23, 989	14, 355	13, 885	Please refer to Appendix C

The resulting data set is a mixture of the sequence of AST tokens and static code metric attributes, bug numbers, and defect labels (class). A complete feature list and the feature's origination are presented in Appendix C.2. The resulted file-level dataset is extracted for each project, each version, as well as a combined data set (a compilation of all the metric and sequence data sets) as presented in

Table *3.4*.

[2] The static code metrics used in this study are:

wmc	dit	noc	cbo	rfc	lcom	ca	ce	npm	lcom3	loc	dam	moa	mfa	cam	ic	cbm	amc	max_cc	avg_cc

3.2.3 Experimentation Framework

The experimentation framework of this study has two parts. The first part deals with the modeling of file-level defect datasets such as defect datasets with mapped metric attributes and AST representations and the analysis on the mapped and unmapped instances i.e. source code files of software defect datasets from the open-source PROMISE repository. The second part deals with the training and the evaluation of the predictive models using the extracted defect datasets.

3.2.3.1 Software Engineering Experimentation Setting

In order to investigate the contributions of the proposed file-level data extraction method on the software defect dataset quality, several experiments were designed and executed to validate the quality of the resulting software defect dataset. As detailed in section 4.1, this experimental setting is focused to test the class imbalance (rate of defects) of the software defect dataset, and an optimal sequence length as a parameter for the extraction of AST representations.

3.2.3.2 Comparative study setting for defect prediction approaches

In order to make a comparison between the defect prediction performances of SDP approaches, we derived the following four scenarios that will be considered in our experiments:

- ✓ DP-CM represents defect prediction using code metric features such as BMF, MF
- ✓ DP-SF that represents software defect prediction using semantic features i.e. SF
- ✓ DP_hybrid which indicates defect prediction using combined features (i.e. CMSFs).

Finally, we will further explore prediction performance under different parameter settings for the SDP-CMSF model. For each project, note that we use the traditional hand-crafted and semantic features from the older version to train the SDP-CMSF model. Thus, in the testing phase, only source code features are required as inputs for the deep model, and no code metrics features are further desired.

3.3 Model Building

Most of the existing SDP methods have been adopted deep learning approaches to learn discriminant feature representation as raw feature representations from either static code metrics (Haonan Tong et al., 2018; Zhou Xu et al., 2019), or program's AST representations (Yili et al., 2019; Shi Meilong et al., 2020), and even from both (Nivetha. R and Kavitha.S, 2019; Shaojian Qiu et al., 2019) for defect prediction tasks. We observe that, the advantage of unified feature representations from code metrics and ASTs for the improvement of defect prediction models. Specifically, the unified features representations by DL models, are capable to capture

discriminative information and utilized for defect prediction tasks like discriminating buggy files from non-buggy ones. In this study, we hope that the defect information carried by handcrafted features can be kept for the defect prediction tasks. As depicted in

Table *3.4*, the learned features representation of code metrics is combined with the deep semantic features from ASTs to form the unified features representation for effective defect prediction.

This section describes how we apply deep learning networks to handle the defect data with different representations (e.g. code metric- and AST-based defect data representation), and different processing methods (e.g. static code attributes and sequence of AST tokens) for learning unified high-level representations with respect to the defects of interest. To process the data representations that are heterogeneous (Yao Wan et al., 2019), our solution is to feed them to different networks and use one of the hidden layers' output as the learned high-level representations.

3.3.1 Raw representations

Prior to feeding the data representations to the respective networks, they need to be converted to a suitable form (i.e. vectors representation) that is compatible with a neural network. In this thesis, the convection processes take place in the encoding phase of SDP-CMSF and considered as data pre-processing. We name the converted data that is ready for feeding to the neural networks for high-level representation learning as raw representations.

3.3.1.1 Source code metric-level data processing

In this study, we are going to utilize discriminant features from code metrics for building a defect prediction model that is capable of discriminant defective files from non-defective ones. Since the values of different datasets usually have a different order of magnitude, we conduct data normalization on these datasets. In this study, the standard z-score normalization method is employed as data preprocessing on traditional hand-crafted features from the PROMIS repository (Tianchi Zhou et al., 2019). Then the normalized numeric vectors containing static code attributes (for more details, please refer to section 2.2.1.1) will become ready for the subsequent discriminant features (i.e. representations) learning processes by the proposed deep neural network.

3.3.1.2 Source code context-level data processing

Prior to extracting source code contextual information (i.e. syntactic and semantic features) for SDP tasks, by adopting language models (Guisheng Fan et al., 2019; Nivetha. R, Kavitha.S, 2019)

as detailed in section 3.2.1, we have to obtain the AST representations of program modules (in case of this study source code files) called a semantic-based model.

1. Robust AST Parsing

ASTs are usually generated by a compiler during the code parsing stage (Guanjun Lin, 2020). However, without a working build environment, obtaining ASTs from Java source code is non-trivial. With javalang, we can extract ASTs from individual java source files or even from fragments of file code without the presence of dependent libraries. By feeding javalang with the source code files, the parsed ASTs can be generated, ready for AST refining processing.

Following the relevant methods of we only select three types of ASTs' nodes as tokens which includes: (1) nodes of method invocations and class instance creations which are represented as method names or class names, (2) declaration nodes (i.e. method declarations, type declarations, and enum declarations) which are represented by their values and (3) control-flow nodes (i.e. while statements, catch clauses, if statements, and throw statements) which are represented by their node types (Cong Pan et al., 2019). As detailed in (Ziyi Cai et al., 2017; sGuisheng Fan et al., 2019 for method invocations we record them as their plain text in the source code, the nodes of declarations we extract their node names as tokens, and Control flow nodes are simply recorded as their node types and also nodes of AssertStatement and TryResource are recorded as their values. Three types of selected nodes are listed in Table *3.6*.

Table 3.6: Representative AST nodes (*Ziyi Cai et al., 2017; Miltiadis Allmanis et al., 2018*)

ID	Node types	AST node record (#)
Node1	Nodes of method invocations and instance creations	MethodInvocation, SuperMethodInvocation, ClassCreator
Node2	Declaration-related nodes	PackageDeclaration, InterfaceDeclaration, ClassDeclaration, ConstructorDeclaration, MethodDeclaration, VariableDeclarator, FormalParameter

Node3	Control-flow-related nodes	IfStatement, ForStatement, WhileStatement, DoStatement, AssertStatement, BreakStatement, ContinueStatement, ReturnStatement, ThrowStatement, TryStatement, SynchronizedStatement, SwitchStatement, BlockStatement, CatchClauseParameter, TryResource, CatchClause, SwitchStatementCase, ForControl, EnhancedForControl
Others	AssertStatement and TryResource	Recorded as their values

2. AST refining

2.1 Code Structure preserving

With the parsed ASTs in a tokenized format, they need to be transformed to vectors so that they can be processed by ML algorithms. To preserve the structural information of ASTs, we applied the same method addressed in our previous work (Guisheng Fan et al., 2019; Guanjun Lin, 2020). Firstly, the ASTs have to be traversed, allowing their components to be assembled in a uniform sequence to form vectors. In this study, we employ the DFT method (Guisheng Fan et al., 2019) to turn ASTs of each program into a vector. Therefore, the converted vector should uniquely identify a file. By doing so, we can preserve the structural and contextual information to a large extent.

2.2 Code Semantic preserving

Since the subsequent deep learning methods (Miltiadis Allmanis et al., 2018) takes numeric vectors as inputs, textual elements of a vector are mapped to numbers. To achieve this, a mapping is built to link each textual element of vectors to an integer. These integers act as ‘tokens" that uniquely identify each textual element (Guanjun Lin, 2020). Assuming that the number of tokens is n and each token corresponds to a unique integer, then the mapping range is from 1 to n. As detailed in (Guisheng Fan et al., 2019), Firstly we count the frequency of each token and then sort them based on the token frequency, then we establish an indexed dictionary of the ordered tokens, in which tokens with higher frequency are in front (Replacing textual elements of vectors with numeric

tokens, their sequence remains intact) and finally we make these digital vectors the same fixed length via padding and truncation methods..

2.3 Padding and Truncation

Padding and truncation (Guanjun Lin, 2020) (Guisheng Fan et al., 2019) are standard practices to handle vectors of various lengths and provide a unified length of input vectors. In order to avoid vectors being too sparse, the appropriate vector length should be selected: for a vector whose length is less than the specified length, it is padded with 0 because 0 does not have any meaning since we map tokens starting from 1, a vector whose length is longer than the specified length, the extra part is truncated (Since the token with a higher frequency is mapped into smaller integers, the token with the lowest frequency is mapped into the maximum integer), then we locate the index of the maximum integer in the vector and delete it each time until the vector length becomes the same as the fixed length (Guisheng Fan et al., 2019). In addition to the structural information, the code semantics (Nivetha. R, Kavitha.S, 2019) also needs to be preserved through word embedding for building a highly accurate defect prediction model.

As described in Section 2.2, the word embedding learning is also taking place in the encoding phase of the proposed language model, and it is considered as a data pre-processing. The word embedding (Cong Pan et al., 2019) represents each ‘word (tokens)" of the input vector as a dense vector of a fixed dimension. To allow the algorithm (Guanjun Lin, 2020) to leverage the information that is held between the nodes (namely, the elements of vectors) and be more expressive we apply the Word2vec model to convert each element of the vectors to word embeddings of 100 dimensions (which is the default settings). Specifically, the source code files are parsed into AST nodes using Algorithm 1. After that, the token vectors are encoded into numeric integer vectors and preprocessed (e.g. tokenization, padding, and truncation), and then, the preprocessed sequence of integer vectors again inputted to Word2vec model for distributed representation learning. The resulting raw representation by the Word2ve model becomes compatible and ready for the subsequent process by language model since neural networks (Song Wang et al., 2016) only take numeric vectors as inputs for further processing.

3.3.2 The Deep Neural Networks for Representation Learning

Software defect prediction is a context-dependent task (Guisheng Fan et al., 2019): such that the occurrence of defective code snippet can be usually associated with either preceding or succeeding

code, or even with both, incense the structure of source code is logical and semantic, being coherent and closely connected. As detailed in Section 3.3.1.2, DFT and word embedding methods have been commonly adopted in context-based SDP methods, in order to preserve the structural, and semantic features of source code from AST representations, respectively. Preserving structural and semantic information of program module through word embedding models (e.g. Word2Vec with Skip-gram model (Cong Pan et al., 2019), Word2Vec with Continues-Bag-Of-Words model (Shayan A. Akbar et al., 2019)), and high-level features representation as contextual features via deep neural network models which includes CNN (Cong Pan et al., 2019), Bi-directional LSTM (Guisheng Fan et al., 2019; Nivetha. R, Kavitha.S, 2019; Ming Wen et al., 2017) had also become a common practice in deep feature-based SDP. In this study, word embedding is adopted as the input layer of the proposed language model for representation learning.

In order to learn the unified representation from vectors of static code attributes and AST tokens of the source code file, two different neural networks are used in this study, which will be described in the following subsections. Since the study is aimed to improve the performance defect prediction model, by utilizing different aspects of key information from source code metric and AST representations (Yili et al., 2019).

1 FCN-based approach

In this study, an attention- based FCN is adopted in order to learn (extract) discriminant features from hand-crafted features of defect data. The staked structure (Cong Pan et al., 2019) of FCNs is also called as MLP, and has the potential of learning richer models than ML algorithms given a large dataset. As depicted in Figure *3.10 (b)* the attention (max pooling) layer is implemented on top of MLP layer, and the MLP is employed in order to learn features from training data including different sets of static code metric types. As detailed in Section 1.1.1.2, the MLP models are capable to fit the complex software defect data patterns (i.e. characteristics) that are highly non-linear and abstract and have been successful adopted in defect prediction studies (Guisheng Fan et al., 2019; Zhou Xu et al., 2019) (Rudolf Ferenc et al., 2020; Xuan Huo et al., 2017).

The training processes of MLP model is also detailed in Section 1.1.1.2, but it is brined here for the matter of clarification. During training a non-linear activation function within the neuron is then applied to the weighted sum to produce an output which will be propagated to neurons of the

subsequent layer. Suppose that the input layer has n neurons and the output of the i^{-th} neuron in the subsequent hidden layer can be calculated by the following formula:

$$yi = f\left(\sum_{i=1}^{n} Wni.xi + b\right) \qquad 3.1$$

where xi and yi are the input and output of the neuron; Wni is the weight matrix between the n^{-th} neuron in the previous layer and the current neuron, and b is a bias term. Particularly, the $f(:)$ is a non-linear activation function (e.g. the Sigmoid function, Hyperbolic Tangent, Rectified Linear Unit (ReLU)) which brings the non-linearity to the output (Bin Liu et al., 2019). With a cascade of neurons in each layer, the application of non-linear activation functions of each neuron will result in high non-linearity of the original input.

The loss function $L(\widetilde{\gamma}, \gamma)$ is used during training phase to minimize the gap between the actual label of γ and the generated output $\widetilde{\gamma}$. The modern networks use the backpropagation rule (Zhou Xu et al., 2019), to propagate the loss backward to adjust the weights of each neuron and the weight-adjusting process is accomplished by iteratively computing the gradients. In practice, stochastic gradient descent (SGD) using mini-batches is the common method for training neural networks and applied by all the reviewed studies (Cong Pan et al., 2019; Nivetha.R, Kavitha.S,, 2019). The mini-batch SGD is to split the training samples into small batches and the gradient is calculated and weights are updated after processing the samples in a small batch.

The FCN (Aston Zhang et al., 2019) has the potential of learning richer models than conventional ML algorithms given a large dataset. This potential has motivated us to use it for modeling the software defective data, which are latent and complex. Another advantage of the FCN is that it is "*input structure agnostic*", which means that the network can take any forms of input data (e.g. textual and numeric data (Thong Hoang etal.) images or sequences (Zhou Xu et al., 2019)). This also offers us the flexibility to traditional hand-craft various types of features for the network to learn from.

2 Attention-based RNN approach: We apply a bi-directional LSTM layer (Guisheng Fan et al., 2019), using an attention (global max pooling) layer on top of it as the weights to apply the importance on each time step during the training as shown in Figure 3.10. *(a)*, the proposed language model learns crucial features from vectors of AST tokens for defect prediction task.

Attention mechanism (Aston Zhang et al., 2019) 'is a generalized pooling method with bias alignment over inputs'. The language model implementation is detailed in 1.1.1.2, for convenient of understanding we have bring it here.

LSTM implementation (Hongliang Liang et al., 2019) is a variant of an RNN which uses a gate mechanism, capable of capturing long-term dependencies for the input sequences. Compared with the RNN, it is not subject to the vanishing gradient issue, which leads to more effective model training and better performance. According to an up-to-date design of the LSTM (Guanjun Lin, 2020), an input *xt* to an LSTM unit at the current time step *t* can be expressed as follows:

$$it \; = \; \sigma\, W^{i} ixt \; + U^{\,i}\, iht - 1 \; + \; b^{\,i}\,;$$

$$it \; = \; \sigma\, W^{i} xt \; + \; U^{\,i} ht - 1 \; + \; b^{\,i};$$

$$ft \; = \; \sigma\, W^{f}\, xt \; + \; U^{f} ht - 1 \; + \; b^{f}\,;$$

$$ot \; = \; \sigma W^{o}\, xt \; + U^{o} ht - 1 \; + \; b^{o})\,;$$

$$ut \; = \tanh(W^{u} xt \; + \; U^{\,u} ht - 1 \; + \; b^{\,u})\,;$$

$$CSt = \; it \odot ut \; + \; ft \odot CSt - 1;$$

$$ht \; = \; ot \; \odot tanh(CSt); \qquad 3.2$$

Where it, ft and ot denote input, forget and output gates, respectively. There are four input weightsW^{u},W^{i}, W^{f} and W^{o}, corresponding to the unit input, the input gate, forget and output gates, respectively. There are also four recurrent weights$U^{\,u}$,$U^{\,i}$, $U^{\,f}$ and $U^{\,o}$, and four bias terms $b^{\,u}$, $b^{\,i}$, $b^{\,f}$ and $b^{\,o}$, respectively. The 'CSt' is a built-in memory cell of an LSTM node which can maintain its internal state over multiple time steps; ht is a hidden state and ut is an input node. The weight matrix *W* is the weight between the input and the current hidden layer, and *U* is the weight between the current and previous hidden layer. The symbol σ, and $\tanh$ represent the non-linear sigmoid function and Hyperbolic Tangent function, respectively. The Θ signifies element-wise multiplication. The gate mechanism controls the update of each unit and the state of the memory cell, which allows the network to learn long range dependencies (Guisheng Fan et al., 2019).

The occurrence of buggy code fragment usually encompasses a number of code parts (i.e. statements) either preceding or subsequent code or even both, modelling such contextual

information significantly discriminate buggy code patterns from non-buggy one (Nivetha.R, Kavitha.S,, 2019; Guanjun Lin, 2020), the modelling methods, was based on the implementation of bidirectional RNN model with LSTM cells as basic building block. The underlying assumption is that the Bi-LSTM represent the long-term dependencies in both forward and reverse directions and the code semantics, and the buggy programming patterns which was hidden in the source code can be revealed by analyzing Bi-LSTM model's feature representations. This thesis is also based on this assumption, and the language model implementation is also based on (Guanjun Lin, 2020). The n^{-th} hidden layer output (i.e. feature representation) in Bi-LSTM network can be calculated using the following equation:

$$ht' = \mathcal{Z}(\overrightarrow{ht}, \overleftarrow{ht}), \qquad 3.3$$

where $n \leq 1$, *is the number of hidden layers*; $\overrightarrow{ht}$, and $\overleftarrow{ht}$ are the values of the hidden layers in the forward and reverse directions, respectively. Function $\mathcal{Z}$ can be a concatenating, summation, averaging or multiplication function. In the implementation, the forward and reverse directions required to output a list (tensor) and we keep if for subsequent processing called unified representation.

Pre-training for Obtaining Semantic Features Representation

During building Bi-LISTM for high-level feature representations, we followed the pre-training procedure commonly adopted in deep feature-based defect prediction studies (Cong Pan et al., 2019; Xuan Huo et al., 2017). The pre-training phase trains the Bi-LSTM network using historical defect data of older version(s) of software projects in a given system. These projects (i.e. training projects) initialize the network parameters for learning low-level features of defective files. As depicted in Figure *3.10* (a), the dense layers are added after the global max-pooling layer to form a complete network and to converge the network into probably. The training inputs are AST-based features from both defective and non-defective source code files. . This makes sure that the hidden nodes capture the sequential interactions that are discriminative of vulnerable programs. The input data are divided into training and validation sets to build and evaluate the language model and guide the model tuning processes to maximize the performance. Once the model is trained and the performance is satisfactory, we remove the dens layers (i.e., component C in Figure *3.10*) and feed the trained networks with the pre-processed AST-based features of the testing project and obtain the learned representations from the max-pooling layer of the networks of component A. Given a

sequence of an arbitrary length as an input fed to the networks, the learned representation is a 128-dimensional vector (Guanjun Lin, 2020). The learned representations are the high-level abstract features that can be used for training a conventional ML classifier as a defect prediction model.

Algorithm 7: Pseudo code for pre-training the LHFR model with combined defect dataset

Input: Hand-crafted Data HD = $\{HF1, HF2, \ldots, HF21\}$; # HF refers to Hand-crafted feature

:Sequence Data SD = $\{SF1, SF2, \ldots, SF128\}$ # SF refers to Semantic Features

Output: DL-generated Features Representations = $\{THF, ConcatF, and\ HybridF\}$;

Procedure $Train_LCFR()$

(1) Initialize BiLSTM, MLP, FCN, layer_num;

(2) **for** i → len(HF or SDV) **do**

(3) FEED HF[i] $to\ MLP$;

(4) FEED SD[i] $to\ BILSTM$;

(5) Concatenate $(BILSTM.Out[i], MLP.Out[i])$;

(6) FEED Concatenate. Out[i] to FCN;

(7) Return FCN. Layer. Out[layer_num]

end Procedure

3 Learning Unified Features Representation

In the previous step, given a sequence of tokenized AST nodes and a set of code metrics, we use two approaches with a global tension mechanism (Guisheng Fan et al., 2019) (Jian Li et al., 2018) to learn curial vector representations for each of source code data representations. Since different vector representations capture different aspects of information of a software program modules (Nivetha. R, Kavitha.S, 2019) (Jin Wang, et al., 2017). To learn the unified vector representation for a program module (in our case source code file), we still need to find a way to combine the two vectors representation of a source code file into one code vector without too much information loss.

To do so, in this thesis an attention model is adopted to fusion different vector representations into a unified representation. Among different types of attention methods (for example, multi-layered perceptron (MLP) (Guisheng Fan et al., 2019), global max polling (Guanjun Lin, 2020), multi-head attention (MHA) (Yili et al., 2019)), the MLP and MHA are effective in learning representation form different other representations. As shown in Figure *3.10* (C), a common attention model (i.e. MLP) is built on top of the Bi-LSTM layer (language model A) and MLP

(model B) to learn a common unified representation from the different representation of vectors for building accurate defect prediction model called SDP-CMSF.

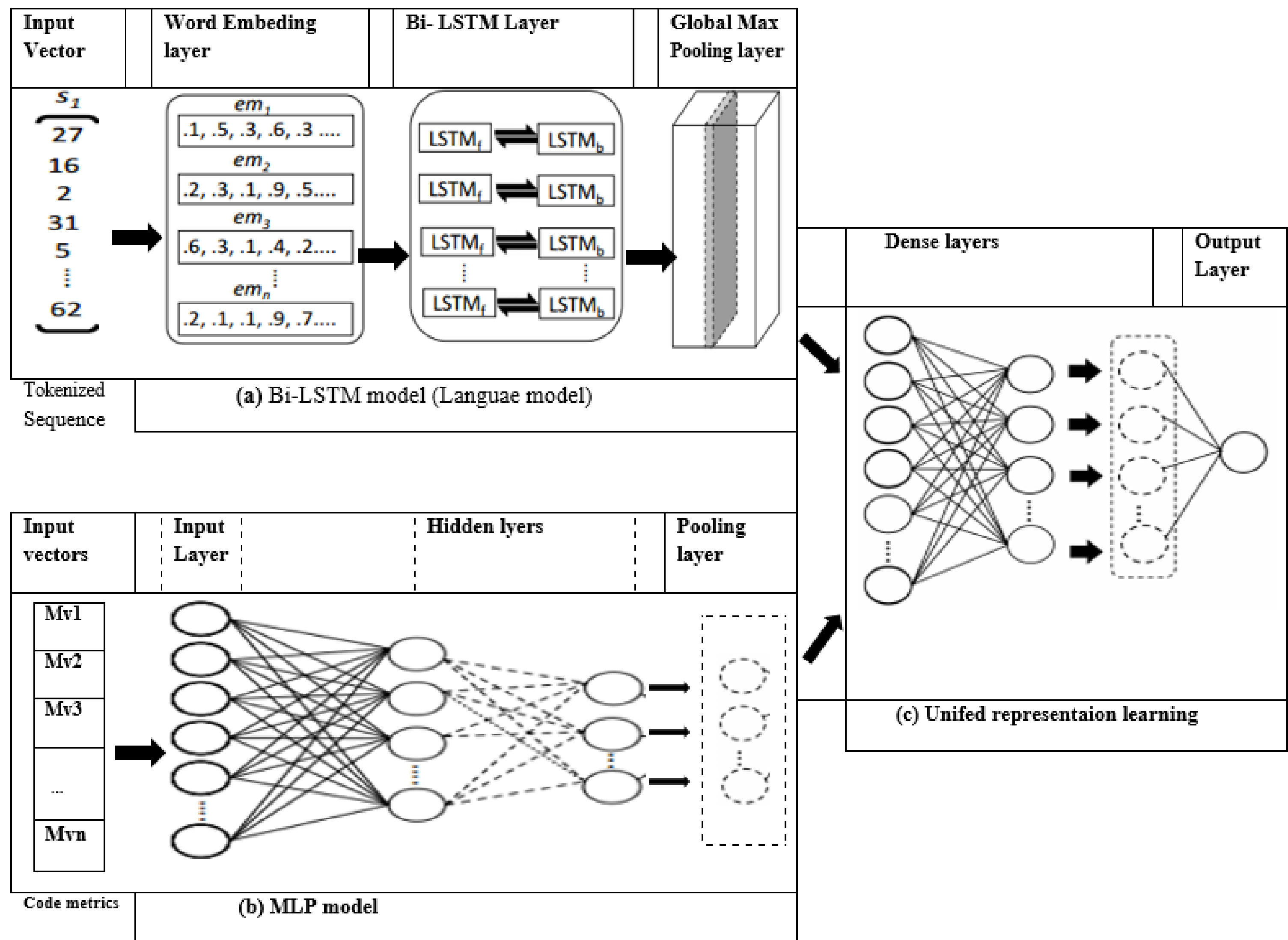


Figure 3.10 The architecture of LHFR for combined features extraction

The structure of the deep neural models (i.e. SDP-CMSF) for extracting high-level representations from the source code metric and semantic features. The representations can be the output of any hidden layer of the component modules, which includes the language model (module A), MLP model (i.e. module B) and dense layers i.e. FCN (module C). In this study, we used the output of module C as the unified high-level representation

The layers in component C of Figure *3.10* will become into common dense layers through non-linear activation functions (Nivetha. R, Kavitha.S, 2019) (Yili et al., 2019) (Ming Wen et al., 2017) for unified representation learning; in this study, the ReLU and Sigmoid activation functions are used for performing non-linear feature extraction and converging network to a probability. In the scenario of defect prediction, the problem is a binary classification. Therefore, the "sigmoid activation" will be applied on the last layer i.e. Component c in figure 8 and the loss function l to be minimized is the "binary cross-entropy":

$$l = -(yl\,og(p) + (1 - y)\log(1 - p)) \qquad 3.4$$

where y is the actual class label for that class (either 0 or 1), and p is the predicted probability of y being either 0 or 1. The experimental results in (Guanjun Lin, 2020) (Guisheng Fan et al., 2019) (Nivetha. R, Kavitha.S, 2019) had verified that using the SGD optimizer converges more quickly than other optimizers. To prevent over-fitting, we also implemented dropout for each sub-components. Different learning strategies are adopted during the training of the proposed DL model (i.e. combined model) which includes automatic min-batch vanishing (Zhou Xu et al., 2019), and cost-sensitive learning/classification (Linchang Zhao et al., 2019). The goal is the automatic min-batch vanishing method is to take the advantage of hand-crafted features and semantic features automatically for unified feature representation learning. Similarly, the objective of the implemented cost-sensitive classification is to minimize the cost of misclassification, which can be realized by choosing the class/label with the minimum risk while learning the unified features representation.

3.3.3 Model Evaluation

The essence of defect prediction in this study is a binary classification problem that can make two possible errors i.e. false positives and false negatives—in addition, a correctly classified defective class file is a true positive, and a correctly classified clean class file is a true negative (Shi Meilong et al., 2020). To evaluate the models, for every obtained (classification) result, we calculate the precision, recall, and F-measure as shown below.

Precision: Precision (equation $\text{Precision} = \frac{\text{True Postive}}{\text{True Postive+False Postive}}$ 3.5) measures the proportion of files that were correctly classified as faulty over the total number of files classified as either faulty or non-

faulty. In other words, Precision or Confidence (as it is called in Data Mining) denotes the proportion of predicted cases that are indeed real faulty files (Tianchi Zhou et al., 2019), this is a measure of how good a prediction model is at identifying actual faulty files.

Recall: Recall (equation $Recall = \frac{\text{True Postive}}{\text{True Postive+False Negative}}$ 3.6) measures the proportion of faulty files that are correctly identified as faulty over the total number of faulty files available. Recall or Sensitivity is the proportion of real faulty files that are correctly predicted as faulty files (Alegre, 2017).

F-measure (equation $\text{F} - \text{measure} = \frac{2*\text{Precission}*\text{Recall}}{\text{Precission+Recall}}$ 3.7) is computed by taking the (weighted) harmonic average of precision and recall (Shi Meilong et al., 2020).

$$\text{Precision} = \frac{\text{True Postive}}{\text{True Postive+False Postive}} \quad 3.5$$

$$Recall = \frac{\text{True Postive}}{\text{True Postive+False Negative}} \quad 3.6$$

$$\text{F} - \text{measure} = \frac{2*\text{Precission}*\text{Recall}}{\text{Precission+Recall}} \quad 3.7$$

Note: False-positive refers to the predicted defective files that actually have no defects, and false negative refers to the actually defect-prone files predicted as clean. Precision and Recall are mutually exclusive in practice—F1_measure, as a weighted average of Precision and Recall, is more likely to be adopted. The value of F-measure ranges between 0 and 1, with values closer to

1 indicating better performance for classification results (Guisheng Fan et al., 2019). In addition to the above performance measurement metrics, in this thesis AUC (Le Hoang Son et al., 2019) is used to evaluate the performance of defect prediction model with a variable cut-off value on the predicted probably of defect-proneness.

Chapter Four

4 Experiments, and Results

Computational system and software setup

All of our experiments were run on a windows 8.1 operating system with Intel(R) core(TM) i5-3337U @1.80GHz CPU, and @6.0GB RAM. We used CPUs for training deep learning. To evaluate and validate the performance of different file-level SDP approaches, experiments were conducted to answer the following research questions and following this, the parameter for each experiment will be detailed in the section of experimental setting, and the obtained results and analysis will be detailed in the subsequent evaluation results sections.

The implementation of Bi-LSTM network and MLP is used Keras (version 2.0.8) with TensorFlow[3] (version 2.3.0) backend (Guanjun Lin, 2020). The conventional ML algorithms were provided by the scikit-learn package (version 0.19.0) (Shi Meilong et al., 2020). The Word2vec embedding was provided by the genism package (version 3.0.1) with all default settings.

The Experiment, and Asks of the study

[3] https://www.tensorflow.org/install

- RQ_1: How much source code files of java software systems are utilized for software supervised defect prediction tasks?
- RQ_2: How the proposed defect prediction approach improves defect prediction performance compared to baseline approaches?
- RQ_3: How does LHFR perform, compared to the state-of-the-art defect prediction methods using combined features?
- RQ_4: How is the prediction performance of SDP-CMSF under different parameter settings?

We ask RQ_1 in order to evaluate the effectiveness of the obtained defect dataset for software defect prediction task as depicted in Figure 11.1(A); RQ_2 to evaluate and compared the performance of static code metric and semantic, and combined features-based predictive models. We ask RQ_3in order to evaluate the performance of the combined and unified feature representation (i.e. module c) compared to state-of-the-art combined features-based defect prediction methods and RQ_4 to analyze the sensitivity of the proposed model—as shown in Figure 12.1(B)—when varying parameters like to verify the significance attention mechanism and different learning strategy (i.e. vanishing the static code features and class weight-based learning while training the defect prediction model) for the improvement of defect prediction model performance.

4.1 File-level Defect Dataset Quality Evaluation

Given the lack of file-level defect datasets and modeling methods dedicated to a specific programming language, we performed an expletory data analysis and extraction to fill this gap. We used the existing software defect datasets and java files of software systems in order to extract matched defect datasets by adapting pattern matching and extraction processes into our file-level defect data modeling method and evaluated the quality of obtained defect dataset relative to the existing defect datasets. In Section 3.2.1 we have presented the file-level defect data extraction processes and resulted in defect datasets. The experiments presented in this sub-section testes the quality of the obtained defect datasets in case of defect rate.

This section is structured as follows. In Section 4.1.1, we introduce the study settings, the targeted systems, and the methodology applied. In Section 4.1.2, we present the results we obtained. Finally, in Section 4.1.3 we discuss the results found.

4.1.1 Experiment Setting

Next, we present the methodology used in this experiment. We used one research question as a guideline in order to analyze the quality of the obtained datasets in case of bug/defect rate within a given target software system. To address the question, we performed an expletory file-level data analysis using 12 different software systems. We next detail our experiments.

Goal and Research Questions

In order to design our study i.e. experiment, we followed the GQM (goal-question-metric) paradigm (Alegre, 2017), according to it, the first step is to specify the goal of the study, which is the following: to assess the effect of the proposed file-level data modeling method on the rate of defects, evaluate defect rate of existing and obtained file-level defect datasets and also investigate an optimal sequence length for resulted source code AST representations using 12 open source software projects. Based on our goal, we derived the following research question:

How the obtained defect datasets of the proposed defect dataset modeling framework improves the rate of defects compared to previously investigated software defect datasets?—as RQ1. Given that most of the investigated defect dataset of software metric-based defect prediction approaches have been considering metric attributes of source code modules implemented with any programming, we investigate the rate of defect-prone instances i.e. matched AST representations with the specific programming language of software datasets, and the matched from AST representations are used to model the cost-sensitive defect datasets with optimal sequence length from 12 open-source software projects. The metrics used to answer RQ1 are detailed in the subsequent experimental procedure sections.

4.1.2 Experimental Procedure

Our experimental procedure is composed of three main steps as depicted in Table *4.1*. We first extracted and analyzed each software project in order to verify the existence of matched source code file instances in a given software defect dataset. In the second step, we prepared our combined defect dataset, by performing five activities: (I) extraction of defects; (II) extraction of static code metrics; and (III) extraction of source code's AST representations (i.e. cost-sensitive files), (IV) matching the defect dataset instances and source code files and (V) extraction of combine features from the matched instance's code metric attributes and AST representation. Last, we executed all the matched AST representations of the target systems and measured their sequence length, and

used them as the predictive feature with code metrics for building predictive models. We next provide details regarding our experiments.

Table 4.1: Python toolkit and experimental datasets for combined defect data preparation

Experimental process of the combined defect data modelling Framework		
#	Blue Print of the modules	Significance of the obtained Datasets in our experiments
Init	Initial_SETUPS( ) Load...! Get accesses to given *'local directory'*, of target datasets and perform the initialization as: (I) extract all of the source projects of given target software systems and corresponding PROMISE defect datasets and initialize as software defect datasets; and (II) get and initialize all of the AST node categories which includes 29 tokens representative tokens .	This module resulted in a dataset of: - $Source_{Projects}$, $Defect_{Datasets}$ and $Reprsentative_{Tokens}$; which are used as input to initialize/load the sub-sequent processes.
1	ExtractJavaFiles(Source Project) Load...? <Input type: Software Project, Output type: File> Input Dataset: $Source_{Projects}$; Processes-1: Transverse and extract all java files of the target source projects; Output Data(#samples): Java files(23,989 files)	Resulted with 23,989 java source code files as $Extracted_{JavaFiles}$, which is used as input to initialize the loading processes of subsequent module i.e. ParseFile2ASTRxns
2	ParseFile2ASTRxns (Java File) Load...? <Input type: Java File; Output type : tree data structure> Input Datasets: $Extracted_{JavaFiles}$, $Reprsentative_{ASTtokens}$;	Store the obtained datasets i.e. 23989 file paths and sequences as: $Obtained_{FilePaths}$ and $Obtained_{Sequences}$, respectively. Similarly, these

	Processes-2.1: Encode given java file into characters i.e. tokens stream via adopting 'cp437', and output $FilePath$ as the IDs of an encoded java file. Processes-2.2: Parse the obtained tokens stream from processes-2.1 into AST via a python-based java source code parser called ‘javalang’ (Ziyi Cai et al., 2017). Processes-2.3: Transvers each representative tokens from given AST by employing DFT as sequences of tokens called sequence representations Output Data (#Samples): File paths and Sequence representations (23, 989).	datasets are used input of the subsequent modules.
3	$\text{ExtractMappedInstances}(\text{defect instance}, \text{file paths})$ Load...? <Input type: (defect instance names, Java file paths, Output type: List> Input Data: unmatched 23,989 java file, and 14,355 defect(labeled) instances Input Data: matched 13,885 java file i.e. Result-3 Processes-3.1: obtain defect instance i.e. labeled file paths, defect data i.e. set of metric attributes, defect information form $\mathrm{D}efect_{Datasets}$, and store as $Labled_{FileNames}$, $Labled_{MetricData}$, and $Defect_{classes}$ datasets, respectively; Processes-3.2: verify the existence of the file names i.e. $Labled_{Instances}$ in $Obtained_{FilePaths}$ If relation existed: store as the corresponding file names and file paths as $Matched_{FileNames}$ called defect instances and $Mapped_{FilePathS}$; else as $Unmatched_{FileNames}$ Output Data(# Samples): Matched file names and mapped File paths and unmatched (13885 and 10,38)	The metric data, sequence data and defect information of the mapped file paths are exported as $Matched_{MetricData}$, $Mapped_{Sequences}$, $Matched_{Lab}$, and the reaming unmatched defect instances and unmapped file paths as $Unmatched_{FileNames}$ and $Unmapped_{FilePaths}$,respectively.

4	ExtractOptimalSequenceLenth(Variable Sequences) Load…? <Input type: List, Output type: List> Input Data: $Mapped_{Sequences}$; Processes-4.1: Filter all non-defective matched instance having sequence greater than all defective s as thresholder matched sequences Output Data(# Samples): Filtered sequences (11,831)	Stored as the obtained datasets as actionable sequence i.e. $Filtered_{Sequences}$, and use for optimal sequence length analysis.

Legend: $FilePath$ indicates java file, $Instance$ indicates labeled file

4.1.2.1 Target Software Projects (Datasets)

Table 4.2 outlines the resources used in each experiment in order to verify the proposed defect dataset modelling method for the improvement of software defect dataset quality, and investigate an optimum sequence length from the matched AST representations.

Table 4.2: Experiments, Projects, and Dataset used for Software Engineering related setting.

Experiment 1	Projects	Dataset	*Summary*
Experiment 1.1	All target projects	Actionable defect datasets	This experiment is designed to investigate and verify the effect of the proposed source code programming language based defect data extraction method on the improvement of software defect dataset quality.
Experiment 1.2	All target projects	Unactionable defect datasets	
Experiment 1.3	All target projects	Actionable file-level datasets (AST representations)	This experiment is designed to investigate and verify the optimum sequence length for the extraction of file level sequence defect data.

4.1.3 Results and Analysis

To measure the file-level software defect dataset qualities, we used the rate of defective file instances as the metrics for measuring the class imbalance rate under different experiment settings of the file-level data modeling framework. This metric has been widely used in various software defect prediction studies (Michele Tufano et al., 2018; Rebeen Ali Hamad et al., 2020).

Experiment-1:

All experiments of defect data modelling settings outlined in Table 4.2 were based on the dataset setup of Initial_SETUP( ) module as presented in Table 4.1 which describes basic requirement experimental data setup of CDDM framework. Different dataset setup (outputs) obtained from different tools of CDDM were used according with the aims of the experiments presented in this section. Obtained full statistical results for different experiments related with CDDM are referenced to *Appendix C. 2*. The results are presented herein.

Experiment-1.1

This experiment is conducted in order to verify the quality of the software defect dataset in terms of all target systems. The experimental datasets for this experiment were obtained from CHDD by loading two modules: (I) ExtractJavaFiles() and then (II) ExtractMappedInstances(), respectively. The execution of ExtractJavaFiles() module resulted in java source code files set as java files set; ideally, source codes i.e. file instances implemented with other than java programming language are not recognized in this process. This module resulted in a total of java files 23,989. The execution of ExtractMappedInstances() module is targeted to matched and extract all java files that existed in the set of defect instances of the PROMISE repository as the set of mapped files and resulted in a labeled java file of 13,885 as *matched defect instance* from labeled files of 13,355.

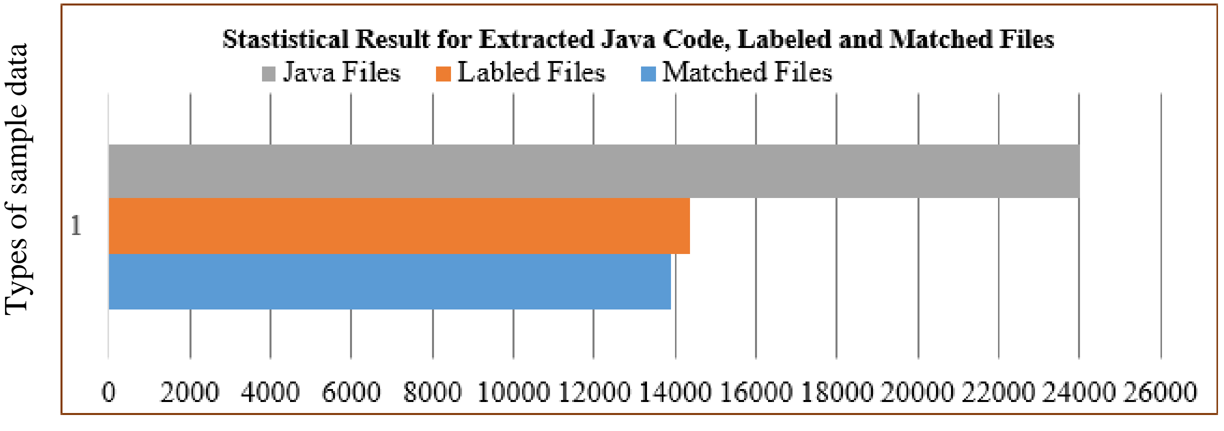


Figure 4.13: Bar Chart of Raw Labeled Files and Extracted Source Code Files

Figure 4.13, depicts the bar chart of total values of the files/instance of the target software system for our CDDM framework under defect instances with three types of indicators; the extracted matched defect datasets i.e. java file type are very similar the labeled java files(matched files) and

all defect instances (labeled files) of PROMISE repository indicators (all except real-world files of target systems i.e. java files). This implies that CDDM resulted in more qualified mapped java source code representations and matched defect instances of PROMISE repository as software defect dataset for our tasks related to supervised learning (i.e. training ML prediction models); in addition to this, CDDM provides plenty of unmapped java files with mapped java files for unsupervised learning-related tasks—e.g. tokenization of AST representation, word embedding—i.e. semantic words/features representation learning—as detailed in section 3.4.1.2.

Experiment 1.2

In this experiment, the analysis of the obtained variation in the results of labeled and mapped files was done in terms of defect rates as indicators. Additionally, this experiment aims to investigate the quality of the matched file set by verifying file types and the imbalanced rate of the dropped labeled files of the PROMISE repository. This analysis helps evaluate the software defect dataset i.e. mapped files quality in terms of hand-crafted static code metric features for defect prediction in our context, i.e. file-level SDP. This experiment resulted in 470 unmatched defect instances (i.e.3.14%) and the analysis using 470 unmatched defect instances set showed in not existed files of 406 and not java files of 64 with a dropped defect rate of 21% as detailed in Table *4.3*. This implies that the quality of the obtained software defect dataset (mapped files) from CDDM is slightly better than all labeled files set in terms of sample-level quality indicator i.e. defect rate. However, the quality of matched software defect datasets obtained from mapped files with source code metrics is low (i.e. is affected negatively by -3.14%) when compared to metric features-based defect datasets of the PROMISE repository. This emphasizes the need for contextual feature extraction.

Table 4.3: Results for software systems with unactionable files.

Software system	Unmatched Files			Defect rate (FP %)
	#	#Not Existed File	#Not Java Files	
Ant	49	49	0	0.23
camel	69	19	50	0.0
Jedit	112	111	1	0.23
log4j	46	37	9	0.16
lucene	32	32	0	0.31
Poi	16	16	0	0.0
synapse	17	17	0	0.29

velocity	4	0	4	0.5
xalan	114	114	0	0.1
xerces	11	11	0	0.29
Total	470	406	64	0.21

Legend*: # indicates total samples*

Experiment 1.3

This experiment tested the optimal Sequence length for the extraction of AST representations using mapped files from 41 software projects. The experimental datasets for this experiment were obtained by loading/initialization the ExtractOptimalSequenceLenth() of CDDM framework which results in total AST representations of 13,885. While mining the optimal AST's optimal sequence length ($SeqLent$), a novel data analysis method named $CPMSE$ by Cost-sensitive Pattern Mining for $SeqLent$ Extraction; in which a non-buggy AST representations having $SeqLent$ less than minimum $SeqLent$ of buggy ones are dropped—with the aim to include and extract all information about the buggy files; and resulted in AST representations of 11,831. Given the defect rate as represented in Figure 4.14, optimal $SeqLent$ with higher coverage and standard deviations results as indicators for our analysis are presented in Table 4.4.

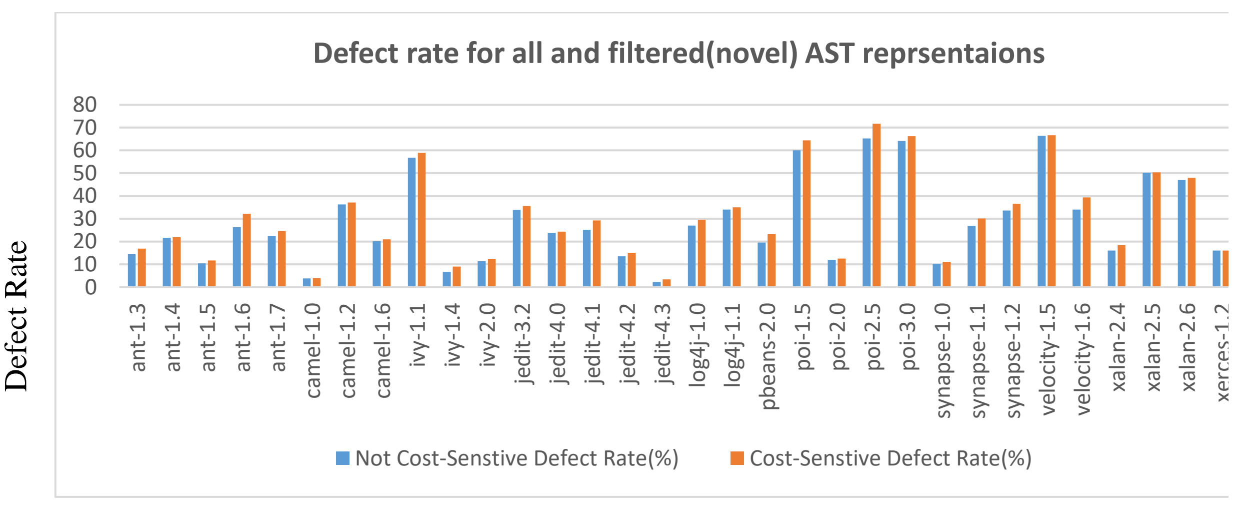


Figure 4.14: Bar chart of cost-sensitive bug Rates

Table 4.4: Experimental results for sequence length of mapped and filtered ASTs

AST Representations	No Files	Bug Rates	Min	25%	50%	75%	90%	95%	98%	Std. Dev
$Mapped_{Sequences}$	13885	34.41	1	34	95	225	487	773	**1333**	436.31

$Filtered_{Sequences}$	11831	**36.01**	1	45	110	255	533	853	1480	**461.58**

Table 4.4 shows the resulted file-level sequence lengths with coverage of 25% up to 98% defect dataset instances of all mapped and cost-sensitive/filtered AST representations. The observation from this this experiment shows the standard deviation and maximum sequence length were increased from 436.31 to 461.58, and 1,333 to 1480 respectively by using 98% our cost-sensitive AST representations. This implies that the classification accuracy is improved by 2.53% with the overhead/cost of 1,480 as maximum sequence length (truncation limit) for semantic features learning. For reference, full list of results are provided in Appendix C. 5.

4.2 Matched Software Defect Dataset (CDD) Quality Analysis (RQ_1)

The research questions were answered based on the results from experiment1. This included the modeled defect dataset quality and evaluation of defect prediction model performance using different features of source code. Below are the answers to the asked research questions. For reference, a full list of results for all experiments is presented in Appendix C.

RQ_1: How many software defect datasets of the PROMISE repository are mapped with the modeled source code files of the java software systems?

Different real-world issues related to software defect dataset modeling in the context of file-level, are included with and raised as the above ask of RQ_1. So in order to assess these issues and drive a precise answer(s) for RQ_1, we have sub-divided RQ_1 as the following questions with the answer(s) presented herein:

[RQ_(1.1)], How match the percentage of the real-world source code files of software systems are utilized as defect instances in the PROMISE software defect dataset?;

As detailed in section 4.1, Experiment-1.1 tested the existence of difference (i.e. ratio) between all of the obtained java source code files and PROMISE defect instances of target software systems from our python tools, which are. This experiment resulted in a total of 14,396 PROMISE defect instances as software defect datasets and 23,989 java source code files of target software systems. The results showed 9,593 java source code files are not to be used for supervised defect prediction. The obtained results showed that only 57.88% of java source code files of target software systems are mapped as PROMISE defect datasets for supervised defect prediction.

[RQ_(1.2)] Did the modeled source code files as matched defect datasets from the proposed file-level data modeling framework vary from PROMISE defect datasets?

Yes. The mapped source code files as defect datasets extracting module/tool called ExtractMappedInstances(), Experiment-1.2, extracted and mapped 13, 885 java source code files as software defect dataset, and dropped 470 instances of PROMISE defect dataset. The file-level defect data modeling module of experiment-1.1 achieved an accuracy of 96.45%.

[RQ_(1.3)] Where some defect datasets of the PROMISE repository never modeled as mapped source code files?

The pattern matching/analysis module using the dropped PROMISE defect instances of RQ_1.2 from five software projects as experiment-1.2.1 resulted in 64 not java files and the remains are recorded as not existed files. The obtained results of experiment-1.2.1, show the proposed pattern matching method improved the software defect dataset quality of 3.39% by dropping 0.07% not java files and 3.32% not existed files (i.e. updated by the third party).

[RQ_(1.4)] Did the obtained defect dataset from the proposed framework improve the rate of defects compared to the PROMISE defect dataset?

The analysis of class imbalance using the obtained software defect dataset PROMISE repository and RQ_1.2 as experiment-1.2.2, and resulted in defect rates of 31.41% and 34.44%, respectively. This result showed the improvement of software defect dataset class imbalance by 3.01%.

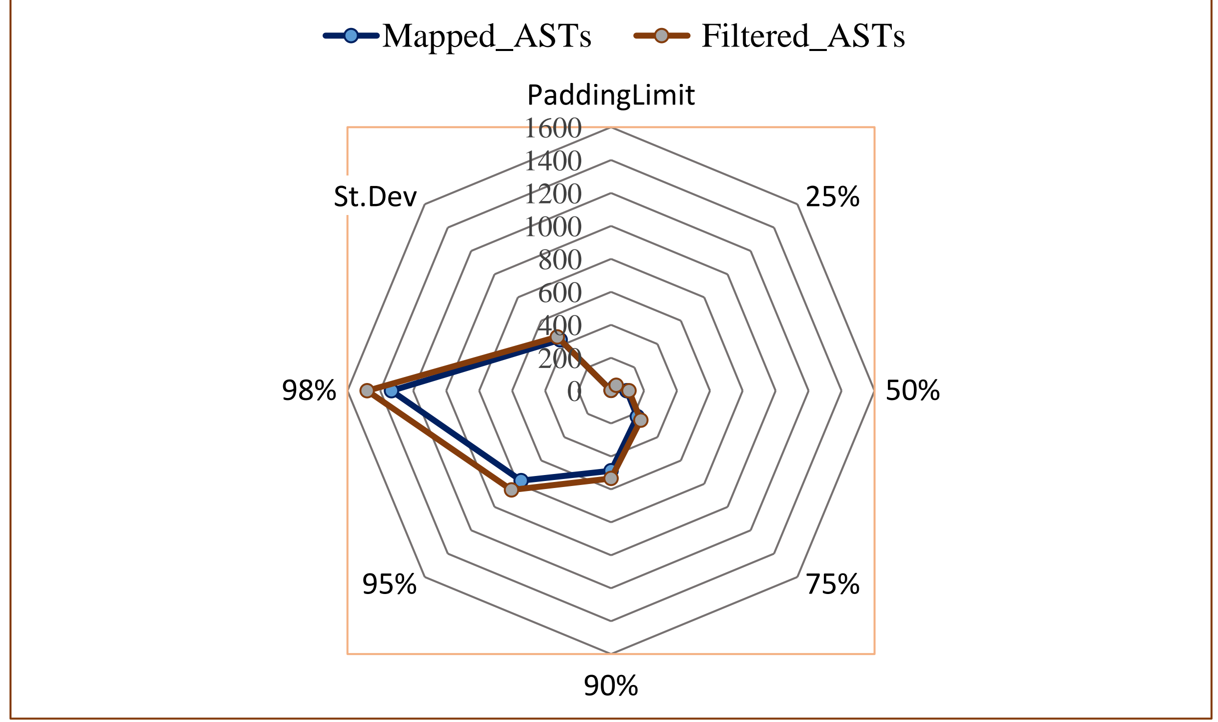


Figure 4.15: Radar charts of optimal $SeqLent$ evaluation

[$RQ_{1.5}$] What is the maximum sequence length for modelling AST representations of source code files as context-based datasets to be used for defect prediction tasks?

As detailed in section 4.1, Experminet-1.3 tested an optimal sequence length of source code's AST representations using datasets from $RQ_{1.2}$ and a new dataset called cost-sensitive (filtered) datasets from ExtractOptimalSequenceLenth() module as presented in Table 4.1. As shown in Figure 4.15, this experiment resulted in an optimal sequence length of 1480 with standard deviation of 461.58 and coverage of 98%.Overall, the effectiveness of all defect instance of PROMISE repository, and our mapped source code files on the performance of defect prediction model is provided in Table 4.7.

4.3 Evaluations of File-level Software Defect Prediction Approaches

In this section, the performance of the defect prediction model is tested using obtained file-level defect datasets of experiment 1. In this section, precision (P), recall (R), and F1-score (F1), and AUC are adopted as the performance evaluation parameters. The values of precision, recall, and F1-score range between 0 and 1, with higher values indicating better performance (Min Zhang, Guohua Geng, and Jing Chen, 2020). The experimental dataset setting for this section is presented as Table *4.5*, which describes the defect prediction methods, predictive features type called defect data information, and details of datasets.

Table 4.5: Experimental data setting of file-level defect prediction

Defect prediction using:	SDP approach	Predictive features type	Dataset Description
Metric attributes of all files	Metric-based	Heterogeneous information	All source code files with hand-crafted metric features of software projects.
Metric-features of all unique files	Metric-based	Homogenous information	All java source code files with metric features of a software system
semantic features of AST representations	Contextual-based	Homogenous information	All java source code files with AST representations of a software system

Metric and semantic features of matched files	Combined features-based	Matched and homogenous information	All source code files with metric and tokenized features i.e. semantic features of a given software system

4.3.1 Baseline approaches

To assess the effect of using the source code metric features, contextual features and both of them as combined features on the performance of defect prediction models, we compared the following defect prediction approaches:

1. An approach of using static code metric attributes of all files of a given software system alone as *base metric features* (BMF) to build defect prediction model F as F_{BMF}.
2. An approach of using all unique source code metrics (i.e. unique source code indicates files with the same file type e.g. '.java') of a software system alone as matched *metric features* (MF) to build F(i.e. predictive model), which is represented as F_{MF} .
3. An approach of using all software systems and mapped AST representations of source codes obtained AST tokens alone as matched *semantic features* (SF) to train language model (L) for defect prediction tasks.
4. An approach using all software systems and matched source code files obtained hand-crafted code metrics and $SeqLent$ as *hand-crafted features combination* (HFC) to build the defect prediction model using combined features and represented as F_{HFC} .

4.3.2 Baseline Models

We used a random forest classifier (RF) as our baseline predictive model ($\boldsymbol{F}$) with the parameter setting of class weight = 'balanced, bootstrap=True, and number of estimators=8000. The reasons for choosing the RF classier as our baseline defect prediction model is as follows. First SDP performance is improved by combined different estimator's decisions of RF (Ashima Kukkar et al., 2019). Second, the randomness learning strategy of RF deals with the overfitting problem of defect prediction models and improved the performance of SDP (Tianchi Zhou et al., 2019). The Bi-LSTM model is used as baseline language model ($\boldsymbol{L}$) for deep semantic feature extraction and its components are shown in Table 13, which describes the layers, shape of output, and parameters.

Table 4.6: Experimental Parameter setting for the language model

Layer (type)	Output Shape	Param #	Description

input_1 (Input Layer)	[(None, 1500)]	0	In this experiment, we have adopted Bi-LSTM model (Guanjun Lin, 2020) as our language model by doing modification on some of its function-level parameters setting to the context of file-level i.e. our targeted defect prediction granularity, which includes input size to 1,500, word embedding to the size of vocabulary.
embedding (Embedding)	(None, 1500, 100)	2400	
bidirectional (Bidirectional	(None, 1500, 128)	84480	
dropout (Dropout)	(None, 1500, 128)	0	
bidirectional_1 (Bidirectional	(None, 1500, 128)	98816	
global_max_pooling1d(Global	(None, 128)	0	
dropout_1 (Dropout)	(None, 128)	0	
dense (Dense)	(None, 64)	8256	
dense_1 (Dense)	(None, 32)	2080	
dense_2 (Dense)	(None, 1)	33	

Note: Total parameters: 196,065, Trainable parameters: 193,665, and Non-trainable Parameters: 2,400

4.3.3 Obtained Results and Analysis

Code Metric Information: Using static code metric attributes of any source code files (i.e. file's with different types of file extensions—e.g. '*.java*') of a software system called base metric features (BMF) to train RF classifier as defect prediction model (F), the results for predicting files defect proneness on latest (i.e. N^{th}) version by training the F as F_{BMF} on the N-1 versions of software systems shown in Table 4.7 of F_{BMF}. The model resulted in an average accuracy of 65.97%, F1 score of 66.81% with precision 76.60%, recall 65.97% and AUC of 69.73%. Similarly, using matched source code's metric attributes (i.e. the matched source codes indicates files with unique file type, e.g. '.java' extension) of a software system called matched *metric features* (MF) to train F, the results for predicting file defect proneness on the latest version by training the model on the set of oldest versions shown in Table 4.7 of F_{MF}. This defect prediction model resulted in an average accuracy of 65.46%, F1 score of 66.53% with precision of 76.59% and recall of 65.46%, and AUC of 68.32%.

Table 4.7: Experimental results for prediction models trained in source code metrics

SDP Models	F_{BMF}					F_{MF}				
Train sets → Test set	Accuracy	Precision	Recall	F1_ score	AUC	Accuracy	Precision	Recall	F1_ score	AUC
ant-1.3, 1.4, 1.5, 1.6 →1.7	80.43	78.40	80.43	78.64	79.76	**81.08**	**79.28**	**81.08**	**79.44**	**80.64**
camel-1.0, 1.2, 1.4 →1.6	77.91	75.58	77.91	**76.49**	70.39	**77.92**	**75.59**	**77.92**	76.46	**71.08**
ivy-1.1, 1.4 → 2.0	80.63	83.32	80.63	81.86	76.73	**80.68**	**83.34**	**80.68**	**81.90**	**76.77**
jedit-3.2, 4.0, 4.1 ,4.2 →4.3	**81.5**	**96.69**	**81.5**	**87.93**	**64.84**	80.45	96.63	80.45	87.26	64.19
log4j-1.0,1.1 →1.2	**33.76**	88.22	**33.76**	**43**	**67.12**	30.21	**90.66**	30.21	42.54	53.02
lucene-2.0 ,2.2 →2.4	**62.18**	**62.82**	**62.18**	**62.41**	**67.61**	61.82	62.12	61.82	61.95	65.82
pbeans-1.0 →2.0	43.14	85.42	43.14	44.41	**71.76**	43.14	85.42	43.14	44.41	71.56
poi-1.5, 2.0, 2.5 →3.0	**66.88**	**70.93**	**66.88**	**67.49**	**73.95**	66.12	70.51	66.12	66.81	73.05
synapse-1.0, 1.1 →1.2	**68.13**	**65.59**	**68.13**	**62.05**	**68.29**	66.57	62.35	66.57	59.64	67.6
velocity-1.4, 1.5 →1.6	55.99	69.8	55.99	55.74	63.08	**56.68**	**70.81**	**56.68**	**56.41**	**63.18**
xalan-2.4, 2.5 →2.6	69.13	69.08	69.13	69.08	**75.62**	**69.3**	**69.26**	**69.3**	**69.24**	75.29
xerces-1.0,1.2 → 1.3	**71.96**	**73.32**	**71.96**	**72.63**	57.55	71.57	73.15	71.57	72.34	**57.61**
Average (%)	**65.97**	**76.60**	**65.97**	**66.81**	**69.73**	65.46	76.59	65.46	66.53	68.32

Legend: *F*, Bold and Underline indicates the random forest classifier, best, equal results, respectively.

Contextual Information: Using contextual information like synthetic and semantic information of matched source code's AST representations as matched semantic features (SF) representation

as shown in Figure 4.16 to train Bi-LSTM language model (L) for defect prediction task (L_{SF}), and deep semantic features (SF) representation extraction, the results for predicting file defect proneness on the latest (i.e. N^{th}) version by training the L on the N-1 versions of a software system shown in Table 4.8 of L_{SF}. The model achieved an average accuracy of 61.74%, F1 score of 52.93% with precision of 57.57% and recall of 61.74%, and AUC of 48.9%. Similarly, using all deep features representation of L alone as prediction features—e.g. SF—to train conventional ML model, the results for predicting file defect proneness on the latest version by training the RF model as F_{SF} on the set of oldest versions of a software system is shown in Table 4.8 of F_{SF}. The model achieved an average accuracy of 66.12%, F1 score of 67.02% with precision of 76.59% and recall of 66.12% and AUC of 70.42%.

Table 4.8: Prediction results using matched source code semantic features

SDP Models	L_{SF}					F_{SF}				
Train sets → Test set	Accuracy	Precision	Recall	F1_ score	AUC	Accuracy	Precision	Recall	F1_ score	AUC
ant-1.3, 1.4, 1.5, 1.6 →1.7	77.6	60.21	77.6	67.81	26.42	80.84	79.07	80.84	79.34	78.63
camel-1.0, 1.2, 1.4 →1.6	79.87	63.79	79.87	70.93	38.6	79.01	77.09	79.01	77.81	71.83
ivy-1.1, 1.4 → 2.0	88.64	78.56	88.64	83.3	19.18	86.36	83.78	86.36	84.83	79.92
jedit-3.2, 4.0, 4.1 ,4.2 →4.3	97.74	95.53	97.74	96.62	51.78	82.96	96.69	82.96	88.8	79.7
log4j-1.0,1.1 →1.2	6.91	95.93	6.91	5.53	69.17	30.85	90.78	30.85	43.31	50.49

lucene-2.0 ,2.2 →2.4	63.33	69.43	63.33	51.78	64.9	55.76	55.5	55.76	55.62	59.45
pbeans-1.0 →2.0	19.61	3.84	19.61	6.43	70.49	47.06	85.69	47.06	49.27	72.93
poi-1.5, 2.0, 2.5 →3.0	68.72	67.51	68.72	67.6	71.33	64.84	68.2	64.84	65.55	72.39
synapse-1.0, 1.1 →1.2	66.41	44.1	66.41	53	31.42	68.75	67.35	68.75	61.97	75.7
velocity-1.4, 1.5 →1.6	34.06	11.6	34.06	17.31	63.93	57.21	71.09	57.21	57.02	63.15
xalan-2.4, 2.5 →2.6	53.03	28.12	53.03	36.75	42.19	67.54	67.69	67.54	67.57	75.55
xerces-1.0,1.2 → 1.3	84.98	72.21	84.98	78.08	37.41	72.2	74.1	72.2	73.12	65.24
Average (%)	61.74	57.57	61.74	52.93	48.90	**66.12**	**76.42**	**66.12**	**67.02**	**70.42**

Legend: *F* and Bold indicates the random forest classifier and best results, respectively.

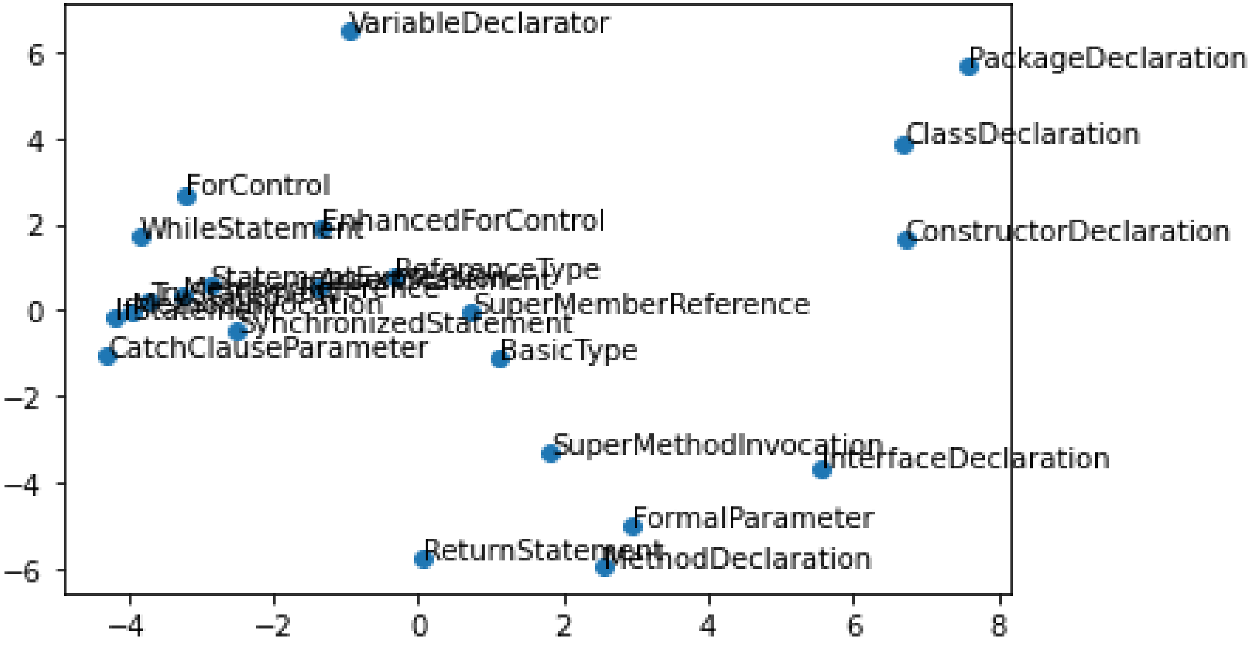


Figure 4.16: Visualization of AST Node types (words) representation as semantic features

Combined Information: Using the all matched source code files of a java project and obtained metric information and context information as the combined features to train RF classifier in the following three seniors:

[1]: By using all of the java projects and mapped source codes, and obtained MFs and $SeqLent$ as HCFs train defect prediction model (F), the results for predicting file defect proneness on the latest version of the software systems is presented in F_{HCF} of Table 4.9. The model achieved an average accuracy of 66.15%, F1 score of 67.3% with precision 77.07%, recall 66.15% and AUC of 68.87%. Similarly, using the concatenation of MFs and deep representation—SFs of all java source codes as combined features ($ConcatF$) to train F, the results of $F_{ConcatF}$ for predicting the defect proneness of source codes on the latest versions of target software systems is shown in $F_{ConcatF}$ of Table 4.9. This model achieved an average accuracy of 66.34%, F1 score of 67.25% with a precision of 76.19% and recall 66.25%, and AUC of 70.93%.

Table 4.9: Prediction results using combined information from source codes

SDP Models	F_{HCF}					$F_{ConcatF}$				
Train sets → Test set	**Accuracy**	**Precision**	**Recall**	**F1_ score**	**AUC**	**Accuracy**	**Precision**	**Recall**	**F1_ score**	**AUC**
ant-1.3, 1.4, 1.5, 1.6 →1.7	80.97	79.16	80.97	79.35	80.99	82.05	80.5	82.05	80.55	79.8
camel-1.0, 1.2, 1.4 →1.6	79.62	77.2	79.62	77.94	70.59	78.48	76.35	78.48	77.15	72.63
ivy-1.1, 1.4 → 2.0	81.87	84.03	81.87	82.87	76.58	87.5	85.56	87.5	86.31	82.36
jedit-3.2, 4.0, 4.1 ,4.2 →4.3	79.06	96.6	79.06	86.39	73.63	83.57	96.45	83.57	89.15	78.82
log4j-1.0,1.1 →1.2	30.85	90.78	30.85	43.31	48.74	32.45	91.04	32.45	45.18	48.82
lucene-2.0 ,2.2 →2.4	59.27	59.69	59.27	59.45	63.87	57.58	57.58	57.58	57.58	60.87
pbeans-1.0 →2.0	47.45	85.72	47.45	49.73	73.85	45.1	79.93	45.1	47.87	70.49

poi-1.5, 2.0, 2.5 →3.0	68.31	72.14	68.31	68.96	73.87	62.79	65.65	62.79	63.51	70.91
synapse-1.0, 1.1 →1.2	67.5	64.51	67.5	60.27	67.46	68.75	67.7	68.75	61.56	79.44
velocity-1.4, 1.5 →1.6	57.38	71.84	57.38	57.09	62.81	56.33	70.63	56.33	56.01	65.09
xalan-2.4, 2.5 →2.6	69.01	68.96	69.01	68.94	73.32	67.54	67.67	67.54	67.57	75.25
xerces-1.0,1.2 → 1.3	72.51	74.19	72.51	73.32	60.66	73.99	75.21	73.99	74.58	66.63
Average (%)	66.15	**77.07**	66.15	**67.3**	68.86	**66.34**	76.19	**66.34**	67.25	**70.93**

Legend: *F* and Bold indicates the random forest classifier and best results, respectively.

[2]: By using all of the oldest versions of the target system and mapped source code file's obtained deep features representation from LHFR as DL-combined features i.e. hybrid features (HybridF) to train defect prediction model F with and without class weighting sachem for resolving class imbalance problem, which is detailed in section 3.4.1.3. The average results for predicting defect proneness on the latest version of software systems are shown in Table *4.10*, which presents results of balanced, and cost-sensitive defect prediction models trained in features representations by the proposed combined DL model. The analysis for results obtained in this section is presented in Section 5.1. For reference, full list of results for this section is provided in Appendix C.5.

Table 4.10: Results for defect prediction using unified features representation

Performance sensitivity analysis under different Feature Combination, and parameter settings							
Features Representation Extraction components of our DL model (SDP_{CMSF})		Prediction model ($F_{HybridF}$) with CI learning strategy of:	Evaluation Metrics				
Sub-Component	N^{th} layer representation with DIM of:		Accuracy	Precision	Recall	f_1-score	AUC
Contextual Model (*Bi-LSTM*)	9th layer with features DIM of 128	Balanced	**65.8**	**75.67**	**65.8**	**66.73**	70.44
		CS	65.79	75.6	65.79	66.67	**70.5**
Metric Model		Balanced	**66.3**	77.35	**66.3**	68.13	68.97

(*MLP*)	10th layer and resulted in a features DIM of 15	CS	66.29	**77.44**	66.29	**68.15**	**69.02**
Dense Model (*FCN*)	11th layer and resulted in a ***concatenated features*** sets of 143	Balanced	66.25	76.02	66.25	67.1	**70.94**
		CS	**66.34**	**76.19**	**66.34**	**67.25**	70.93
	12th layer with 143 features representation	Balanced	**66.32**	**76.03**	**66.32**	**67.18**	**70.96**
		CS	66.22	75.98	66.22	67.09	70.93
	13th layer, resulted in a ***unified features*** representation with DIM of 64	Balanced	**67.57**	**77.31**	**67.57**	**69.08**	69.8
		CS	67.48	77.29	67.48	69	69.8
	14th layer resulted in a features DIM of 32	Balanced	67	77.07	67	68.77	69.57
		CS	**67.13**	**77.18**	**67.13**	**68.89**	**69.59**
Combined Model		Balanced (average)	66.54	76.56	66.54	67.83	70.11
		CS(average)	66.54	**76.61**	66.54	**67.84**	**70.13**

Legend: Bold, CI, and $HybridF$ indicates the best result, layers of a combined model, class imbalance, and combined features representation of Nth layer respectively.

Chapter five

5 Discussions, Limitations, and Conclusion

5.1 Discussions

We next present and analyses our experiment results with the aim of answering research questions (RQ2- RQ4). We detail collected data that provide foundations for our observations, and the complete obtained results are available in Section 4.2.3. Results presented in this section refer to the average values of the evaluation indicators for each defect prediction model in our context. To be specific, this section concerns with the analysis and discussions of obtained results of accuracy, precision, recall F1-Score/measure and AUC indicator) from a defect prediction model(s) trained in software information type(features set) of (I) source code metric features (Metric Information), (II) source code semantic features (Contextual Information), and (III) combination of source code metric and semantic features (Combined Information) as presented in Table *5.1* .A, B, and C respectively. For reference, the detailed collected data that provide foundations for our observations on RQ_1 as detailed in section 4.1.2 are provided in Appendix C.2 - C.5. Please note that results for RQ_1 are not shown here because RQ_1 not relates to defect prediction performance comparison, instead relates to the quality of defect data modeling.

Table 5.1: Defect prediction results using source code metric, and/or semantic features

Building Defect Prediction Models using Source Code:									
Evaluation	Metric Features (A)			Semantic Features (B)				HFC (C)	
Metrics(%)	F_{BMF}	F_{MF}	%(Δ)	L_{DSF}	F_{DSF}	%(Δ)	%(Δ)	$F_{MF+SeqLen}$	%(Δ)

Accuracy	65.97	65.46	-0.52	61.74	**66.12**	+4.38	+0.66	**<u>66.15</u>**	+0.03
Precision	76.60	**76.59**	-0.01	57.57	76.42	+18.85	-0.17	**<u>77.07</u>**	+0.65
Recall	65.97	65.46	-0.51	61.74	**66.12**	+4.38	+0.66	**<u>66.15</u>**	+0.03
F1_Score	66.81	66.53	-0.21	52.93	**67.02**	+14.09	+0.49	**<u>67.3</u>**	+0.28
AUC	69.73	68.32	-1.65	48.90	**<u>70.42</u>**	+21.52	+2.10	**68.86**	-1.56

Legend: $HFC, Bold, and\ Bolded\ pluse\ Underline$ indicates the Hand-crafted Features Combination, best results, and winner defect prediction approach respectively.

5.1.1 Individual Information Effectiveness (RQ_2.)

To investigate the effectiveness of each individual source code features set (information), including those proposed in the identification of defect-prone files, we analyzed results obtained with each feature set containing metric information and contextual information. Table 5.1 details average results obtained with source code metrics and semantic features set ordered from left to right order according to a performance improvement (change) indicator symbolized by Δ. When results change from high to low performance or the revers one, we use + or- and then the ranking/measure of change obtained defect prediction models (from left to right) with respect to given indicators, respectively. This analysis helps evaluate different models with respect to each indicator and contribution of each feature set i.e. information of source code for defect prediction performance improvement in file-level context.

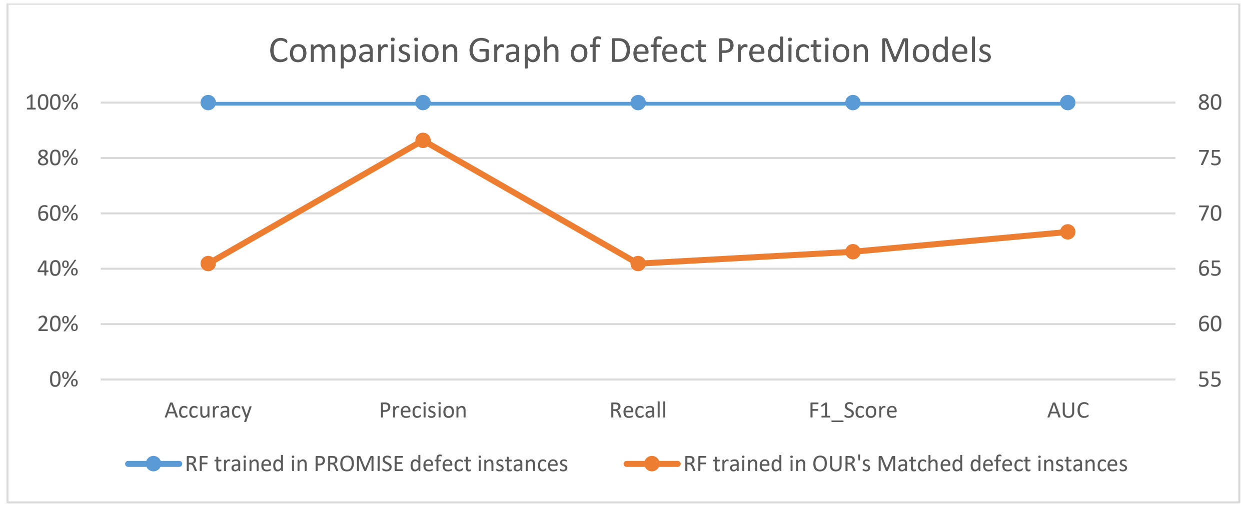
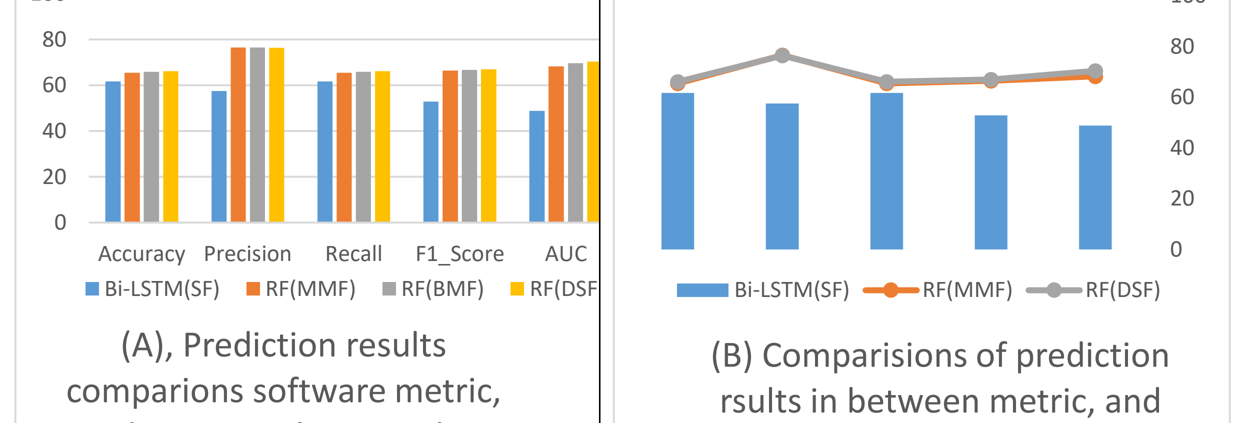


Figure 5.1: Stacked lines comparing the effectiveness of Matched and PROMISE code metrics

The analysis for the source code metric features-based defect prediction approach is shown in Table *5.1*.A. Figure 5.1 presents the average results of defect prediction models (F_BMF, and F_MF)

trained in PROMISE and our matched defect datasets, respectively. As expected, results of F_MF are low when compared to F_BMF, because the quality of matched defect datasets is likely affected by the dropped unactionable datasets as detailed in Section 4.1.2, this implies that metric features of mapped source code(obtained from CDDM) not enough to correctly distinguish non-defective files and defective ones. Interestingly, this is more evident in recall (65.97- 65.46) than in precision (76.6-76.59) and accuracy (65.97 vs. 65.46), with a change of -0.51 and -0.52, respectively. This may occur because certain metric values of mapped files are affected—e.g. due to CRUD (Create, Remove, Update, and Delate) operations by the third party and drooped relation in between mapped java files and unactionable not java files—lead to the introduction of high and low metric values for non-buggy and buggy mapped files, respectively. Therefore, a classification algorithm may be unable to classify such files as defect-prone or not based solely on the information of the source code metric suite. This emphasizes the need for contextual features for building a highly accurate defect prediction model.

Table 5.2: Comparison of defect prediction using source code metric and semantic features



$\boldsymbol{RQ_{2.2}}$: With respect to the performance of defect prediction models trained in source code context representations—e.g. semantic features visualized in Figure *4.16*, in comparison with MFs—as presented in Table *5.1*.A, and B. The analysis of the results depicted within Table *5.1*.B indicates that conventional ML model (i.e. random frost classifier) trained in DL-generated SFs represented by SF performed better when compared to MFs-based models which are provided in Table *5.1*.A. On the other hand, the results of the SFs-based DL model i.e. language model for defect prediction resulted in low performance when compared to the ML models trained in MFs and SFs

representation as presented in Table *5.2*.A. This implies that the language model is not fit for predicating defect-proneness, having overall performance indicators values below results of conventional ML models.

Given that the source code context representations deep SFs obtained from the language model achieved better results, a contextual information-based approach would be to believe that a context representation containing sequential and semantic features would achieve better results. Obtained an improvement in the results of accuracy, F1 Score, recall, and AUC by +0.66, +0.49, +0.66, and +2.10 respectively. However, none of these initially investigated SFs achieved precision better than those obtained with MFs alone, having values of (76.42 vs. 76.59). Therefore, we can conclude that contextual information alone—e.g. SFS is a weak predictor of defect proneness in terms of precision when compared to source code metric features. This emphasizes the need for a combination of source code metric and contextual features combination—e.g. hand-crafted features combination, DL-generated features combination.

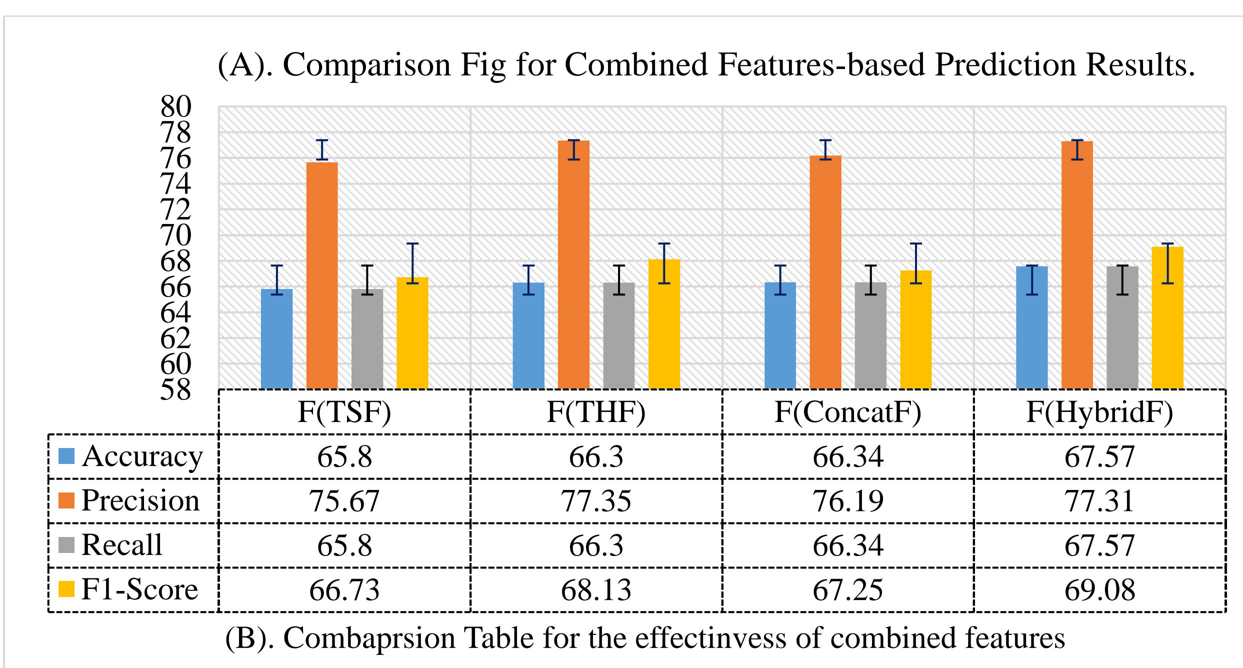
5.1.2 Combined Information Effectiveness (RQ_3)

Regarding hand-crafted features combination (HFC) which includes static source code metric features with our proposed hand-crafted contextual feature called *RTF* (highlighted in Table 19.C), totally 21 hand-crafted features set. It can be seen that HFC achieved high results in comparison with the MFs set in terms of all indicators, and also fellfield the gaps of best performing SFs-based model, having precision values of precision (77.07 vs. 76.42. Considering f-measure, which balances precision and recall, our proposed contextual feature i.e. *RTF* with MFs obtained results that are only not better than the hand-crafted MFs but are better than DL-generated SFs. HFC presented slight improvement in F1-score of +0.28, accuracy and recall values of +0.03, and relatively high improvement in precision values of +0.65. This gives evidence that our proposed hand-crafted contextual feature with metric features can indeed be useful to predict bugs in our context. However, we observed that results of AUC, have a high standard deviation and the worst result for HFC. This may be explained by project-specific feature representation. For example, if source code's patterns i.e. AST representations are taken into account by the developer of a certain project, this feature will likely be very helpful to identify fault-prone files in that project which emphasizes the need for DL-generated features representation. We thus next investigate all

possible DL-based features combination and representations, focusing on those that obtained the best results.

As explained in our study procedure, the number of all possible feature combinations considering HFC with 21 and SF with 128 features, a total of 159 features set for unified features representation learning is extremely large. Therefore, we transformed and combined each individual features set into a unified features representation while training the proposed LHFR model to be analyzed, presented in Table *5.3*.A, and B, split into three groups. The DL-generated features representation composed of: (I) the HCFs representation named THF by Transformed Hand-crafted features; (II) the non-linearly Concatenated THFs and SFs Representation named as ConcatF, (III) the non-linearly unified features representation of concatenated features named HybridF by Hybrid Features Representation. For each combination of features, we present not only the average precision, recall, and f-measure obtained across the different projects, but also the dimension of the features present in the respective representation.

Table 5.3: Evaluation of defect prediction performance using d/t features combination



	F(TSF)	F(THF)	F(ConcatF)	F(HybridF)
Accuracy	65.8	66.3	66.34	67.57
Precision	75.67	77.35	76.19	77.31
Recall	65.8	66.3	66.34	67.57
F1-Score	66.73	68.13	67.25	69.08

(B). Combaprsion Table for the effectinvess of combined features

Note: ┬, and ─ on top of the bars indicates the relative improvement and improved results, respectively.

Large improvements can be seen in the DL-combined and unified features i.e. non-linear features combination and transformation—e.g. hybrid features, with respect to the combined features sets discussed above and HF. For all individual feature sets, which are 159 in total as there are 15, 143, and 64 DL-generated features for each of THF, ConcatF, and HybridF respectively, results are improved not only considering the measurement that was used for the combination of the hand-crafted features, but also the other measurements. While, the DL-generated combined features representation by considering the non-linear combination and transformation of the source code metrics with *RTF* feature (i.e. hand-crafted features set) and semantic features, referred as *HybridF* were resulted in a lower AUC value when compared to *THFs* set alone and combination of THFs and SFs set named *ConcatF* by *C*oncatenated *F*eatures set (69.08 vs. 77.31 & 70.93, respectively) as presented in Table 18. However, obtained results can be perceived in each of the measurements—gains are at least +1.23 of accuracy and recall, +1.23 of precision, and +1.83 of F1-score when compare to the concatenation of the DL-generated source code metric, and semantic features set. While the DL-based features transformation alleviated the problem of high dimension of combined features set for defect classification, the AUC of the prediction was not improved when comparted to concatenated features set

With respect to the lower result obtained associated with the AUC measurement, we clarify that this is expected because DL-based features transformation with non-linear dimensionality redaction criteria, other than concatenation, do not aim at maximizing the probability of buggy instances or keeping maximum features. While the DL-based features transformation alleviated the problem of high dimension of combined features set for defect classification, the AUC of the prediction was not improved when compared with the concatenated features set which emphasize the need for a cost-sensitive defect classification (highlighted in Section 5.1.3). Additionally, there are software systems—e.g. '*log4j*' and '*velocity*' with f1-score of 45.18% and 53%, respectively—with poor results that emphasize the need for cross-project defect prediction sachems by incorporating more advanced class imbalance handling and features selection methods (Homona Jacob & Geetha Raju, 2017; Goeschel, 2019), with the LHFR model in the future will be fruit full.

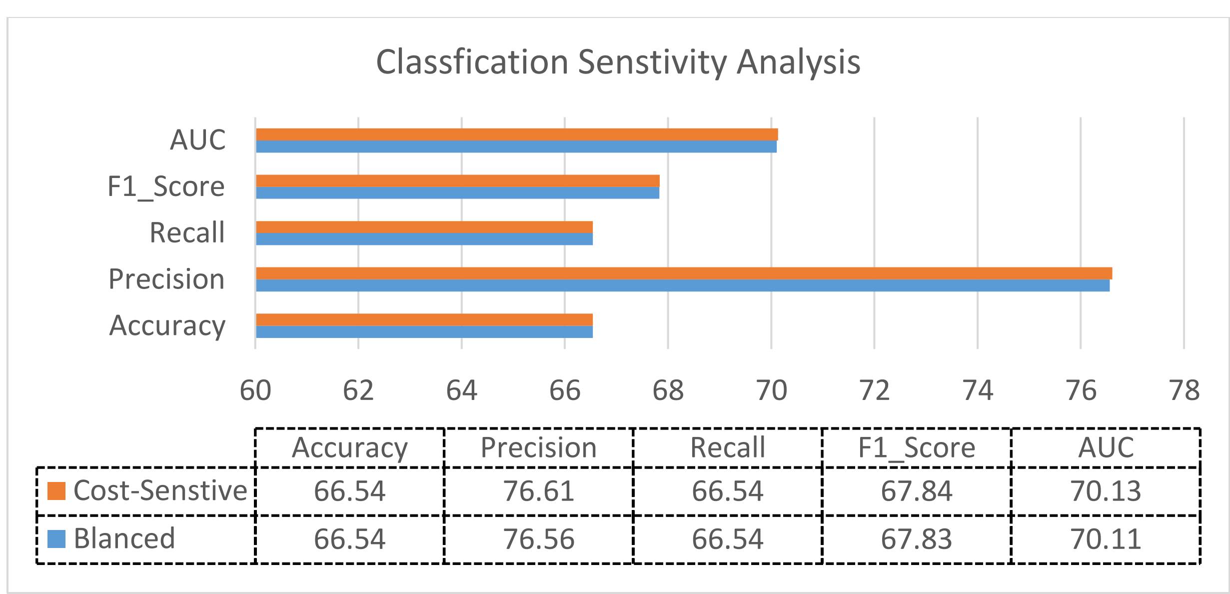


Figure 5.2: Prediction performance under different defect classification sachem

5.1.3 Prediction Performance Sensitivity Analysis (RQ_4)

With respect to the performance of combined features-based defect prediction models under different class weighting sachems as shown in Figure *5.2*; the defect prediction model trained in, cost-sensitive defect classification manner were performed slightly better than that of the balanced class weighs-based models in values of precision, f1-score and AUC. To be specific, while the cost-sensitive defect prediction model showed the improvement precision, f1-score, and AUC values by +0.05%, and 0.01% respective. However, the relative class weights-based (cost-sensitive) prediction model achieved the same performance when compared with to the balanced weights-based model in terms of both accuracy and recall measurements, having the same average prediction result of 66.54%. Therefore, depending on the goal of using a defect predictor, a features combination and representation that would better satisfy the goal should be adopted.

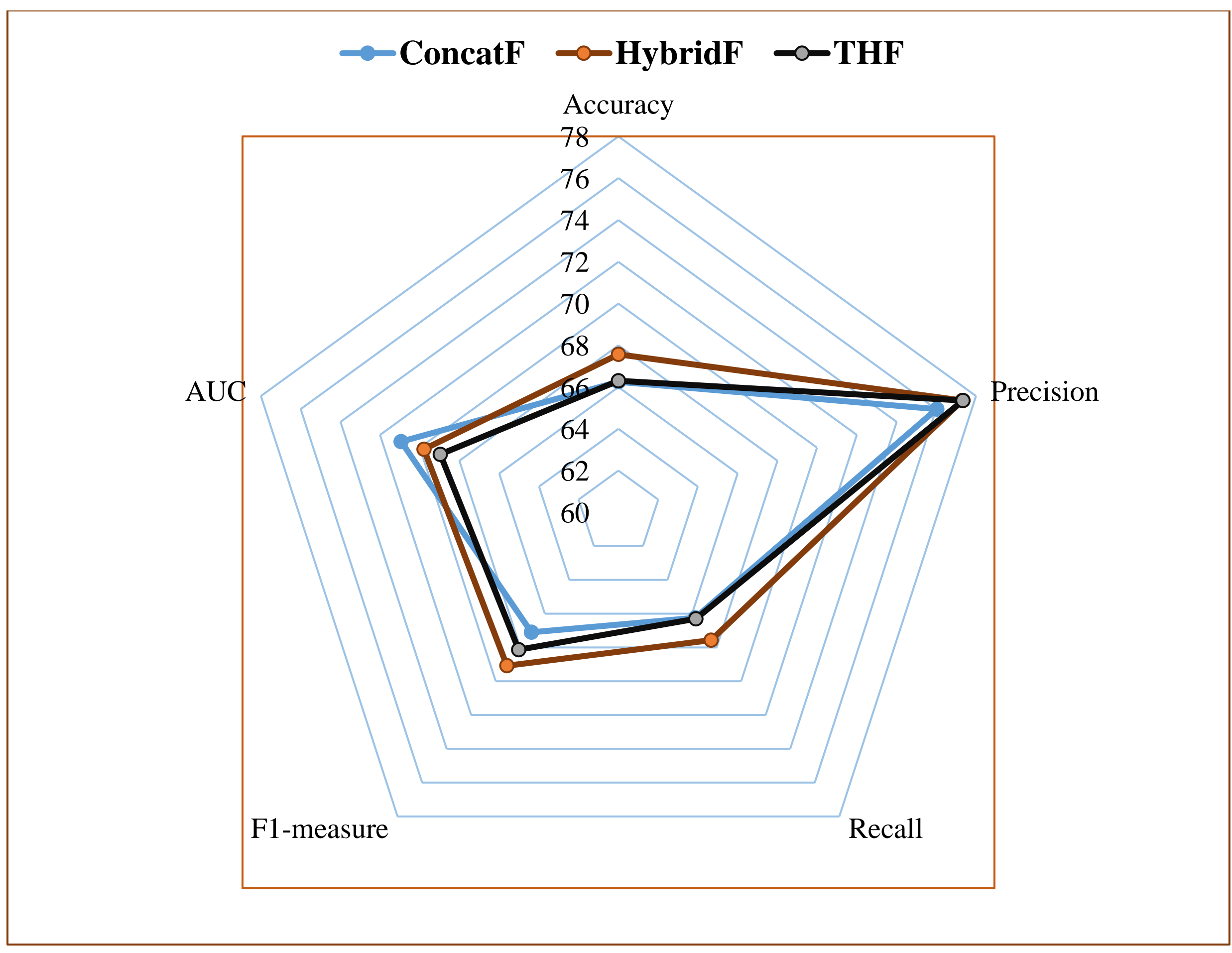


Figure 5.3: Radar charts for prediction indicators of LHFR model with 3 types of features

Figure 5.3 depicts the radar chart of average values of the five indicators for our LHFR framework under defect data (i.e. combined features) with three types of transformation. Figure 5.3 shows that the performance on combined features with hybrid features representation are very similar in terms of four indicators and they are higher than the performance on concatenated features and features with transformed features dimension in terms of four indicators (all except AUC), respectively. This implies that LCFR tends to achieve better AUC values on defect data with concatenated features dimension and obtains better other five indicator values on defect data with transformed features dimension/representation.

Overall, combined features were better features to be used than individual source code metric, and semantic features. The possible explanations stemming from this finding are: (I) For Java system, often source code with small metrics profiles (e.g. small lines of code) and context representations profiles(e.g. few AST tokens, small sequence length) does not necessarily mean high-quality code,

as often a context representation of code is used to preserve specific contextual information's (e.g. syntactic and/or semantic information) and small code is used to achieve specific non-functional requirements (e.g. code reusability) – in these respect individual features—e.g. metric features, and semantic features—may not be a good predictor of defect proneness. (II) Combined features (Hybrid information) obtained from source code metrics and contextual features—e.g. semantic and/or synthetic features— extracted from Bi-LSTM (i.e. language model) context representation may provide a better and more "amalgamated" view of the software system's health.

5.1.4 Threats to validity

External validity

The threats to external validity focus on the generalization of the experimental results (Zhou Xu et al., 2019). To mitigate these external threats, we choose a publicly available open-source benchmark dataset that was provided in a top journal in the software engineering domain (Zhou Xu et al., 2019). This dataset contains 42 software project versions, and static code metrics were collected for each project. Thus, conducting experiments on such a combined feature of a large dataset reinforces the representativeness of our conclusions. However, the projects and metric features studied were all developed with the Java language and product metric types. Thus, replication experiments on the projects developed with other programming languages (i.e. Python, C, and C++) with other metrics (i.e. Processes metrics), may prove fruitful.

Internal validity

The threats to internal validity concern the faults that may occur in the implementation of the studied methods (Zhou Xu et al., 2019). To minimize the internal validity, we implement our LHFR framework by using Keras API and pair programming in an attempt to avoid mistakes in programming. For the baseline models, we make full use of the methods available in the third-party libraries of Python and the source codes provided in previous studies. Future studies should explore state-of-the-art combined deep learning model's structure for unified features representation extraction. However, the implementations of such combined models are seldom available and our implemented versions may include differences and errors compared with the individual ones. Hence, we choose to compare against some off-the-shelf individual, combined, and unified models. For the effects of reducing combined code metric and semantic features

dimension into unified features dimension, we use non-linear features transformation (i.e. ReLU) of the DL model. This is a commonly used setting for combined features-based SDP studies.

Construct validity

The threats to construct validity concern the bias of the evaluation indicators used (Zhou Xu et al., 2019). In this study, we choose widely used traditional indicators to show the empirical evaluation of our LCFR framework for the SDP task. Other indicators, such as g-mean, Balance, and effort-aware indicators are not used, but we have reported their detailed results with our LHFR framework in this paper (within the future the online supplementary materials will provide) for future studies to compare.

Reliability validity

The threats to reliability validity concern the possibility of repeating our proposed framework. To enable further replication and comparison with results from future studies, we have made the used publicly available benchmark dataset and source code provided in this paper. The threats to reliability validity concern the possibility of repeating our proposed framework. To enable further replication and comparison with results from future studies, we have made the used publicly available benchmark dataset and source code provided in this paper.

5.1.5 Final Remark

In this chapter, we presented a study that evaluated the software metric and contextual features/information using the DL approach proposed in chapter 3. We found that these DL-combined and transformed source code metric and semantic features attributes, enhance the defect predictor performance in the java software systems. Next, we present a summary of our contributions and issues that shall be addressed in future work.

5.2 Conclusion

In this thesis, we report results from an evaluation study as to whether information obtained from software context and source code metrics can be used as training features in a machine learning framework so that the trained model can classify a file as one containing a defect/bug that can produce an error and ultimately a failure or not. The evaluation study focused on twelve large open-source systems written in Java. First, the result indicated that information (RTF) obtained from software context representation(i.e. AST) with static source code metrics and used as training features performed equal to or better than source code semantic features (DL-based AST tokens

representation) in all of the research questions. Second, in all of the scenarios, the prediction capability of the model was weak for Java systems when using individual information—e.g. source code metrics, and semantic features—alone as a training feature, while it is better when using combined information extracted from the source code metrics and context representation. Third, the evaluation study indicates that one could use unified training features—e.g. DL-generated features from hand-crafted and deep semantic features—irrespective of individual features combination to train the model, and still perform batter predictions. Finally, the study also indicates that this approach is input structure agnostic because the model was built using twenty-one hand-crafted features and twenty-nine AST tokens, but was extracted transformed features representations for training prediction models.

5.2.1 Summary of thesis findings

Given the results presented in this thesis, we list below our main findings:

In Chapter 4 we presented a comparison of defect prediction approaches using source code metric and semantic features. Based on our findings we determined which DL-combined features of source code metric, AST sequence length and semantic features named unified features perform best together with the associated cost-sensitive LHFR and ML prediction model. We also concluded that adapting a class weighting scheme improves defect prediction models for the evaluated context. A difference was also obtained in between sub-components (DL-generated features representation) of the combined model as described in Section 5.1.

Use code metric with hand-crafted contextual and semantic features as defect prediction features. We also proposed a combined DL model to use source code metrics and contextual features i.e. static sequence length of AST, semantic features to enhance defect prediction models for java software systems. The reasoning and details of these features are detailed in Chapter 3. A study, described in Chapter 4, was performed and we concluded that the proposed combined features (I) perform better for prediction than the baseline approach when used alone, and (II) enhanced the defect prediction performance for java software systems when used combined.

Importance of DL-based feature combination and transformation for java software systems defect prediction. Based on the study presented in Section 4.2, we found improvement of by more than eighty-three percent prediction performance when used the DL-generated unified features for the java system. Nevertheless, this unified representation may perform much worse for individual features (i.e. the remained two target systems). Hence for a specific software system with an

established dataset, performing a cross-project defect prediction by considering feature selection will enhance the software systems resulted in the worst prediction performance for combined features set, corroborating the literature that found this best results with the code metrics set (Yaning Wu et al., 2018), (homona Jacob and Geetha Raju, 2017).

5.2.2 Future works

The work presented in this thesis can be extended in three major directions:

The first direction deals with adapting our combined feature representation learning framework to cross-project defect prediction by combining transfer learning techniques. In addition to this, we plan to incorporate more information in our model in an attempt to understand the potential reasons for defective software modules by considering a semi-supervised deep learning model—trained in seldom labeled data and abundant unlabeled data to attain an effective performance. The second direction deals with expanding the training and the analysis by considering more software systems implemented in more programming language and more training features. The third direction deals with considering more advanced class-imbalance handling methods and more AST tokens in our combined feature representation learning framework, fulfill the limitations of AST tokens in distinguishing the defective files from non-defective ones as presented in Appendix C.6.

References

Alegre, P. (2017). Bug Prediction in Procedural Software Systems. CATALOGING-IN-PUBLICATION(Araujo, Cristiano Werner).

Alyahya, S. (2020). Crowdsourced software testing: A systematic literature review,. ISSN 0950-5849.

Ashima Kukkar et al. (2019). A Novel Deep-Learning-Based Bug Severity Classification Technique Using Convolutional Neural Networks and Random Forest with Boosting. *Sensors 2019, 19, 2964*, 10.3390/s19132964.

Aston Zhang et al. (2019). *Dive into Deep Learning, Release 0.7* (Vol. Release 0.7). https://discuss.mxnet.io/.

Bin Liu et al. (2019). Feature Generation by Convolutional Neural Network for Click-Through Rate Prediction. *International World Wide Web Conference Committee*, . https://doi.org/10.1145/3308558.3313497.

Brownlee, J. (2018). *How to Configure the Number of Layers and Nodes in a Neural Network*. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/

Cesar Couto et al. (2016). Predicting Software Defects with Causality Tests. *HAL*, hal-01086783.

Cong Pan et al. (2019). An Improved CNN Model for Within-Project Software Defect Prediction. *Appl. Sci. 2019, 9, 2138*, 10.3390/app9102138.

Dario Di Nucci et al. (2018). Dynamic Selection of Classifiers in Bug Prediction: an Adaptive Method. *University of Salerno, Italy — 2TU Delft, The Netherlands — 3University of Molise, Italy*, . http://www.apache.org.

David Wehr et al. (2019). Learning Semantic Vector Representations of Source Code via a Siamese Neural Network. *ICSE '16, May 14-22, 2016, Austin, TX, USA*, : http://dx.doi.org/10.1145/2884781.2884804.

Feng Zhang et al. (2017). Data Transformation in Cross-project Defect Prediction. *Empirical Software Engineering*, 10.1007/s10664-017-9516-2.

Ghanathey, S. (2018). Predicting Software Fault Proneness Using Machine Learning. *Electronic Thesis and Dissertation Repository*, 5936.

Goeschel, K. (2019). Feature Set Selection for Improved Classifcation of Static Analysis Alerts. *Part of the Artifcial Intelligence and Robotics Commons, and the Databases and Information Systems Commons* , https://nsuworks.nova.edu/gscis_etd.

Guanjun Lin. (2020). The Application of Neural Models for Software Vulnerability Discovery. *Swinburne University of Technology*.

Guisheng Fan et al. (2019). Software Defect Prediction via Attention-Based Recurrent Neural Network.

Haonan Tong et al. (2018). Software defect prediction using stacked de-noising auto-encoders and two stage ensemble learning.

Hoa Khanh Dam et al. (2018). Automatic feature learning for predicting vulnerable software components.

homona Jacob and Geetha Raju. (2017). Software Defect Prediction in Large Space Systems through Hybrid Feature Selection and Classification.

Hongliang Liang et al. (2019). A Semantic LSTM Model for Software Defect Prediction.

Jayanthi.R et al. (2017). A Review on Software Defect Prediction Techniques Using Product Metrics.

Jian Li et al. (2018). Code Completion with Neural Attention and Pointer Networks.

Jian Li et al. (2018). Software Defect Prediction via Convolutional Neural Network.

Jin Wang, et al. (2017). Combining Knowledge with Deep Convolutional Neural Networks for Short Text Classification.

Kang, N. (2017). *Multi-Layer Neural Networks with Sigmoid Function— Deep Learning for Rookies* . Retrieved from Towards Data Scince : https://towardsdatascience.com/multi-layer-neural-networks-with-sigmoid-function-deep-learning-for-rookies-2-bf464f09eb7f

Le Hoang Son et al. (2019). Empirical Study of Software Defect Prediction: A Systematic Mapping.

Maryam M Najafabadi et al. . (2015). Deep learning applications and challenges in big data analytics.

Maryam M Najafabadi, F. V. (2016). Deep Learning Techniques in Big Data Analytics. Big Data Technologies and Applications (pp.133-156).

Michele Tufano et al. (2018). Deep Learning Similarities from Different Representations of Source Code. *IEEE Transactions on Software Engineering*, DOI 10.1109/TSE.2018.2877612.

Michele Tufano et al. (2018). Deep Learning Similarities from Different Representations of Source Code. *IEEE Transactions on Software Engineering*, 10.1109/TSE.2018.2877612.

Miltiadis Allmanis et al. (2018). A Survey of Machine Learning for Big Code and Naturalness.

Min Zhang, G. G. (2020). Semi-Supervised Bidirectional Long Short-Term Memory and Conditional Random Fields Model for Named-Entity Recognition Using Embeddings from Language Models Representations. *Entropy 2020, 22, 252; doi:10.3390/e22020252*.

Ming Wen et al. (2017). How Well Do Change Sequences Predict Defects? Sequence Learning from Software Changes.

Mustafa Hammad et al. (2019). Predicting Software Faults Based on K-Nearest Neighbors Classification.

Nivetha.R, Kavitha.S,. (2019). Bidirectional Recurrent Neural Network Language Model: Cross Entropy Churn Metrics for Defect Prediction Modelling.

Okutan, Ahmet. (2018). Use of Source Code Similarity Metrics in Software Defect Prediction.

Rebeen Ali Hamad, M. K. (2020). Efcacy of Imbalanced Data Handling Methods on Deep Learning for Smart Homes Environments. *SN Computer Science 1:204*, 2 of 10.

Rudolf Ferenc et al. (2020). Deep learning in static, metric-based bug prediction.

Shaojian Qiu et al. (2019). Transfer Convolutional Neural Network for Cross-Project Defect Prediction. *www.mdpi.com/journal/applsci*, 10.3390/app9132660.

Shayan A. Akbar et al. (2019). Source Code Retrieval With Semantics and Order.

Shi Meilong et al. (2020). An Approach to Semantic and Structural Features Learning for Software Defect Prediction. *Hindawi Mathematical Problems in Engineering Volume 2020, Article ID 6038619, 13 pages*, https://doi.org/10.1155/2020/6038619.

Simone Bonechi et al. (2019). Confidence Measures for Deep Learning in Domain Adaptation.

Song Wang et al. (2016). Automatically Learning Semantic Features for Defect Prediction.

Thong Hoang etal. (n.d.). An End-To-End Deep Learning Framework for Just-In-Time Defect Prediction. *2020.*

Tianchi Zhou et al. (2019). Improving defect prediction with deep forest.

Xuan Huo et al. (2017). Learning Semantic Features for Software Defect Prediction by Code Comments Embedding.

Yaning Wu et al. (2018). An Information Flow-based Feature Selection Method for Cross-Project Defect Prediction.

Yao Wan et al. (2019). Multi-Modal Attention Network Learning for Semantic Source Code Retrieval.

Yili et al. (2019). Improving Bug Detection via Context-Based Code Representation Learning and Atention-Based Neural Networks.

Zhao, L., Shang, Z., Qin, A., Zhang, T., Zhao, L., Wei, Y., & Tang, Y. Y. (2019). A cost-sensitive meta-learning classifier: SPFCNN-Miner.

Zhong Zhang and Donghong Li. (2018). Transfer deep convolutional activation-based features for domain adaptation in sensor networks. *Journal on Wireless Communications and Networking (2018) 2018:49* , https://doi.org/10.1186/s13638-018-1059-8.

Zhou Xu et al. (2019). Learning deep feature representation for software defect prediction.

Ziyi Cai et al. (2017). An Abstract Syntax Tree Encoding Method For Cross-Project Defect Prediction.

Appendix C: Full Experimental Results

Appendix C. 2: The PROMISE Source Codes of Target Software Systems.

No	Project	Version	#Files	#Defects	Buggy Rate (%)
1	Ant	1.3	124(–)	20(0)	16.0(+0.1)
		1.4	177(–1)	40(0)	22.6(+0.1)
		1.5	278(–15)	29(–3)	10.4(–0.5)
		1.6	350(–1)	92(0)	26.3(+0.1)
		1.7	741(–4)	166(–1)	22.4(0)
2	Camel	1.0	339(–0)	13(0)	3.8(0)
		1.2	595(–13)	216(0)	36.3(+0.8)

		1.4	847(–25)	145(0)	17.1(+0.5)
		1.6	934(–31)	188(0)	20.1(+0.6)
		1.1	111(0)	63(0)	56.8(0)
		1.4	241(0)	16(0)	6.6(0)
3	Ivy	2.0	352(0)	40(0)	11.4(0)
		3.2	260(–12)	90(0)	34.6(+1.5)
		4.0	281(–25)	67(–8)	23.8(–0.7)
		4.1	266(–46)	67(–12)	25.2(–0.1)
		4.2	355(–12)	48(0)	13.5(+0.4)
4	JEdit	4.3	487(–5)	11(0)	2.3(0)
		1.0	119(–16)	34(0)	28.8(+3.4)
		1.1	104(–5)	37(0)	35.6(+1.6)
5	Log4j	1.2	194(–11)	186(–3)	95.9(+3.7)
		2.0	186(–9)	91(0)	48.9(+2.3)
		2.2	234(–13)	143(–1)	61.1(+2.8)
6	Lucene	2.4	330(–10)	203(0)	61.5(+1.8)
		1.0	26(0)	20(0)	76.9(0)
7	Pbeans	2.0	51(0)	10(0)	19.6(0)
		1.5	235(–2)	141(0)	60.0(+0.5)
		2.0	309(–5)	37(0)	12.0(+0.2)
		2.5	380(–5)	248(0)	65.3(+0.8)
8	Poi	3.0	438(–4)	529(0)	64.2(+0.6)
		1.0	157(0)	16(0)	10.2(0)
		1.1	205(–17)	55(–5)	26.8(–0.2)
9	Synapse	1.2	256(0)	86(0)	33.6(0)
10	Velocity	1.4	195(–1)	147(0)	75.4(+0.4)

		1.5	214(0)	142(0)	66.4(0)
		1.6	229(0)	78(0)	34.1(0)
		2.4	676(–47)	110(0)	16.3(+1.1)
		2.5	754(–49)	379(–8)	50.3(+2.1)
11	Xalan	2.6	875(–10)	411(0)	47.0(+0.5)
		Initial	162(0)	77(0)	47.5(0)
		1.2	436(–4)	70(–1)	16.1(–0.1)
12	Xerces	1.3	446(–7)	68(–1)	15.2(0)
Total		-	14,066(–289)	6542(–77)	31.4(+0.6)

Legend: Numbers in brackets indicate numeral changes compared with the original dataset from PROMISE1 Repository (Cong Pan et al., 2019).

Appendix C. 3: Statistics of Raw and Actionable Datasets.

Project	#Labeled Files (Total)	# Actionable Files	#Non-Defective (TP)	#Defective (TP)	Defective Rate (%)
ant-1.3	126	116	99	17	14.65517
ant-1.4	179	166	130	36	21.68675
ant-1.5	294	270	242	28	10.37037
ant-1.6	352	350	258	92	26.28571
ant-1.7	746	741	575	166	22.40216
camel-1.0	340	339	326	13	3.834808
camel-1.2	609	595	379	216	36.30252
camel-1.4	873	847	702	145	17.11924
camel-1.6	966	934	746	188	20.12848
ivy-1.1	112	111	48	63	56.75676
ivy-1.4	242	241	225	16	6.639004
ivy-2.0	353	352	312	40	11.36364
jedit-3.2	273	248	164	84	33.87097
jedit-4.0	307	281	214	67	23.84342
jedit-4.1	313	266	199	67	25.18797

jedit-4.2	368	355	307	48	13.52113
jedit-4.3	493	487	476	11	2.258727
log4j-1.0	136	115	84	31	26.95652
log4j-1.1	110	100	66	34	34
log4j-1.2	206	188	8	180	95.74468
lucene-2.0	196	186	95	91	48.92473
lucene-2.2	248	234	91	143	61.11111
lucene-2.4	341	330	127	203	61.51515
pbeans-1.0	27	26	6	20	76.92308
pbeans-2.0	52	51	41	10	19.60784
poi-1.5	238	235	94	141	60
poi-2.0	315	309	272	37	11.97411
poi-2.5	386	380	132	248	65.26316
poi-3.0	443	438	157	281	64.15525
synapse-1.0	158	157	141	16	10.19108
synapse-1.1	223	205	150	55	26.82927
synapse-1.2	257	256	170	86	33.59375
velocity-1.4	197	192	47	145	75.52083
velocity-1.5	215	214	72	142	66.35514
velocity-1.6	230	229	151	78	34.06114
xalan-2.4	724	668	561	107	16.01796
xalan-2.5	804	754	375	379	50.26525
xalan-2.6	886	875	464	411	46.97143
xerces-1.2	441	436	366	70	16.05505
xerces-1.3	454	446	379	67	15.02242
xerces-init	163	162	85	77	47.53086
Total	14396	13885	9536	4349	34.41%

Legends: **Actionable** indicates usable files in this study, and **TP** indicates True Positive sample files.

Appendix C. 4: Results for Unmatched Project's Defect Instances.

Project	Unmatched PROMISE Defect Instances	#Defects (FN)	

	#Files	#Not Java File	#Unfound File		Defect Rate (%)
ant-1.3(FP)	9	0	9	3	33.33
ant-1.4(FP)	12	0	12	4	33.33
ant-1.5(FP)	23	0	23	4	17.39
ant-1.6(FP)	1	0	1	0	0
ant-1.7(FP)	4	0	4	0	0
camel-1.2(FP)	13	0	13	0	0
camel-1.4(FP)	25	24	1	0	0
camel-1.6(FP)	31	26	5	0	0
jedit-3.2(FP)	24	1	23	6	25
jedit-4.0(FP)	25	0	25	8	32
jedit-4.1(FP)	46	0	46	12	26.09
jedit-4.2(FP)	12	0	12	0	0
jedit-4.3(FP)	5	0	5	0	0
log4j-1.0(FP)	20	0	20	3	15
log4j-1.1(FP)	9	9	0	3	33.33
log4j-1.2(FP)	17	0	17	9	52.94
lucene-2.0(FP)	9	0	9	0	0
lucene-2.2(FP)	13	0	13	1	7.69
lucene-2.4(FP)	10	0	10	0	0
poi-1.5(FP)	2	0	2	0	0
poi-2.0(FP)	5	0	5	0	0
poi-2.5(FP)	5	0	5	0	0
poi-3.0(FP)	4	0	4	0	0
synapse-1.1(FP)	17	0	17	5	29.41
velocity-1.4(FP)	4	4	0	2	50
xalan-2.4(FP)	55	0	55	3	5.45
xalan-2.5(FP)	49	0	49	8	16.33
xalan-2.6(FP)	10	0	10	0	0

xerces-1.2(FP)	4	0	4	1	25
xerces-1.3(FP)	7	0	7	2	28.57
Ava	470	64	406	74	14.362

Appendix C. 5: Experimental Results for cost sensitive defect data analyses.

Project	#Files	Sequence Length (90%)	Not Cost Sensitive Defect Rate (%)	Cost Sensitive Defect Rate (%)	Δ
ant-1.3	116	485	14.66	16.83	2.17
ant-1.4	166	515.5	21.69	21.95	0.26
ant-1.5	270	465	10.37	11.67	1.3
ant-1.6	350	633	26.29	32.28	5.99
ant-1.7	741	580	22.4	24.56	2.16
camel-1.0	339	208.8	3.83	3.99	0.16
camel-1.2	595	233	36.3	37.18	0.88
camel-1.4	847	250	17.12	17.12	0
camel-1.6	934	266	20.13	20.91	0.78
ivy-1.1	111	389	56.76	58.88	2.12
ivy-1.4	241	503	6.64	9.04	2.4
ivy-2.0	352	591.6	11.36	12.35	0.99
jedit-3.2	248	721.6	33.87	35.59	1.72
jedit-4.0	281	820	23.84	24.36	0.52
jedit-4.1	266	793.5	25.19	29.26	4.07
jedit-4.2	355	934.8	13.52	15.09	1.57
jedit-4.3	487	776.8	2.26	3.33	1.07
log4j-1.0	115	261.8	26.96	29.52	2.56
log4j-1.1	100	256.8	34	35.05	1.05
log4j-1.2	188	285	95.74	95.74	0
lucene-2.0	186	506.5	48.92	48.92	0
lucene-2.2	234	454.2	61.11	61.11	0
lucene-2.4	330	524.9	61.52	61.52	0
pbeans-1.0	26	302.5	76.92	76.92	0

pbeans-2.0	51	312	19.61	23.26	3.65
poi-1.5	235	428.8	60	64.38	4.38
poi-2.0	309	477.4	11.97	12.5	0.53
poi-2.5	380	484.6	65.26	71.68	**6.42**
poi-3.0	438	483.4	64.16	66.27	2.11
synapse-1.0	157	292.4	10.19	11.11	0.92
synapse-1.1	205	458	26.83	30.05	3.22
synapse-1.2	256	404	33.59	36.6	3.01
velocity-1.4	192	337.7	75.52	75.52	0
velocity-1.5	214	359.4	66.36	66.67	0.31
velocity-1.6	229	333	34.06	39.39	5.33
xalan-2.4	668	619.7	16.02	18.45	2.43
xalan-2.5	754	574.4	50.27	50.4	0.13
xalan-2.6	875	657	46.97	48.01	1.04
xerces-1.2	436	568	16.06	16.09	0.03
xerces-1.3	446	561	15.02	15.47	0.45
xerces-init	162	1081.3	47.53	47.53	0
Total	13885	492.45	34.41	36.01	1.6

Legend: Δ refers to improvement of software defect datasets relative to baseline (existing) datasets.

Appendix C. 6: Full Results List for Combined Features-based Defect Prediction

Features	Balanced Defect Prediction Results	Cost-sensitive Defect Prediction Results

THF

Project	Accuracy	Precision	Recall	F1-measu	AUC
ant	81.38	80.01	81.38	80.36	80.26
camel	76.87	74.84	76.87	75.67	66.76
ivy	84.09	86.71	84.09	85.21	79.2
jedit	69.2	96.67	69.2	79.83	80.81
log4j	33.51	91.2	33.51	46.4	41.32
lucene	62.42	63.27	62.42	62.73	66.06
pbeans	47.06	80.41	47.06	50.15	68.54
poi	70.78	72.86	70.78	71.29	75.18
synapse	71.48	70.81	71.48	67.33	74.1
velocity	54.15	66.95	54.15	54.02	58.88
xalan	69.6	69.56	69.6	69.56	74.95
xerces	75.11	74.96	75.11	75.03	61.52
AVA	66.3042	77.3542	66.304	68.13167	68.97

Project	Accurac	Precisio	Recall	F1-measu	AUC
ant	81.51	80.14	81.51	80.47	80.32
camel	76.87	74.74	76.87	75.61	66.82
ivy	83.81	86.61	83.81	85	79.34
jedit	68.99	96.67	68.99	79.69	80.98
log4j	33.51	91.2	33.51	46.4	40.97
lucene	62.12	63.03	62.12	62.45	66.11
pbeans	47.06	80.41	47.06	50.15	68.78
poi	70.78	72.86	70.78	71.29	75.15
synapse	71.88	71.32	71.88	67.91	74.15
velocity	54.59	67.79	54.59	54.39	58.89
xalan	69.71	69.68	69.71	69.69	75.01
xerces	74.66	74.82	74.66	74.74	61.75
AVA	66.291	77.439	66.291	68.14917	69.02

ConcatF

Project	Accuracy	Precision	Recall	F1-measu	AUC
ant	82.05	80.5	82.05	80.55	79.8
camel	78.48	76.35	78.48	77.15	72.63
ivy	87.5	85.56	87.5	86.31	82.36
jedit	83.57	96.45	83.57	89.15	78.82
log4j	32.45	91.04	32.45	45.18	48.82
lucene	57.58	57.58	57.58	57.58	60.87
pbeans	45.1	79.93	45.1	47.87	70.49
poi	62.79	65.65	62.79	63.51	70.91
synapse	68.75	67.7	68.75	61.56	79.44
velocity	56.33	70.63	56.33	56.01	65.09
xalan	67.54	67.67	67.54	67.57	75.25
xerces	73.99	75.21	73.99	74.58	66.63
AVA	66.3442	76.18917	66.344	67.25167	70.93

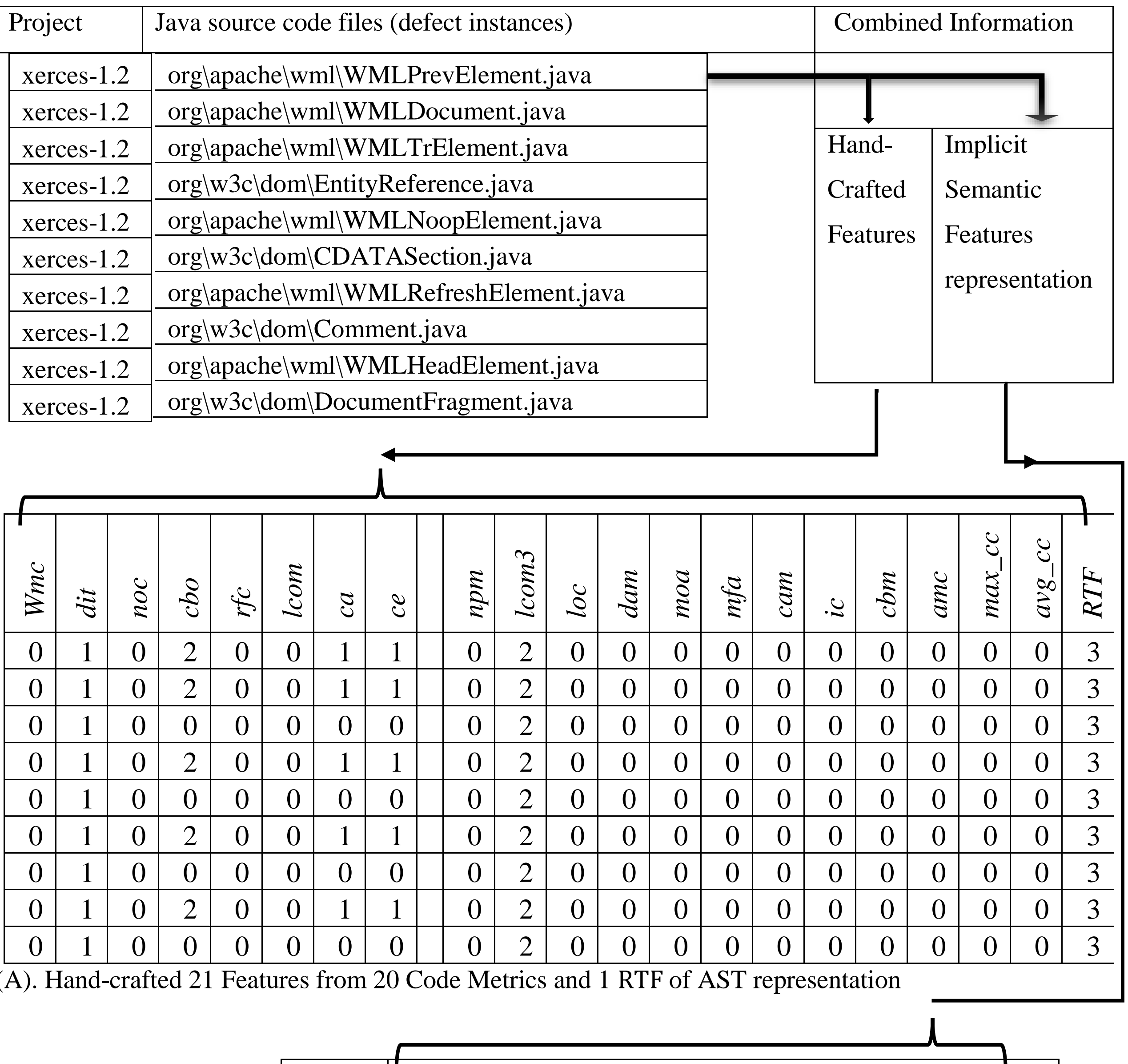
Project	Accuracy	Precision	Recall	F1-measu	AUC
ant	81.24	79.58	81.24	79.81	78.37
camel	78.48	76.02	78.48	76.89	66.85
ivy	84.94	84.78	84.94	84.86	78.73
jedit	74.54	96.78	74.54	83.47	79.53
log4j	31.91	90.96	31.91	44.56	48.4
lucene	63.03	64.27	63.03	63.42	63.37
pbeans	60.78	83.2	60.78	64.56	74.63
poi	68.04	71.7	68.04	68.69	74.52
synapse	70.7	70.27	70.7	65.55	73.99
velocity	53.28	66.42	53.28	53	59.64
xalan	69.49	69.44	69.49	69.43	75.54
xerces	73.32	74.11	73.32	73.71	63.99
AVA	67.4792	77.29417	67.48	68.99583	69.797

HybridF

Project	Accurac	Precision	Recall	F1-meas	AUC
ant	81.11	79.43	81.11	79.7	78.39
camel	78.69	76.28	78.69	77.12	66.81
ivy	84.94	84.78	84.94	84.86	78.53
jedit	74.54	96.78	74.54	83.47	79.68
log4j	32.45	91.04	32.45	45.18	48.54
lucene	62.42	63.67	62.42	62.82	63.27
pbeans	60.78	83.2	60.78	64.56	74.39
poi	68.26	71.84	68.26	68.91	74.56
synapse	70.7	70.27	70.7	65.55	74.11
velocity	53.28	66.42	53.28	53	59.91
xalan	70.29	70.25	70.29	70.21	75.57
xerces	73.32	73.8	73.32	73.56	63.78
AVA	67.565	77.3133	67.57	69.0783	69.8

Project	Accuracy	Precision	Recall	F1-measu	AUC
ant	81.24	79.58	81.24	79.81	78.37
camel	78.48	76.02	78.48	76.89	66.85
ivy	84.94	84.78	84.94	84.86	78.73
jedit	74.54	96.78	74.54	83.47	79.53
log4j	31.91	90.96	31.91	44.56	48.4
lucene	63.03	64.27	63.03	63.42	63.37
pbeans	60.78	83.2	60.78	64.56	74.63
poi	68.04	71.7	68.04	68.69	74.52
synapse	70.7	70.27	70.7	65.55	73.99
velocity	53.28	66.42	53.28	53	59.64
xalan	69.49	69.44	69.49	69.43	75.54
xerces	73.32	74.11	73.32	73.71	63.99
AVA	67.4792	77.29417	67.48	68.99583	69.797

Appendix C. 7: Future work by considering more AST node types to fill the gap shown below:



Project	Java source code files (defect instances)
xerces-1.2	org\apache\wml\WMLPrevElement.java
xerces-1.2	org\apache\wml\WMLDocument.java
xerces-1.2	org\apache\wml\WMLTrElement.java
xerces-1.2	org\w3c\dom\EntityReference.java
xerces-1.2	org\apache\wml\WMLNoopElement.java
xerces-1.2	org\w3c\dom\CDATASection.java
xerces-1.2	org\apache\wml\WMLRefreshElement.java
xerces-1.2	org\w3c\dom\Comment.java
xerces-1.2	org\apache\wml\WMLHeadElement.java
xerces-1.2	org\w3c\dom\DocumentFragment.java

Wmc	*dit*	*noc*	*cbo*	*rfc*	*lcom*	*ca*	*ce*		*npm*	*lcom3*	*loc*	*dam*	*moa*	*mfa*	*cam*	*ic*	*cbm*	*amc*	*max_cc*	*avg_cc*	*RTF*
0	1	0	2	0	0	1	1		0	2	0	0	0	0	0	0	0	0	0	0	3
0	1	0	2	0	0	1	1		0	2	0	0	0	0	0	0	0	0	0	0	3
0	1	0	0	0	0	0	0		0	2	0	0	0	0	0	0	0	0	0	0	3
0	1	0	2	0	0	1	1		0	2	0	0	0	0	0	0	0	0	0	0	3
0	1	0	0	0	0	0	0		0	2	0	0	0	0	0	0	0	0	0	0	3
0	1	0	2	0	0	1	1		0	2	0	0	0	0	0	0	0	0	0	0	3
0	1	0	0	0	0	0	0		0	2	0	0	0	0	0	0	0	0	0	0	3
0	1	0	2	0	0	1	1		0	2	0	0	0	0	0	0	0	0	0	0	3
0	1	0	0	0	0	0	0		0	2	0	0	0	0	0	0	0	0	0	0	3

(A). Hand-crafted 21 Features from 20 Code Metrics and 1 RTF of AST representation

(B). AST representation (Patterns) for semantic features extraction

File-1	*PackageDeclaration ,InterfaceDeclaration, ReferenceType*
File-3	*PackageDeclaration, InterfaceDeclaration, ReferenceType*
File-4	*PackageDeclaration, InterfaceDeclaration, ReferenceType*
File-5	*PackageDeclaration, InterfaceDeclaration, ReferenceType*
File-6	*PackageDeclaration, InterfaceDeclaration, ReferenceType*
File-7	*PackageDeclaration, InterfaceDeclaration, ReferenceType*
File-8	*PackageDeclaration, InterfaceDeclaration, ReferenceType*
File-9	*PackageDeclaration, InterfaceDeclaration, ReferenceType*
File-10	*PackageDeclaration ,InterfaceDeclaration, ReferenceType*

Legend: Red and Green indicates the buggy and non-buggy file, respectively.