Chapter 6

An Unsupervised SBP Technique in Functional Paradigm

This chapter focuses on the fourth contribution of this thesis detail in Section 1.7.4. We provide an introduction, motivation, and key feature for the problem to develop classification-based unsupervised SBP for FP in Section 6.1. The description of newly created Haskell datasets is given in Section 6.2. Section 6.3 demonstrates the proposed work to develop the classification-based unsupervised SBP for FP. The experimental setup is given in Section 6.4. Section 6.5 covers the results obtained using the proposed technique. Section 6.6 reports the threat to validity associated with the chapter. Section 6.7 focuses on the conclusion and future scope of this chapter. Detailed related work is provided in Section 2.3.

6.1 Introduction

A functional programming paradigm is one where instructions are constructed by applying and composing functions. It is based on a declarative programming style and is known to evaluate mathematical functions using Lambda Calculus. Lambda Calculus is a Turing-complete language, i.e. anything that can be computed by a

Turing machine can also be calculated using Lambda Calculus [230]. Because of its modular nature, the FP has attracted programmers and researchers for the last several years [231]. Many researchers have worked on software design, security, cost estimation, testing, and defects in functional programming [232]. Other research works going on in the area of the FP are design methodologies [232], refactoring [233, 234], and reusability of components [235].

Detecting defects in software after it's been delivered to the customer is a costly and complex process [127, 128]. Early defect detection can significantly reduce testing, diagnostics, and resolution efforts, potentially lowering development and maintenance costs [130]. Research indicates that a considerable portion of software development costs, 30% to 50%, is contributed to testing [131]. However, there's a lack of prior research using software metrics (SMs) to develop a Software Defect Prediction (SDP) system in the functional programming (FP) domain, particularly in Haskell. In this study, datasets were created from Haskell packages, leveraging six SMs. High metric values indicate functions that may need refactoring, review, and testing [132]. SMs serve as indicators, much like compiler warnings, suggesting areas for code improvement. Many SDP models for unlabeled datasets employ clustering methods that require manual labeling by experienced software engineers, which can be costly and challenging to implement [86]. This study aims to develop a fully automated SDP system that eliminates the need for software experts and manual labeling, focusing on Haskell packages.

Based on the recent review article authored by Li et al. [34], we present Table 6.1 to demonstrate novel contributions of the proposed approach as compared to directly related threshold-based SDP methods described in this article [34]. In Table 6.1, the ✓ indicates the steps included in the SDP approaches, while × represents the steps not included. In Table 6.1, we created new defect datasets from Haskell, marking the first exploration of the functional paradigm for SDP. Subsequently, we observed that no existing threshold-based SDP method addressed the skewness

TABLE 6.1: Fundamental difference between the proposed approach UMV and other existing threshold-based SDP methods

Threshold Based SDP Approaches	Datasets Creation (Haskell)	Skewness Reduction (log, sqrt, cbrt)	Threshold Derivation (log, sqrt, cbrt)	Clustering	Labelling	Metrics Selection	Instances Selection	Balancing Dataset	Models Building	Models Prediction	Performance of (log, sqrt, cbrt) Based Models	Majority Voting Ensemble	Silhouette Width Coefficient	Defect Wise Metric Distribution
CT [158]	×	×	×	✓	✓	×	×	×	×	×	×	×	×	×
CLA/CLAMI [116]	×	×	×	✓	✓	✓	✓	×	✓	✓	×	×	×	✓
ACL [140]	×	×	×	✓	✓	×	×	×	×	×	×	×	×	×
CLAMI+ [117]	×	×	×	✓	✓	✓	✓	×	✓	✓	×	×	×	×
UMV	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

reduction of software metrics (SMs). The existing threshold-based SDP models, including Clustering Threshold (CT), CLA/CLAMI, ACL, and CLAMI+, employed expert-based thresholds, median, average, and median values of SMs as thresholds, respectively. Therefore, we introduced a novel approach, employing three skewness reduction techniques [logarithmic (log), square root (sqrt), cubic root (cbrt) transformation] and deriving three thresholds using the mean and standard deviation of each metric. Based on these thresholds, we developed three individual SDP models and assessed their significance based on the obtained results. Furthermore, within this novel approach, we applied an unsupervised majority voting (UMV) ensemble method to enhance the performance of the three individual models and prevent overfitting of results [70]. In contrast to this, none of the existing techniques (CT, CLA/CLAMI, ACL, CLAMI+) have exploited the power of ensemble to enhance the performance of their models and remove overfitting. Also unlike the existing techniques, the results of our novel proposed SDP model are illustrated with silhouette width coefficients to measure the goodness of clusters. We have also included the defect-wise boxplot of software metric distribution to identify the suitable SMs and understand the importance of different SMs to develop SDP using UMV.

The outcomes of our proposed UMV technique are rigorously compared with five established supervised machine learning models, four unsupervised machine learning models, and six threshold-based unsupervised SDP models.

The step-by-step process to implement the proposed technique, unsupervised majority voting (UMV) on Haskell packages for defect proneness detection in each

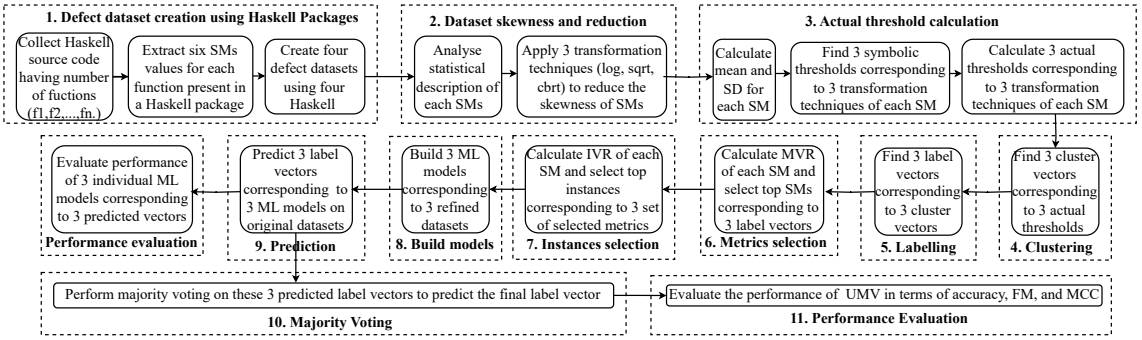

FIGURE 6.1: Block diagram of the proposed approach showing the steps involved in UMV

function is shown in Fig. 6.1. Our research work is conducted on four new datasets extracted from Haskell packages. We proposed an unsupervised majority voting UMV ensemble model to identify the defect proneness in each function of Haskell packages. We have compared the results of the UMV approach with the other existing methods (threshold-based model, unsupervised learning algorithms, supervised learning algorithms).

6.2 Datasets Description

Software metrics play a crucial role in describing the internal characteristics of source code. So, SMs are extracted directly from the source code and are used to measure the reliability, quality, and design of the source code. The metrics used in the construction of our SDP model are inspired by the SMs used in OO and procedural paradigms. These are general metrics affecting defect proneness and are not specific to any particular programming paradigm. These metrics, from both paradigms, translate directly to Haskell without any modification. Based on a study by Ryder and Thompson [132,171], it is found that Indegree (IND), Outdegree (OUTD), Cyclomatic Complexity (CC), Type Signature Argument (TS), Branching Depth (BD), and Lines of Code (LOC) are suitable metrics for the development of SDP systems. The definition and scope of metrics used in this research work are explained below:

1. Branching Depth (BD): This SM signifies the maximum nesting of if/case conditional statements. High nesting of conditional statements makes the source code complex and difficult to understand, thereby increasing the chances of defects [171].

2. Lines of Code (LOC): It is the simplest SM to measure and indicates the size of the code. If the value of LOC is high, then it is difficult to understand the functions and hence increases the probability of defects [171].

3. Indegree (IND): The indegree of a function is defined as the number of functions that call this function. If the IND value of a function is high, it indicates that the function is heavily used in the program, and making changes in the function can cause a rippling effect [132].

4. Outdegree (OUTD): It is defined as the number of functions a given function calls. OUTD indicates the dependency of a function on other functions. If a function calls a defect-prone function, then its output is also likely to be incorrect. IND and OUTD metrics closely resemble the Fan-in and Fan-out metrics of the OO paradigm.

5. Type signature arguments (TS/TSA): This SM indicates the number of unique data types that a given function accepts as an argument. A high value of this SM indicates that the function is complicated and hence is more likely to have data type error/defects [132].

Four different Haskell packages are used for extracting SMs and exploring the scope of these metrics. These Haskell packages are collected from the Haskell library, and their six SM values are extracted using various tools. The collected datasets are manually labeled by the authors. We have experience of more than 10 years in computer programming. The main problem is categorizing Haskell's modules (functions) in two classes as defective or non-defective functions. We labeled the datasets by making and following some set of rules. We investigated the source code

style, size, and complexity to label the module. While finalizing the labeling, we considered extracted SMs values and some information given in Haskell packages.

6.2.1 Dataset Collection

In this experiment, the created Haskell datasets and package-wise statistical description of metrics are shown in Table 6.2. These 4 Haskell datasets are created by us from the Hackage source code[1]. The procedure we followed to extract SMs from Haskell packages is shown in Fig. 6.2.

There is no dataset available in Haskell. So, we created a dataset from four Haskell packages Ethereum (v 0.0.4), Hburg (v 1.1.3), Mios (v 1.6.0), and Piet (v 0.1), which are collected from Hackage (Haskell package repository). We extracted SM values of each function present in these packages. These datasets are collected and then converted into CSV (Comma Separated Values) format. Fig. 6.2 shows the step-by-step method used to generate a CSV file from each Haskell Package. We have selected only limited Haskell packages due to the unavailability of the proper software metric extraction tools. We have applied Homplexity[1], Argon[1], and SourceGraph[1] on many randomly selected Haskell packages for dataset creation, but we created only these four Haskell packages successfully. The main hindrance was that these software metric extraction tools were not working properly on all the Haskell packages.

[1]https://hackage.haskell.org/

TABLE 6.2: Statistical description of the SMs extracted from Haskell Packages

	Etherium (HP1)						Hburg (HP2)						Mios (HP3)						Piet (HP4)					
Parameter	IND	OUTD	CC	TSA	BRD	LOC	IND	OUTD	CC	TSA	BRD	LOC	IND	OUTD	CC	TSA	BRD	LOC	IND	OUTD	CC	TSA	BRD	LOC
#Function	119	119	119	119	119	119	162	162	162	162	162	162	116	116	116	116	116	116	78	78	78	78	78	78
Minimum	0.0	0.0	1.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	1.0	1.0	0.0	1.0
Maximum	16	26	22	4	2	256	9	37	9	8	6	116	7	24	14	4	3	89	17	17	18	6	4	33
Mean	1.5	4.1	2.2	1.9	0.5	18.8	1.5	3.4	1.8	2.4	0.5	6.9	1.4	3.7	1.9	2.2	0.5	10.3	1.7	2.7	1.8	2.1	0.3	5.6
Median	1.0	2.0	1.0	2.0	0.0	4.0	1.0	1.0	1.0	2.0	0.0	2.0	1.0	2.0	1.0	2.0	0.0	4.0	1.0	2.0	1.0	2.0	0.0	3.0
Sum	178	483	265	229	58	2240	242	555	284	385	79	1112	162	430	226	252	53	1189	135	210	138	161	25	433
Variance	3.1	28.0	7.7	0.6	0.3	1921.5	2.4	25.4	2.0	1.7	0.7	209.6	1.3	22.0	4.6	1.0	0.6	286.3	5.0	10.6	5.1	1.3	0.5	52.1
Stdev	1.8	5.3	2.8	0.8	0.6	43.8	1.5	5.0	1.4	1.3	0.8	14.5	1.1	4.7	2.2	1.0	0.8	16.9	2.2	3.3	2.3	1.1	0.7	7.2
Skewness	5.0	2.3	4.3	0.9	0.7	3.5	2.0	3.2	2.8	0.9	3.2	4.8	1.3	1.8	3.1	0.0	1.7	2.8	4.7	2.6	5.2	1.1	2.7	2.3
Kurtosis	35.9	5.9	22.9	1.0	-0.5	12.7	5.0	13.4	9.0	2.8	15.7	28.4	3.7	3.1	10.6	-0.8	2.3	8.2	27.0	7.0	31.8	0.9	9.1	4.8
Defect %	11.76						17.9						18.96						21.79					

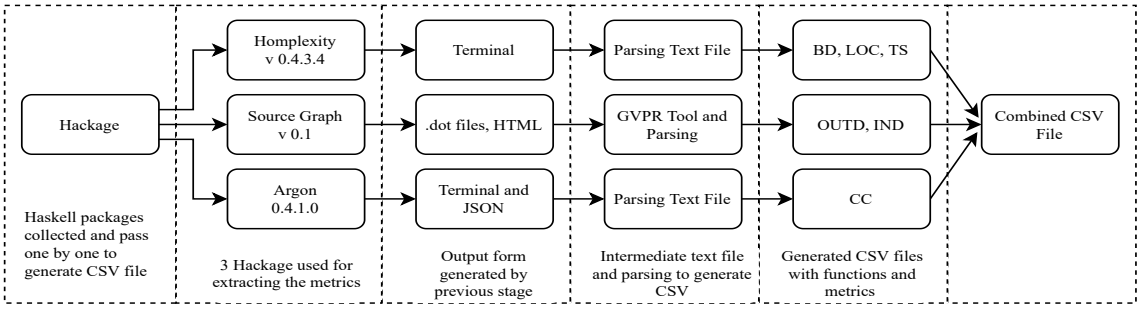

FIGURE 6.2: Block diagram for step-by-step procedure to extract software metrics from the Haskell Packages

6.2.2 Software Metric Values Extraction:

Following steps have been followed for SMs extraction from Haskell packages (also shown in Fig. 6.2).

1. Three Haskell packages viz. Homplexity (v 0.4.3.4), SourceGraph (v 0.1), and Argon (v 0.4.1.0) are used as tools to extract metrics values. The output format generated by the above Haskell packages is shown in Fig. 6.2, along with the specific metrics extracted.

2. The metrics extracted by three different extraction tools are in a different format. So, we parsed two of these formats to generate CSV files. One contains BD, LOC, and TS metrics, and the other one contains CC metrics for each function.

3. The output generated by Source Graph is in .dot file format. To parse it, we have used the GVPR tool [236], a graph pattern reading and processing tool. It generates a CSV file that contains the values of OUTD and IND metrics for each function.

4. Now, these generated CSV files are combined to create a single CSV file with six metrics values for each function.

5. Only the common metrics from the three CSV files are selected to generate the combined CSV file. Finally, this composite CSV file contains functions as the

rows and the corresponding six SM values as the column. These six metrics are indegree (IND), outdegree (OUTD), cyclomatic complexity (CC), type signature arguments (TS), branching depth (BD), and lines of code (LOC).

6. These above steps are repeated for all the Haskell packages.

6.3 Proposed Approach: UMV

The proposed novel approach is an unsupervised majority voting (UMV) ensemble technique that combines the SDP results of the three SM threshold-based SDP models using majority voting. The proposed method is an extension of the already published work [47], where only natural logarithmic transformation was considered on the publicly available datasets. While, this work consists of threshold derivation using three transformation techniques Natural Log transformation (log), Square root transformation (sqrt), and Cubic root transformation (cbrt), and then we have built the SDP model same as discussed in the paper [47]. After building the SDP model, the majority voting ensemble technique is applied on predicted results by these three different threshold-based SDP models. In this proposed approach, UMV has experimented on novel FP (Haskell) datasets, which is the first research work to propose an SDP model for the FP.

The SM threshold value is a crucial tool for both software developers and software testers to audit and analyze the software [171, 237]. It is found that SMs play a crucial role in predicting defect proneness in a given function in Haskell packages. The step-by-step process to develop the unsupervised majority voting (UMV) ensemble technique is described as follows:

Step 1 Dataset creation: Created the Haskell datasets $HP[1:n, 1:m]$ as discussed in Subsection 6.2.1. Where n is total number of functions and m is the total number of metrics in a Haskell package (HP).

Step 2 Distribution and transformation: After extracting relevant metrics, we calculated the values of basic statistical parameters of the metrics, as shown in Table 6.2. Based on skewness index γ_m, calculated using equation (6.1), we concluded that the distribution of the metrics is positively skewed. In equation (6.1), n is the number of instances/functions, and μ is the mean of SM.

$$\gamma_m = \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^3}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2\right)^{3/2}} \tag{6.1}$$

We have concluded from the earlier research [238], that all SMs are positively skewed to the right. Many data transformation methods are available to reduce the skewness of data and make it close to fit the normal distribution [1]. In this research, we have applied three transformation methods, viz. log(), sqrt(), and cbrt() transformation. One is added to the data because the natural log is not possible for zero values (viz. as $x = x + 1$, where x is SM value). Let there be a Haskell package dataset as $HP[1:n, 1:m]$. The transferred dataset $TD[1:n, 1:m]$ is calculated using equations (6.2,6.3,6.4).

$$TD_{log} = log(HP[1:n, 1:m]) \tag{6.2}$$

$$TD_{sqrt} = sqrt(HP[1:n, 1:m]) \tag{6.3}$$

$$TD_{cbrt} = cbrt(HP[1:n, 1:m]) \tag{6.4}$$

Step 3 SM Threshold Calculation: After transferring the data using the three transformation methods, the distribution parameters i.e., mean (μ_m) (using equation (6.5), and standard deviation (SD) (Ω_m) (using equation (6.6) are calculated for each SM m.

$$\mu_m = \frac{\sum_{i=1}^{n} x_i}{n} \quad \forall \quad TD_{log}, TD_{sqrt}, TD_{cbrt} \tag{6.5}$$

$$\Omega_m = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu_m)^2}{n-1}} \quad \forall \quad TD_{log}, TD_{sqrt}, TD_{cbrt} \tag{6.6}$$

Next, we find the value of SM threshold (θ'_m). Equations (6.7,6.8,6.9) are used to calculate the threshold values $\theta'_{TD_{log}}$, $\theta'_{TD_{sqrt}}, \theta'_{TD_{cbrt}}$: Where $\forall \quad j = 1, 2, .., m$

$$\theta'_{TD_{log}}[j] = mean(TD_{log}[j]) + SD(TD_{log}[j]) \tag{6.7}$$

$$\theta'_{TD_{sqrt}}[j] = mean(TD_{sqrt}[j]) + SD(TD_{sqrt}[j]) \tag{6.8}$$

$$\theta'_{TD_{cbrt}}[j] = mean(TD_{cbrt}[j]) + SD(TD_{cbrt}[j]) \tag{6.9}$$

$\theta'_{TD_{log}}, \theta'_{TD_{sqrt}}, \theta'_{TD_{cbrt}}$ are symbolic thresholds for the transformed data TD. To calculate the representative/actual threshold for the original data $HP[1:n, 1:m]$, $\theta_{TD_{log}}, \theta_{TD_{sqrt}}$, $\theta_{TD_{cbrt}}$ can be calculated from equations (6.10,6.11,6.12).

$$\theta_{TD_{log}}[j] = exp(\theta'_{TD_{log}}[j]) \quad \forall \quad j = 1, 2, .., m \tag{6.10}$$

$$\theta_{TD_{sqrt}}[j] = (\theta'_{TD_{sqrt}}[j])^2 \quad \forall \quad j = 1, 2, .., m \tag{6.11}$$

$$\theta_{TD_{cbrt}}[j] = (\theta'_{TD_{cbrt}}[j])^3 \quad \forall \quad j = 1, 2, .., m \tag{6.12}$$

The above equations are the inverse functions of log, sqrt, and cbrt functions. After calculation of the actual threshold vectors ($\theta_{TD_{log}}[1:m], \theta_{TD_{sqrt}}[1:m], \theta_{TD_{cbrt}}[1:m]$) using the three equations, values of cluster vectors $C_{log}[1:n], C_{sqrt}[1:n], C_{cbrt}[1:n]$ are calculated. Each of these cluster vectors is defined as the total number of metrics greater than their corresponding threshold in each instance.

Step 4 Clustering: To find the cluster vectors $C_{log}[1:n], C_{sqrt}[1:n], C_{cbrt}[1:n]$, firstly, we find those metrics whose values are greater than their corresponding threshold value for each instance. Then, we count the total number of metrics whose values surpass their corresponding threshold value in an instance. The following equations (6.13,6.14,6.15) are used to find the cluster vectors. Where

$\forall \quad i = 1, .., n, j = 1, .., m.$

$$C_{log}[i] = count(HP[i, j] > \theta_{TD_{log}}[j]) \tag{6.13}$$

$$C_{sqrt}[i] = count(HP[i, j] > \theta_{TD_{sqrt}}[j]) \tag{6.14}$$

$$C_{cbrt}[i] = count(HP[i, j] > \theta_{TD_{cbrt}}[j]) \tag{6.15}$$

Step 5 Labelling instances: We have chosen the average silhouette width vs. number of clusters method to find the optimal number of clusters [239]. In Fig. 6.3, there are four subfigures for each Haskell dataset, in which the X-axis represents the number of clusters k, and the Y-axis represents the average silhouette width coefficient. The silhouette coefficient is used to measure the goodness of the cluster (see subsection 6.4.2.1). It can be seen that when the number of clusters is 2, then the average silhouette width value is maximum. Hence, from Fig. 6.3, it is concluded that the optimal number of clusters is two. So, to label the instances in two clusters, mark the group of instances for $C > \lceil (max(C)/2) \rceil$ as defect-prone and the remaining as non-defective. The final labeling of instances is done based on the fact that higher SM values cause the instances to be defect prone [113, 212, 213]. On completion of this Step 5, we get a temporarily labeled Haskell package $HP[1 : n, 1 : m]$ dataset where labels are achieved based on SM threshold vector $(\theta_{TD_{log}}[1 : m], \theta_{TD_{sqrt}}[1 : m], \theta_{TD_{cbrt}}[1 : m])$. Now, we have three label vectors $(L_{log}[1 : n], L_{sqrt}[1 : n], L_{cbrt}[1 : n])$ using three threshold methods.

Step 6 SM selection: After labeling HP, we have found that the provided labels don't completely conform to the defect proneness tendency. Defect proneness tendency is defined as the condition where higher complexity in the source code causes more defect proneness [24, 28]. So, we have calculated the SM violation ratio $MVR[1 : m]$ for each SM using equation (6.16). Thereafter, we arranged the metrics in ascending order of MVR values and selected the minimum number of required metrics $(\lceil log_2(m) \rceil)$ for SDP [220]. In this way, we get a reduced datasets

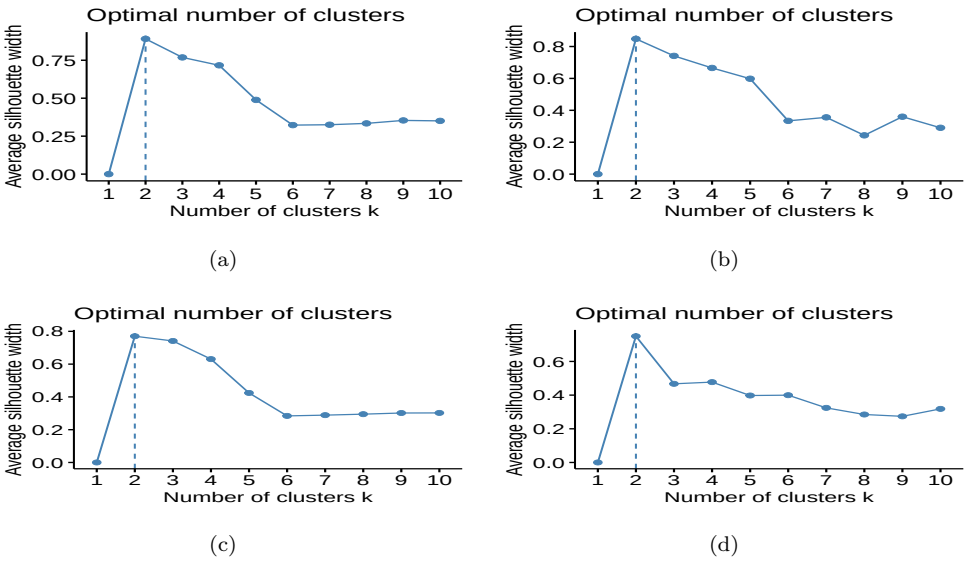

FIGURE 6.3: Plots showing the Average silhouette width vs. number of clusters k in the four datasets (a) Etherium (b) Hburg (c) Mios (d) Piet

$RD_{log}[1:n, 1:p], RD_{sqrt}[1:n, 1:p], RD_{cbrt}[1:n, 1:p]$ after selecting p metrics out of m metrics.

$$MVR[i] = MVS[i]/n \quad \forall \quad i = 1, 2, 3, ..., m \tag{6.16}$$

Where, $MVS[1:m]$ is metric violation score. MVS is defined as total number of SM values (number of instances) exceeding the threshold in a particular SM [47].

Step 7 Instance selection: After selecting the metrics, we have found that the reduced dataset $HP[1:n, 1:p]$ is still not following the defect proneness tendency. So, we have calculated the instance violation ratio $IVR[1:n]$ using equation (6.17) for each instance and selected a minimum 70% of instances based on minimum IVR with the condition that at least two defective instances are selected. After this, we achieve a filtered and cleaned datasets $RD_{log}[1:q, 1:p], RD_{sqrt}[1:q, 1:p], RD_{cbrt}[1:q, 1:p]$ and label vectors $(L_{log}[1:q], L_{sqrt}[1:q], L_{cbrt}[1:q])$ that closely follow the defect proneness tendency. Where q is the selected number of functions.

$$IVR[j] = IVS[j]/p \quad \forall \quad j = 1, 2, 3, ..., n \tag{6.17}$$

Where, $IVS[1:n]$ is the instance violation score. IVS is defined as the total number of SM values violated in a particular instance [47].

If we fail to find a minimum of two defective instances in the filtered dataset, then we select more number of instances up to 99%. If still, we fail to find a minimum of two defective instances in the filtered dataset, then we select more metrics based on minimum MVR and repeat Step 6 and Step 7.

Step 8 Model building: After Step 7, we get cleaned and reduced datasets $(RD_{log}[1:q, 1:p], RD_{sqrt}[1:q, 1:p], RD_{cbrt}[1:q, 1:p])$, and label vectors $(L_{log}[1:q], L_{sqrt}\ [1:q], L_{cbrt}[1:q])$, which are extremely imbalanced. So, we balance the dataset using random oversampling techniques (ROSE) and build 10-fold cross validation random forest (RF) models using $RD_{log}[1:q, 1:p], RD_{sqrt}[1:q, 1:p], RD_{cbrt}[1:q, 1:p]$ and $L_{log}[1:q], L_{sqrt}[1:q], L_{cbrt}[1:q]$. So now, we have three trained models using equations (6.18,6.19,6.20).

$$model_{log} = train_{RF}(RD_{log}[1:q, 1:p], L_{log}[1:q]) \tag{6.18}$$

$$model_{sqrt} = train_{RF}(RD_{sqrt}[1:q, 1:p], L_{sqrt}[1:q]) \tag{6.19}$$

$$model_{cbrt} = train_{RF}(RD_{cbrt}[1:q, 1:p], L_{cbrt}[1:q]) \tag{6.20}$$

Step 9 Model prediction: After building these three models, we feed original dataset $HP[1:n, 1:m]$ as input to the built models to predict the defect in Haskell packages using equations (6.21,6.22,6.23).

$$prediction_{log} = model_{log}(HP[1:n, 1:m]) \tag{6.21}$$

$$prediction_{sqrt} = model_{sqrt}(HP[1:n, 1:m]) \tag{6.22}$$

$$prediction_{cbrt} = model_{cbrt}(HP[1:n, 1:m]) \tag{6.23}$$

Step 10 Majority voting ensemble: After predicting the labels $prediction_{log}$, $prediction_{sqrt}$, $prediction_{cbrt}$ using the three models, we have applied the unsupervised majority voting ensemble technique (UMV) on these labels and calculated the final label using the equation (6.24). Where $\forall \quad i = 1, 2, ..., n$

$$prediction_{UMV}[i] = mode(prediction_{log}[i], prediction_{sqrt}[i], prediction_{cbrt}[i]) \quad (6.24)$$

Step 11 Performance evaluation: Based on the predicted class using the majority voting ensemble technique, we have evaluated its performance in terms of accuracy, FM, and MCC.

The proposed algorithm UMV is written as Algorithm 10. The run time complexity of Algorithm 10 is $O(n * m)$, where n is instances and m is metrics.

6.4 Experimental Design

The introduced research questions and performance measures are reported in Section 6.4.1 and Section 6.4.2, respectively.

6.4.1 Research Questions

RQ1: Is the performance of proposed UMV model on Haskell code better than standard supervised machine learning algorithms?
To answer this question, we have implemented five supervised machine learning algorithms viz. Random Forest (RF), C5.0 (C50), Naive Bayes (NB), Support Vector Machine (SVM), and K-Nearest Neighbor (KNN), using a 10-fold cross-validation technique to compare the performance with UMV.

RQ2: Is the performance of proposed UMV method on Haskell code better than standard unsupervised learning algorithms and threshold-based models?
To answer this question, we have implemented four unsupervised machine learning algorithms viz. K-means (KM), Hierarchical Clustering (HC), Neural Gas Clustering

Algorithm 10: Algorithm to implement unsupervised majority voting (UMV)

Input: Haskell packages source code HP1, HP2, HP3, HP4
Output: Performance of UMV as Accuracy, FM and MCC
Algorithmic Steps:
Step1: Create the four Haskell datasets $HP[1:n, 1:m]$ (Subsection (6.2.1))
for $d \leftarrow$ *1 to 4* **do**
  Step2: Calculate skewness index γ_m of each SM using (6.1)
  Step3: Apply the three transformation method viz. log(), sqrt(), and cbrt() on the datasets using (6.2, 6.3, 6.4)
  Step4: Calculate the mean and standard deviation using (6.5, 6.6)
  Step5: Calculate symbolic threshold for transformed dataset using (6.7, 6.8, 6.9)
  Step6: Calculate actual threshold for original dataset using (6.10, 6.11, 6.12)
  Step7: Find the cluster vector with the help of each threshold using (6.13, 6.14, 6.15))
  for $i \leftarrow 1$ **to** n **do**
    $t \leftarrow 0$
    for $j \leftarrow 1$ **to** m **do**
      if $HP[i,j] > \theta_{TD}[j]$ **then**
        $t = t + 1$
    end
    $C[i] \leftarrow t$
  end
  Step8: Label the functions with the help of each cluster vector as follows
  for $i \leftarrow 1$ **to** n **do**
    if $C[i] > \lceil (max(C)/2) \rceil$ **then**
      $label\_defect[i] \leftarrow$"YES"
    else
      $label\_defect[i] \leftarrow$ 'NO'
  end
  Step9: Select the metrics based on MVR using (6.16)
  Step10: Select the instances based on IVR using (6.17)
  Step11: Now balance the dataset using ROSE technique
  Step12: Perform training using random forest algorithm using (6.18, 6.19, 6.20)
  Step13: Perform prediction in original dataset using trained model as equations (6.21, 6.22, 6.23)
  Step14: Perform majority voting ensemble on the results predicted by three models using (6.24)
  Step15: Calculate the performance parameters of the UMV
end

(NGC), and Model-Based Clustering (MBC), and four threshold-based algorithms viz. CLAMI, CLAMI+, ACL, TCLP to compare the performance with UMV.

RQ3: Are the extracted SMs capable of predicting defect proneness in the functions of the Haskell?
To answer this question, we have investigated the performance of supervised, unsupervised, and threshold-based models based on these metrics. We have shown silhouette plots and defect-wise plots to determine which metrics play a crucial role in predicting the defects in Haskell code snippets.

6.4.2 Performance Measure

Three performance parameters, viz. accuracy, f-measure (FM), and MCC, have been evaluated on four Haskell packages. To visualize the result of UMV on four Haskell packages, boxplots, average silhouette width plot, defect-wise boxplot, and critical diagram of the models are presented. The description of the performance measures and boxplot are provided in Section 2.4.3.1. The average silhouette width coefficient is explained in the following subsection. The defect-wise boxplot and critical diagram are described with the results.

6.4.2.1 Average Silhouette Width Coefficient

The silhouette coefficient is used to measure the goodness of a cluster. It depends on two principles, viz. cohesion and separation. Cohesion is used to measure the degree of similarity of the data points within a cluster. Separation is used to measure the degree of similarity of data points with those that are there in another cluster. For the goodness of the cluster, values of cohesion should be high, and separation should be low. Let us consider a dataset D having N number of data points. Suppose each data point $N_i \in R^m$ is to be assigned a cluster using any clustering algorithm like k-means [240], or k-median [241]. On the dataset D, let C be the obtained number of clusters. For a specific cluster $U \in C$, suppose U has n number of data points. So for each data point U_i, Silhouette coefficient $S(i)$ is calculated using

parameters $a(i)$ and $b(i)$.

$$\mathrm{a(i)} = \frac{\sum_{\mathrm{j}\in U} \parallel U_i - U_j \parallel}{n_u} \tag{6.25}$$

Euclidean distance $d(i, v)$ is used to measure the dissimilarity between two data points i and v. So, let us consider a cluster $V \in C$ having size n_v and we compute $d(i, v)$ using equation (6.26)

$$\mathrm{d(i, v)} = \frac{\sum_{\mathrm{j}\in v} \parallel U_i - V_j \parallel}{n_u} \tag{6.26}$$

$$\mathrm{b(i)} = \mathrm{Min}_{\forall \mathrm{v}\in \mathrm{C}, \mathrm{v}\neq \mathrm{u}} d(i, v) \tag{6.27}$$

So, the silhouette coefficient can be calculated using the following equation (6.28)

$$S(i) = \frac{b(i) - a(i)}{\max[a(i), b(i)]} \tag{6.28}$$

A better quality cluster formation is indicated by a value of $S(i)$ that is close to 1. However, a low or negative value shows that the data point belongs to another cluster. So, to determine the overall goodness of a cluster, the average silhouette coefficient [242] of the entire dataset D is calculated using the following equation (6.29).

$$AvgSil = \frac{\sum_{i\in D} S(i)}{N} \tag{6.29}$$

6.5 Experimental Results

In this section, we present the experimental results of the supervised, unsupervised, and threshold-based model on FP datasets. We have implemented seven threshold-based techniques (CLAMI, CLAMI+, ACL, TCLP, SQRT, CBRT, UMV), four unsupervised models (KMS, NGC, MBC, HC)), and five supervised models (NB, SVM, KNN, RF, C50) on each Haskell dataset. Table 6.3 shows the performance of

different models in terms of accuracy, FM, and MCC values. The performance of each SDP technique is presented as a boxplot in Fig. 6.4.

A statistical test, Nemenyi test is conducted to compare the models statistically using a critical diagram (CD) that is shown in Fig. 6.5. The critical diagram helps to notice the performance of all the SDP methods in one view. It is a better way to compare the performance of the different models on different datasets statistically. The CD contains the Friedman test p-values (0.00) with the hypothesis (Ha) as significantly different, critical distance (11.534), and a mean score of each model. The diagram shows the mean score of the models in ascending order [47]. The average silhouette plot for cluster and defect-wise boxplot for SMs generated by proposed techniques are shown in Fig. 6.6 and Fig. 6.7, respectively.

Silhouette plot, shown in Fig. 6.6, is used to visualize the goodness of clusters obtained by our proposed approach in Haskell packages. The goodness of a cluster is measured using the average silhouette width coefficient that has a range of [-1,+1]. Silhouette coefficient value near +1, indicates that the sample is far away from the neighboring cluster. While a silhouette coefficient value near 0, indicates that the sample is on the boundary or very close to the decision boundary between two neighboring clusters. Silhouette coefficient value near -1, indicates that the sample is likely to be in the wrong cluster. E.g., in Fig. 6.6(c), the X-axis represents the silhouette width S_j, and the Y-axis shows the instances (functions). There are 116 horizontal lines pertaining to each instance whose length shows the silhouette width value of each sample. These samples are classified into two clusters C_j where j=1,2. For first cluster j=1, n_j=104 (number of samples in cluster 1 (C_1)), and 0.91 shows the average silhouette width value ($ave_{i \epsilon C_1} S_i$) of cluster 1. Similarly, for second cluster j=2, n_j=12 (number of samples in cluster 2 (C_2)), and 0.16 is the average silhouette width value ($ave_{i \epsilon C_2} S_i$) of cluster 2. In cluster 2, we can see that the silhouette width value ($ave_{i \epsilon C_2} S_i$) of some samples is negative, which shows the wrong assignment of these samples to cluster 2. But, overall, an average silhouette

TABLE 6.3: Performance of different models in terms of accuracy, f-measure, and MCC

(a) Accuracy %

Haskell	Threshold Based Methods							Unsupervised Methods				Supervised Methods				
	CLAMI	CLAMI+	ACL	TCLP	SQRT	CBRT	UMV	KMS	NGC	MBC	HC	NB	SVM	KNN	RF	C50
HP1	14.29	41.18	77.31	93.28	90.76	83.19	93.79	94.96	95.80	68.91	94.12	94.29	94.41	94.47	97.50	95.38
HP2	92.59	49.38	87.04	90.12	95.06	94.44	95.96	85.80	88.27	86.42	83.33	95.07	94.24	98.25	98.52	96.57
HP3	37.07	65.52	97.41	93.97	91.38	93.97	93.99	93.10	93.10	82.76	84.48	96.55	98.40	98.44	99.76	99.09
HP4	30.77	61.54	94.87	89.74	87.18	87.18	89.99	88.46	88.46	88.46	85.90	89.62	95.00	87.71	96.53	95.61
AVG	43.68	54.40	89.16	91.78	91.09	89.70	**93.43**	90.58	**91.41**	81.64	86.96	93.88	95.51	94.72	**98.08**	96.66

(b) F-measure

Haskell	Threshold Based Methods							Unsupervised Methods				Supervised Methods				
	CLAMI	CLAMI+	ACL	TCLP	SQRT	CBRT	UMV	KMS	NGC	MBC	HC	NB	SVM	KNN	RF	C50
HP1	0.316	0.521	0.852	0.963	0.947	0.947	0.970	0.727	0.977	0.816	0.968	0.977	0.977	0.970	0.986	0.973
HP2	0.955	0.554	0.915	0.942	0.970	0.970	0.980	0.920	0.933	0.910	0.908	0.982	0.984	0.990	0.991	0.979
HP3	0.365	0.730	0.984	0.963	0.949	0.949	0.969	0.778	0.959	0.688	0.913	0.978	0.990	0.991	0.995	0.994
HP4	0.206	0.681	0.966	0.935	0.924	0.924	0.942	0.640	0.931	0.929	0.917	0.945	0.963	0.928	0.980	0.970
AVG	0.460	0.621	0.929	0.951	0.948	0.948	**0.965**	0.766	**0.950**	0.836	0.926	0.971	0.979	0.970	**0.988**	0.979

(c) MCC

Haskell	Threshold Based Methods							Unsupervised Methods				Supervised Methods				
	CLAMI	CLAMI+	ACL	TCLP	SQRT	CBRT	UMV	KMS	NGC	MBC	HC	NB	SVM	KNN	RF	C50
HP1	0.159	0.200	0.504	0.631	0.569	0.538	0.598	0.735	0.783	-0.179	0.685	0.792	0.792	0.783	0.917	0.838
HP2	0.742	0.317	0.684	0.728	0.826	0.805	0.828	0.420	0.549	0.689	0.239	0.850	0.832	0.958	0.958	0.916
HP3	0.227	0.451	0.723	0.796	0.702	0.807	0.810	0.766	0.766	0.642	0.391	0.918	0.947	0.972	0.973	0.973
HP4	0.166	0.389	0.770	0.688	0.665	0.675	0.697	0.640	0.640	0.636	0.547	0.774	0.890	0.640	0.886	0.925
AVG	0.323	0.339	0.670	0.711	0.690	0.706	**0.733**	0.640	**0.685**	0.447	0.465	0.834	0.865	0.838	**0.933**	0.913

width value of 0.83 (close to 1) indicates that the samples are correctly clustered. The following conclusions are inferred from Table 6.3, Fig. 6.4 and Fig. 6.5:

1. In Table 6.3, the maximum average performance of the models in each group (threshold-based, unsupervised, supervised) is shown in bold.

2. The performance of proposed defect proneness prediction technique UMV in terms of average accuracy, f-measure, and MCC on Haskell datasets are 93.43%, 0.965, and 0.733 respectively (as shown in Table 6.3). It is the highest in the two groups Viz. threshold-based and unsupervised methods. The highest performance values are shown in bold in the table. Supervised model, random forest (RF) performs best in comparison to all other models in terms of average accuracy (98.08%), f-measure (0.988), and MCC (0.933).

3. In Table 6.3, it can be seen that the average performances of TCLP, SQRT, and CBRT are almost equivalent in terms of accuracy, f-measure, and MCC. So, it can be stated that instead of natural log transformation, square root

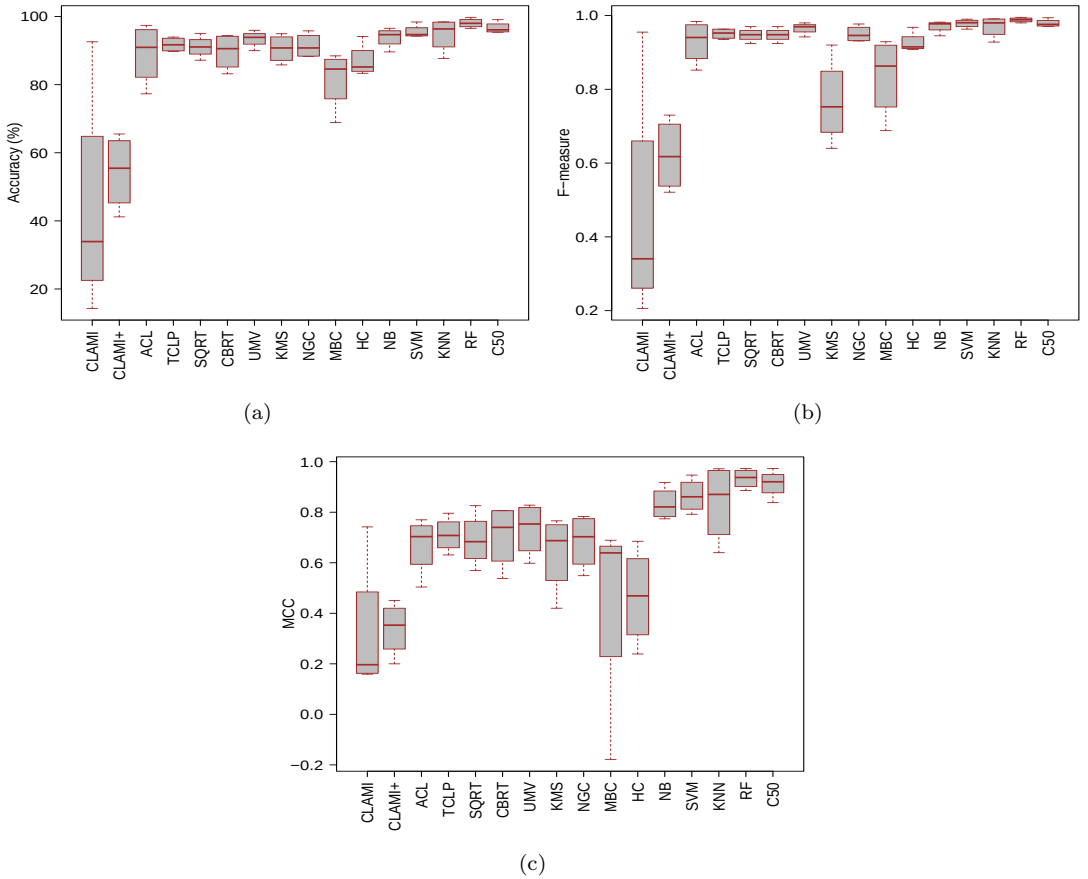

FIGURE 6.4: Boxplot presentation of the different models in terms of (a) accuracy, (b) f-measure, and (c) MCC

transformation, and cube root transformation can also be beneficial to build the SDP models. The proposed ensemble model UMV based on TCLP, SQRT, and CBRT gives a better average performance value (in terms of accuracy, f-measure, and MCC) with respect to other threshold-based and unsupervised techniques. Therefore, ensembling prediction results of these three threshold-based SDP models using majority voting is an effective approach for SDP in FP.

4. Fig. 6.4 shows that UMV has the highest median value in terms of accuracy (93.89%), f-measure (0.970), and MCC (0.753). Among the supervised models, RF has the maximum median value in terms of accuracy (98.01 %), FM (0.989), and MCC (0.938). Therefore, supervised model RF is an effective SDP model

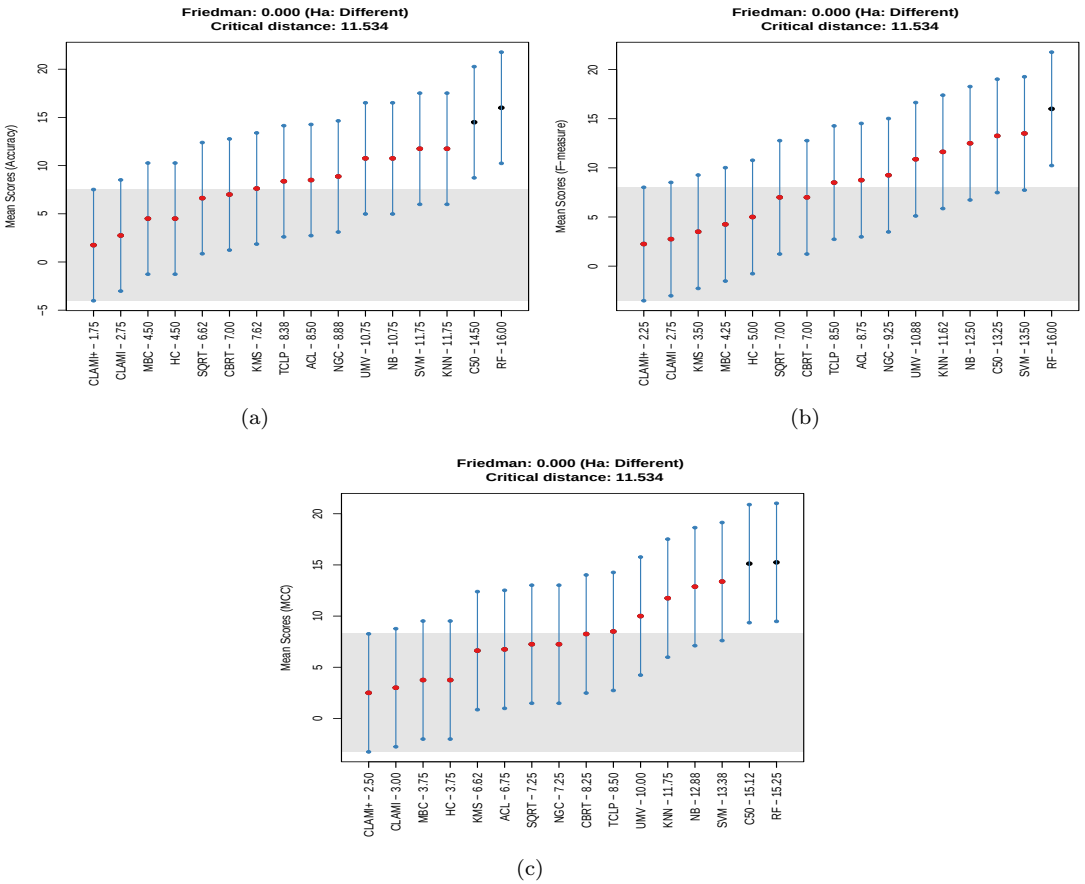

FIGURE 6.5: Critical diagram presentation of the different models in terms of (a) accuracy, (b) f-measure, and (c) MCC

for labeled datasets, and UMV is better for unlabelled datasets.

5. Fig. 6.5 has two horizontal areas; the lower (shaded) one shows the low-performing group, and the upper one shows the high-performing group. In terms of accuracy mean score (Fig. 6.5(a)), UMV (10.75) proves to be a statistically better model than other threshold-based and unsupervised models. CLAMI+ (1.75), CLAMI (2.75), MBC (4.50), HC (4.50), and SQRT (6.62) belong to the statistically low-performing group. CBRT (7.0) and KMS (7.62) are on the borderline of both groups, so it is not easy to deduce their performance. C50 (14.50) and RF (16.0) are the best models statistically among all the models.

6. In terms of FM mean score (Fig. 6.5(b)), UMV (10.88) shows statistically best

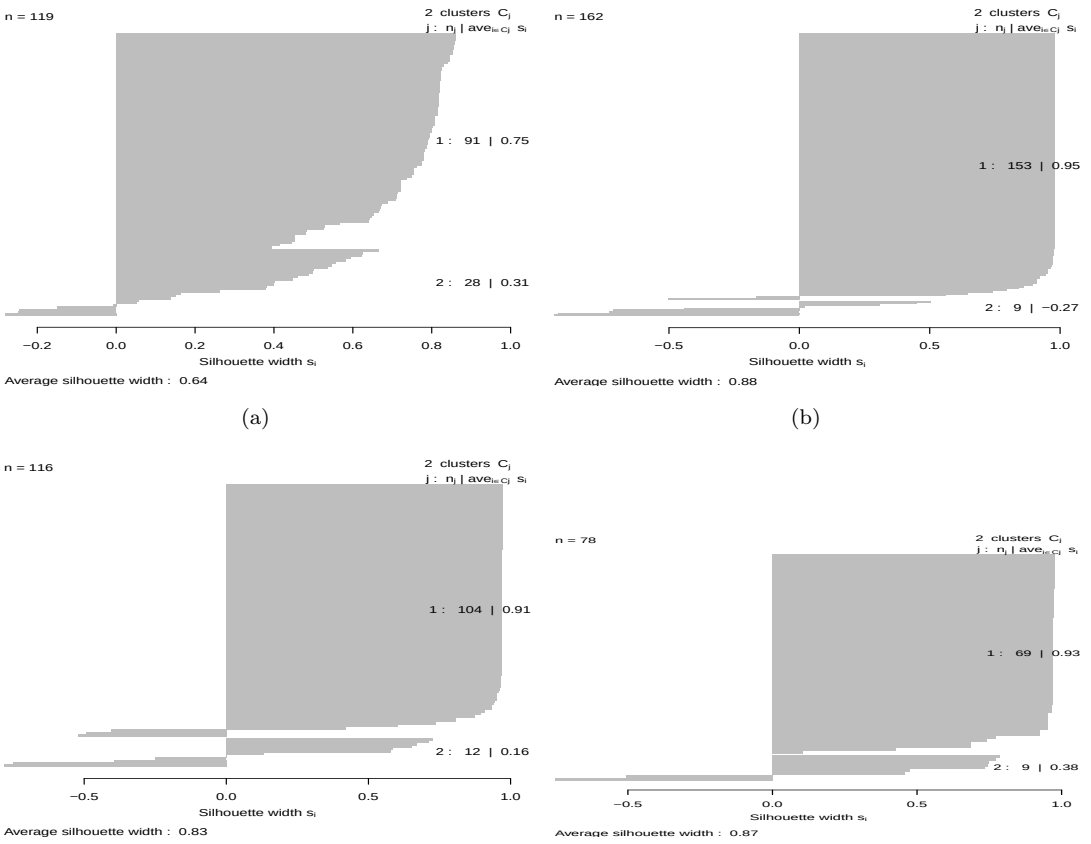

FIGURE 6.6: Average silhouette plot for clusters generated by UMV techniques for four Haskell packages (a). Etherium, (b) Hburg, (c) Mios, (d) Piet

models among threshold-based and unsupervised models. CLAMI+ (2.25), CLAMI (2.75), KMS (3.50), MBC (4.25), HC (5.0), SQRT (7.0), and CBRT (7.0) belong to the statistically low-performing group. RF (16.0) is the best model statistically among all the models.

7. In terms of MCC mean score (Fig. 6.5(c)), UMV (10.0) is the best model among the threshold-based and unsupervised techniques. CBRT (8.25) and TCLP (8.50) are almost at the border, so it is not easy to deduce whether they belong to low low-performing group or high high-performing group. However, C50 (15.12) and RF (15.25) come out as the best models amongst all the others and show significantly better results.

8. In Fig. 6.6, the silhouette plot of the proposed defect proneness prediction

technique UMV on four Haskell packages is shown. In Fig. 6.6, j is the cluster number, n_j is the number of data points in the cluster, and $ave_{i \epsilon C_j} S_i$ is the average silhouette width of the cluster. Outliers are shown as negative silhouette width values. The average silhouette width coefficient values of UMV on four Haskell datasets are 0.64, 0.88, 0.83, and 0.87 showing high goodness of cluster except 0.64 (Etherium). Based on these silhouette width coefficient values, we can say that UMV is creating good clusters for classifying the function as defective or non-defective.

9. Defect-wise box plots of functions predicted by UMV technique on four Haskell packages are shown in Fig. 6.7. In Fig. 6.7, functions with higher SM values are labeled as defect prone (second boxplot of each SM (blue color)), and those with low SM values are labeled as non-defective (first boxplot of each SM (red color)). From this figure, we can conclude that IND is not a suitable SM for building SDP model. The values of IND SM are overlapping, which means that these values do not play a significant role in deciding which function is defective and which is not. Besides IND, all other five SM (OUTD, CC, BRD, LOC, and TSA) values effectively play a role in predicting a function as defective or non-defective. This figure also shows that the above five SMs follow the defect proneness tendency. We have also found that CC and BRD metrics values of Hburg are overlapping. This overlapping condition is left to handle in the future.

Overall, in terms of accuracy, FM, and MCC, the proposed threshold-based approach UMV reported a statistically better SDP model for unlabelled datasets, and supervised models RF reported as the best SDP model for labeled datasets in FP. So, it can be concluded that the SM threshold can be a tool for SDP in FP.

6.5.1 Answer of Research Questions

RQ1: Is the performance of proposed UMV model on Haskell code better than standard supervised machine learning algorithms ?
Answer of RQ1: From Table 6.3, Fig. 6.4, and Fig. 6.5, we conclude that the proposed method UMV is not performing better than supervised models on Haskell code.

RQ2: Is the performance of proposed UMV model on Haskell code better than standard unsupervised learning algorithms and threshold-based models?
Answer of RQ2: From Table 6.3, Fig. 6.4, and Fig. 6.5, we conclude that the proposed method UMV is performing better than unsupervised models and threshold-based models on Haskell datasets.

RQ3: Are the extracted SMs capable of predicting the defect proneness in the functions of the Haskell?
Answer of RQ3: Based on obtained results in Table 6.3 and Fig. 6.4, it can be concluded that the five metrics (OUTD, CC, BRD, LOC, and TSA) are effective to predict defect proneness using UMV in functions up to a specific accuracy (93.43%), FM (0.965), and MCC (0.733).

6.5.2 Experimental Findings

Overall, based on stringent analysis, it is found that the proposed approach is the first initiative to investigate the SDP in FP (especially Haskell). It has the potential to identify a defect in functions based on extracted metrics of Haskell packages. Out of these six SMs, IND is not a suitable SM for SDP models which is concluded from defect-wise boxplot (Fig. 6.7). Two other transformation techniques, sqrt and cbrt also perform well on Haskell packages. So, we also have two other transformation techniques available other than log transformation for further investigation. To leverage the use of machine learning and ensemble approach, we have developed a generalized and robust SDP model UMV. UMV model improves the performance of

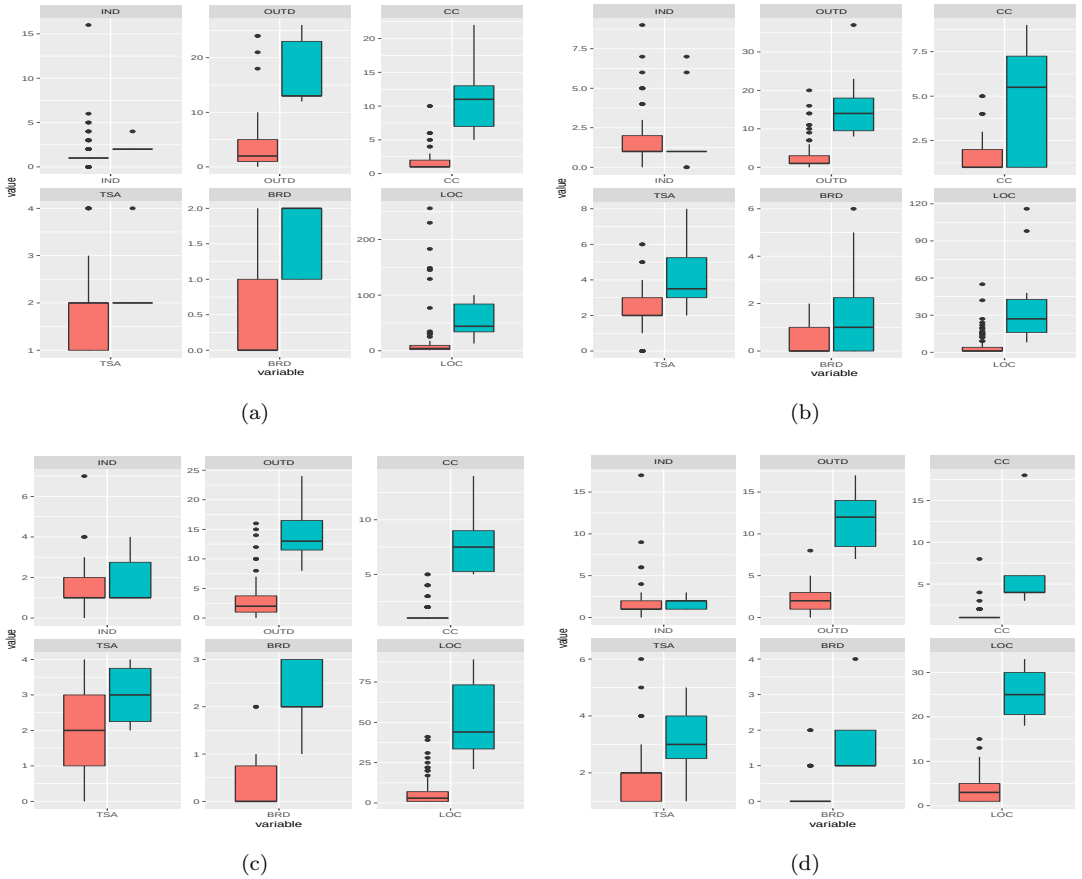

FIGURE 6.7: Defect-wise boxplot to represent distribution of software metric values of defective and non-defective instances in each DS for clusters generated by UMV technique on four Haskell packages (a). Etherium, (b) Hburg, (c) Mios, (d) Piet. Red boxplot (first) is for non-defective functions and blue boxplot (second) is for defective functions

individual threshold-based models. Still, there is a scope to propose new unsupervised techniques that can enhance the prediction performance even further in FP. Based on the critical diagram, UMV, with a mean score of 10.75 (accuracy), 10.88 (FM), and 10.0 (MCC), is the best model among threshold-based and unsupervised models. The average silhouette width coefficient (Fig. 6.6) of generated clusters is greater than 0.82 on Haskell datasets (except Etherium), concluding that UMV is making better clusters for binary classification. Defect-wise boxplot clearly shows that SM-threshold-based classification based on defect proneness tendency is effective in Haskell packages (Fig. 6.7). We have conducted Cohen's D test to calculate

the effect size between UMV and other SBP models and results shown in Table B.8.

6.5.3 Discussion

The proposed approach UMV is significantly superior in performance as compared to threshold-based and unsupervised techniques (Fig. 6.5). As we can see, SQRT and CBRT models are performing better than TCLP on Hburg dataset in terms of accuracy, FM, and MCC (Table 6.3). CBRT (0.807) is performing better than TCLP (0.796) on Mios in terms of MCC. So, there is a need to investigate an effective ensemble model that can significantly combine the performance of threshold-based models. We can see that the average performance of UMV is better than TCLP, SQRT, and CBRT in terms of accuracy, FM, and MCC (Table 6.3). From Fig. 6.5, it can be concluded that there is a need to enhance the performance of threshold-based and unsupervised machine learning models in terms of MCC because all these models belong to a significantly low-performing group (except UMV). These experiments are conducted on a limited number of SMs (6) and datasets (4). As shown in Fig. 6.6, based on average silhouette width, the proposed model is not performing better on Etherium (0.64), so it requires further investigation. From Fig. 6.7, we get that proper clusters are not formed by just using the IND SM. Overall, based on the above discussion, the proposed UMV method shows strong potential to predict software defects.

We have analyzed that the ACL method is more accurate than the UMV on the HP3 and HP4 projects. The threshold of ACL is calculated based on half of the average of each metric that shows ACL has a more effective threshold value to predict defective function in HP3 and HP4 projects, while UMV is an ensemble of models built using three different thresholds calculated using statistical parameters viz. mean and SD. From boxplot (Figure 6.4), overall performance of UMV is better compare to other unsupervised SDP including ACL. Moreover, this concept holds significant importance in evaluating the effectiveness of threshold-based SDP approaches for a given dataset, taking into consideration the dataset's statistical

characteristics. This is an area we explore to investigate further in the future across a broader spectrum of datasets. Also, in the future, we intend to extend our research by extracting more metrics for SDP in Haskell as well as other languages of FP.

6.6 Threat to Validity

The proposed work is an initial work based on related research that considered some existing risks to validate the complete systems. This work may suffer from the following risks:

Internal validity: The first threat is the consistency and limitation of the number of Haskell datasets. We have extracted six SMs from each Haskell package. They are created consistently to the best of our knowledge, but we cannot claim that the datasets are 100% accurate. The labeling of the datasets is totally subjective and depends on expert's knowledge and experience.

Construct validity: In this work, the developed SDP system predicts only whether the specific function is defective or non-defective but does not predict the number of defects. Only six SMs are considered to design the prediction system. Some of the other SMs can also be extracted to improve performance. So, it is a potential threat to validate the procedures.

External validity: In this research work, the prediction system is developed for the FP. In this research, developer expertise levels, developer types, developed software standards, history of the development system, and other stakeholders involved in the software project datasets development processes are not considered. Many human prospects related to dataset extraction, labeling, and the system development process also impact the reliability of Haskell datasets.

6.7 Conclusion and Future Work

In this research work, we have focused on developing software defect prediction (SDP) in the functional paradigm. We have analyzed that no defect datasets are available in the functional paradigm, so in the first phase, we have developed four functional paradigm datasets using four Haskell packages selected randomly. These datasets consist of six software source code metrics (CC, BD, LOC, TS, IND, and OUTD) values for each function of the Haskell packages. These software metrics represent internal characteristics of software source code and can help to predict defects in Haskell functions. Hence, these metrics are investigated to classify the Haskell functions into defect-prone and non-defect-prone based on statistics and machine learning models. In the second phase, a more accurate SDP model UMV is developed. The key concept behind UMV is to utilize transformation techniques to reduce the skewness of software metrics, derive an accurate SM threshold, and ensemble the prediction results. These SDP models can predict the defects in each function without considering the entire source code of the Haskell packages. These SDP models eliminate the need for building a control flow graph or abstract syntax tree of the entire source code. Since the software metrics are source code metrics not specific to Haskell packages, the SDP model should also work across other functional paradigm source codes. This can be further explored and verified for other functional paradigm defect datasets in future research.

The performance of the proposed technique UMV on four Haskell datasets shows better average performance (accuracy: 93.43%, FM: 0.96, MCC: 0.733) as compared to other threshold-based models or unsupervised machine learning models (Table 6.3). However, random forest algorithm (supervised model) with 10-fold cross-validation is the best model among all the considered models and also shows significantly better results in terms of all considered performance measures (Fig. 6.4 and 6.5). It was observed that the ACL method is more accurate than the UMV on HP3 and HP4 datasets. So, to deduce which threshold-based method will perform better on a specific dataset based on its statistical characteristics, more experiments

need to be performed in the future. However, these threshold-based methods give better average performance than the unsupervised machine learning SDP models.

To the best of our knowledge, this is the first research work that has developed a SDP system for FP. The proposed technique UMV is verified with an average silhouette coefficients width plot, defect-wise boxplot, and also compared statistically with existing techniques on four Haskell datasets.

We will develop more FP datasets to accomplish more comprehensive results in the future. UMV method can be implemented for other functional programming languages like Standard Meta Language (SML), List Processing Language (LISP), and Clean. The proposed UMV technique can also be used for refactoring and code review problems in software engineering.