Chapter 2

Related Work and Preliminaries

This chapter presents an overview of machine learning (ML), ensemble learning, and state-of-the-art SBP techniques. We discuss the related work as supervised SBP in Section 2.1, unsupervised SBP in Section 2.2, and SBP techniques for functional paradigm (FP) in Sections 2.3. Section 2.4 discusses preliminary, which includes software metrics in Section 2.4.1, datasets descriptions in Section 2.4.2, performance measures in Section 2.4.3, statistical test in Section 2.4.4 and baseline methodology used in this thesis are given in Section 2.4.5. Section 2.4.6 mentions the hardware and software used for experimental work performed in this thesis.

2.1 Related Work on Supervised SBP

The thesis mainly focused on significant and newer work using standard ML and ensemble algorithms for classification-based SBP and regression-based SBP. We have presented related work for classification-based supervised SBP in Section 2.1.1 and regression-based supervised SBP in Section 2.1.2.

2.1.1 Classification-Based Supervised SBP

We have already discussed the basic concepts and introduction to classification-based SBP in Section 1.3.1. We explained that SBP is a growing field in software engineering, and it is the most common SBP approach [45,55]. SBP is a method for identifying software module defects by analyzing software metrics as variables. SBP serves as a link between software bugs and software metrics, and various software metrics have been introduced for this purpose. A summary of SBP methods and software metrics used by different researchers is shown in Table 2.1.

Previously, many researchers have developed SBP models using standard ML algorithms, ensemble techniques, and deep learning [135]. Numerous research works in the literature focus on ensemble techniques for SBP. They also exhibit that ensemble learning can be used to improve the performance of ML and overcome the problem of overfitting [56]. Hence, the following paragraphs focus on the ensemble-based SBP with their limitations.

Zhou et al. [37] found that the performance of existing supervised SBP models was still far from being satisfactory, so they proposed a more advanced technique Deep forest-based defect prediction (DPDF) model. DPDF takes full advantage of ensemble learning and deep learning by transforming the RF into a layer-by-layer structure. They concluded that DPDF increases AUC value by 5% as compared to Deep belief network (DBN), NB, LR, RF, SVM, Deep logistic regression (DLR), and gcForest (deep forest model). But, average recall value 0.39 of DPDF was quite low.

Based on the NB classifier, Ji et al. [136] proposed a new Weighted NB method based on Information diffusion (WNB-ID) for SBP. The six methods, viz. Chi-square (CS), Information gain (IG), Gain ratio (GR), Symmetrical uncertainty (SU), ReliefF, and Information flow (IF), are investigated to choose the most suitable method for weight assignment based on FM.

Table 2.1: Summary of SBP methods and metrics used by different researchers

Citation	Dataset	SBP Methods	Software Metrics Used
[19]	Mylyn dataset (An eclipse plugin)	Bayesian Network (BN), Logisitcs, J48, Linear regression	56 novel micro interaction metrics, 42 source code metrics, 15 history metrics
[137]	34 DS collected from 10 open source projects (PROMISE DSs)	Naive Bayes (NB), J48, Support Vector Machine (SVM), Decision Table (DT), and Logistic Regression (LR).	20 object oriented metrics (DIT, WMC, RFC, CBO, LCOM, NOC, Ce, Ca, LCOM3, NPM, DAM, MOA, LOC, CAM, MFA, CBM, AVG-CC, MAX-CC, AMC, IC)
[31]	7 DS from NASA, 3 DS from Softlab	Transfer Naive Bayes (TNB), KNN filter	2 Mccabe metrics, 4 LOC metrics, 10 Halstead metrics, 1 branch count
[26]	Apache, Columba, Gaim, Gforge, Jedit, Mozilla, Eclipse, Plone, PostgreSQL, Subversion, JCP	Naive Bayes, SVM	Source code, complexity, meta data, change log, added delta, deleted data, dictionary, file name, new revision group metrics

Continued on next page

Table 2.1 continued from previous page

Citation	Dataset	SBP Methods	Software Metrics Used
[138]	HTTPCLIENT, LUCENE, RHINO, and JACKRABBIT	KNN, LR, RPART, svmRadial, tree-bag, RF	19 Code churn (CM), 25 dependency network (NM), change genealogy network (GM) metrics
[139]	3 Versions of Apache integration framework (AIF) PROMISE DS	Linear Regression, LR, ANN, RBFN (basic, gradient, hybrid), and PNN	Chidambar and Kemerer (CK) metrics: WMC, DIT, NOC, CBO, RFC, LCOM
[66]	Turkish Softlab DS (AR3, AR4, and AR5), NASA DS (CM1, KC1, KC2 , MW1, PC1)	Hybrid self-organizing map (HySOM) based on SOM and ANN, SSOM, CT, CO, QDK, CX, NB, RF, ANN	29 metrics in Turkish ds, 21 metrics in NASA DS, only 6 metrics selected (LoC, CC, Uop, Uopnd, Top, Topnd)
[116]	4 DS form NetGene (Http-client, Jackrabbit, Lucene, Rhino), and 3 DS from ReLink (Apache, Safe, Zxing)	CLA/CLAMI, THD and EXP, BN, J48, LMT, LR, NB, RF, and SVM	465 network and change genealogy metrics, class level metrics and 26 code complexity metrics

Continued on next page

Table 2.1 continued from previous page

Citation	Dataset	SBP Methods	Software Metrics Used
[61]	1,398 Projects (SourceForge and GoogleCode), 4 Eclipse projects 1 Apache project	Universal defect prediction model based on context-aware rank transformation method	21 code metrics, and five process metrics (Nrev, Nfix, AddedLoc, DeletedLoc, ModifiedLoc)
[140]	3 DS (ReLink), 5 DS (AEEEM), and 8 DS (PROMISE)	ACL models	26 code complexity metrics, 61 class level metrics, 22 object oriented metrics
[69]	Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla, PostgreSQL	CBS, LT, EALR	5 diffusion (NS, ND, NF and Entropy), 3 size (LA, LD and LT), FIX, 3 history (NDEV, AGE and NUC) and 3 experience (EXP, REXP and SEXP) metrics.
[14]	45 PROMISE repository datasets (DS)	5 Base classifiers LOGR, ANN, RBFN-RAN, RBFN-FCM, and RBFN KMC and 3 ensemble methods (BTE, MVE, NDTF)	20 source code object oriented (OO) metrics (DIT, WMC, RFC, CBO, LCOM, NOC, Ce, Ca, LCOM3, NPM, DAM, MOA, LOC, CAM, MFA, CBM, AVG-CC, MAX-CC, AMC, IC)

Continued on next page

Table 2.1 continued from previous page

Citation	Dataset	SBP Methods	Software Metrics Used
[67]	Licq, Seamonkey 1.0.1	K-means clustering and multi-layer perceptron model (MLP)	OO metrics (EncF, DIT, NOC, Coh, WMC, CBO , RFC)
[117]	14 Open source projects	Change-prone class prediction model CLAMI+	Five CK metrics: WMC, DIT, NOC, RFC, LCOM; Four Li and Henry metrics: MPC, DAC, NOM, SIZE2; and one traditional lines of code metric (SIZE1)
[62]	34 Public DSs (5 from AEEEM, 3 from ReLink, 10 from MORPH, 11 from NASA, and 5 from SoftLab group)	Simple logistic, LR, RF, BN, SVM, J48, and Logistic model tree (LMT)	20 OO metric, 26 code complexity metrics, McCabe's cyclomatic metrics, CK metrics

Continued on next page

Table 2.1 continued from previous page

Citation	Dataset	SBP Methods	Software Metrics Used
[141]	11 PROMISE DSs- Ant 5 versions, IVY, LUCENE, POI, TOMCAT, KC1, JEdit and 1 Eclipse project	3 Thresholds: ROC Curves, VARL (Value of an Acceptable Risk Level) and Alves rankings), BN, ANN, C4.5 and SVM, K-means and SOM	Source Code metrics (SLOC, CBO, RFC, WMC, LCOM, DIT and NOC)
[16]	5 DS from AEEEM, 12 DS from NASA, 3 DS from ReLink, 62 DS from PROMISE repository	Supervised method EASC, Unsupervised methods (ManualUp and ManualDown)	Class level metrics, function level metric, OO metrics, file level metrics,
[142]	19 Open source projects	L-RNN classifier, NB, ANN, LR, k-NN and C4.5	20 metrics (CK and OO metrics)
[17]	7 PROMISE, 3 SoftLab, 2 ReLink DSs	5 Algorithm: K-Means, CLA, CLAMI, X-means, PowerLaw Function	20 object oriented metrics, 26 file level metrics, 29 function level metrics

Continued on next page

Table 2.1 continued from previous page

Citation	Dataset	SBP Methods	Software Metrics Used
[49]	12 DS from NASA, tera PROMISE repository	BPDET, NB, DT, J48, PART, BN, LBoost, RF, Adaboost, MLP, SVM	22 software metrics, Halstead metrics, derived Halstead metrics, McCabe metrics

The final experimental results demonstrate that WNB-ID is an effective SBP model as compared to NB, SVM, LR, Random tree (RT), NB-ID, and State-of-the-art ensemble (STSE). But, the average performance of WNB-ID in terms of precision (0.69) is not satisfactory.

Balogun et al. [143] proposed a combination of SMOTE with bagging and boosting methods to predict software bugs. They experimented with two BCs viz. Decision tree (DT) and Bayesian network (BN) on NASA DSs and showed that their proposed approach outperformed independent BCs (DT and BN) with 86.81% accuracy and 0.93% AUC. Aljamaan and Alazaba [144] investigated the performance of two bagging and five boosting ensemble techniques for SBP on eleven NASA DSs. The results of tree-based bagging methods showed superiority over tree-based boosting algorithms. They also concluded that Adaboost algorithm showed the worst performance among all the tree-based ensemble methods. Rathore and Kumar [106] did an empirical study on the performance of 7 ensemble techniques viz. Dagging, Rotation forest, decorate, RealAdaBoost, MultiBoost, Grading, and ensemble selection with three BCs NB, J48, and LR on 28 DSs. Experimental results revealed that the Rotation forest with J48 BCs achieved the highest performance. Aforesaid three discussed papers [106, 143, 144] deal with homogeneous ensemble but have not investigated heterogeneous ensemble models (e.g. Voting ensemble).

Kulmala et al. [145] investigated the Least Square Support Vector Machines (LSSVM) with Linear, Polynomial, and Radial Basis Function (RBF) kernels for the SBP model. The experimental results show that LSSVM with polynomial kernel is similar to LSSVM with RBF but better than LSSVM linear, LR, NB, DT, MLP, SVM, and RF. However, they have not addressed the DS imbalance problem.

Turabieh et al. [142] investigated three feature selection (FS) techniques, viz. Binary Genetic Algorithm (BGA), Binary Particle Swarm Optimization (BPSO), and Binary Ant Colony Optimization (BACO) to enhance the performance of Layered recurrent neural network (L-RNN) for SBP. They concluded that L-RNN outperforms

existing methods in terms of area under curve (AUC). Manjula and Florence [102] proposed a hybrid approach by combining a Genetic algorithm (GA) for feature optimization with a deep neural network (DNN) to improve the performance of SBP model. The experimental results show that proposed technique performs better than NB, RF, majority vote (MV), etc. But, these two models [102,142] were implemented on only four DSs. The execution time of these proposed models was higher than ML algorithms because these models were based on optimization techniques and deep neural networks.

Wei et al. [146] solved the problem of SBP by proposing a novel model named Local tangent space alignment support vector machine (LTSA-SVM). LTSA is used to extract the intrinsic structure of low-dimensional data, and after dimension reduction, SVM is used to train the model. They concluded that LTSA-SVM improves the prediction accuracy and FM by 1-4% as compared to SVM and Local linear embedding SVM (LLE-SVM). But, LTSA-SVM leads to high time complexity for optimizing the parameters of the algorithm and large scale matrix eigen value decomposition of test DS [146].

Li et al. [147] focused on five ensemble learning algorithms implemented on 5 NASA DSs and concluded that RF is the best algorithm among J48, Adaboost, Bagging, Random subspace, and Vote for SBP. Mehta and Patnaik [148] discussed an improved ensemble technique for SBP. This work considered two significant issues: data imbalance problem and high dimensionality of bug DSs. They concluded that Extreme Gradient Boosting (XGBoost) and Stacking ensemble methods produced better results (more than 90% accuracy) than other standalone BCs and ensemble methods.

Aforesaid four papers used a small number of DSs for the model validation. Wei et al. [146] used only 6 NASA DSs, Manjula and Florence [102] used only 4 NASA DSs, Li et al. [147] used 5 NASA DSs, and Mehta and Patnaik [148] used only 4 NASA DSs.

Kumar et al. [55] 2017 implemented NN with five algorithms viz. Gradient Descent (GD), GD with momentum (GDM), GD with adaptive learning rate (GDA), Quasi-newton method (NM) and Levenberg Marquardt (LM). Further, they also investigated three ensemble methods, viz. Best Training Ensemble (BTE), Majority Voting Ensemble (MVE), and Nonlinear Ensemble Decision Tree Forest (NDTF). BTE is performing better than MVE and NDTF. Yucalar et al. [149] investigated the performance of 10 ensemble methods (AdaboostM1 (AB), Logic Boost (LB), Multiboost AB (MAB), Bagging (BG), RF, Dagging (DG), Rotation Forest (ROF), Stacking (ST), Multi scheme (MS) and Voting (VT)). They concluded that the rotation forest algorithm could be used as a competent SBP model. The other successful SBP models were RF, AdaBoost, Logic Boost, and voting methods. Papers described above have used majority voting ensemble technique viz. MVE [55], VOT [149] and MV [102]. But, they considered same weights for the BCs in the majority voting ensemble.

Tong et al. [45] proposed a novel SBP method, SDAEsTSE, which combines Stacked denoising autoencoders (SDAEs) and Two-stage ensemble (TSE) learning. In SDAEsTSE, the authors first used the deep learning phase to extract the deep representation from the software metrics, and then TSE was used to handle the class-imbalance problem. They concluded that SDAEsTSE that was applied on 12 NASA DSs showed significantly better results than RF, Bayesian networks (BN), NB, and AdaBoost neural network (AB_NN) in terms of FM, AUC, and MCC. Later, Pandey et al. [49] extended the work of Tong et al. [45] by proposing bug prediction using Deep representation (DR) and Ensemble learning (BPDET) model to improve the performance of SDAEsTSE. In BPDET Staked denoising auto-encoder (SDA) was used for the deep representation of software metrics, and two layers of ensemble learning (TEL) were used to deal with class imbalance problem. They concluded that BPDET is a stable SBP model and outperforms SVM, NB, BN, Multilayer perceptron (MLP), Pruning rule-based classification tree (PART), RF, bagging, AB and logistic boost in terms of f-measure, MCC, and AUC. These SDAEsTSE and

BPDET models used the deep learning architecture in two phases, so it took much execution time as compared to ML models.

Recently, Wan et al. [150] proposed a novel Self-paced ensemble of ensembles (SPE2) for handling class imbalance. SPE2 is an undersampling method that uses the concept of instance hardness. They compared the performance of SPE2 with 8 imbalance learning techniques viz. SMOTE, MAHAKIL, RUS, SMOTEBoost, SMOTEBagging, RUSBoost, UnderBAgging, Cascade, and concluded that SPE2 achieved a better and more significant FM performance than existing counterparts. Run time cost of SPE2 was acceptable, but the average FM performance value of 0.47 was not satisfactory.

The last three papers discussed above deal with the class imbalance problem in software bug DSs. But, they achieved quite a low performance viz. FM=0.41, MCC=0.290 [45], MCC=0.293 [49], FM=0.47 [150]. Also, some research works are affected by a weak validation of applying their proposed technique only on a small number of DSs [102,146–148]. Since only a few papers have used the majority voting ensemble technique [55, 102, 149], so we thought of exploring ways to improve this technique in order to enhance the performance of SBP. Few papers used different groups of DSs to validate their proposed methods [37, 136, 149]. But, the average performance presented by these papers was not satisfactory.

The above techniques were implemented on a limited number of DSs, and none had used the weighted majority voting ensemble technique. Therefore, we have proposed the weighted majority voting ensemble technique that is applied on 28 DSs with different granularity. This research focuses on the voting mechanism in ensemble of multiple classifiers to make a final prediction. Each classifier is considered equal and provides a single vote in the simple majority voting ensemble [151]. The majority of votes determines the final prediction, i.e., the class with maximum votes (the most recurring vote) determines the final prediction. While, in weighted majority voting, the classification methods have varying degrees of influence on the

final prediction. Each classifier is assigned a coefficient (weight), which is typically proportional to its classification results on a validation DS [151]. The final decision is made by adding up all weighted votes and choosing the class with the maximum aggregate. Hence, in Chapter 3, a novel weighted majority voting technique for ensemble learning is proposed that is presented with a strong validation.

2.1.2 Regression-Based Supervised SBP

Numerous approaches have been introduced to construct SBCV prediction models. These can be categorized into three main groups, viz. those based on ML algorithms, ensemble methods, and deep learning methods. These SBCV prediction models are often evaluated using widely recognized software bug DSs collected from the PROMISE repository. In the assessment of these models, performance measures such as Mean Absolute Error (MAE), Mean Relative Error (MRE), and Pred(l) are commonly used.

Researchers have employed various ML algorithms to construct SBCV prediction models, including Linear regression (LM) [78, 152], Support vector regression (SVR) [1, 74–76], Decision tree regression (DTR) [1, 75, 78], Negative binomial regression (NBR) [153], K-nearest neighbor regression (KNN) [75, 76, 78], Genetic algorithm (GA) [41, 154, 155], Poisson regression (PR) [153], Bayesian ridge regression (BRR) [75, 76, 78], Zero-inflated poisson regression (ZIP) [153], Zero-inflated negative binomial regression (ZINBR) [153], etc. are used by many researchers to build the SBCV prediction models. The performance of SBCV prediction models utilizing these algorithms is presented as follows.

Among the ML algorithms, Decision Tree Regression (DTR) emerged as the most effective predictor in comparison to LM, BRR, SVR, KNN, and Gradient Boosting Regression (GBR) [75, 76]. The SBCV prediction model developed using NBR exhibited superior performance than LM but required considerable time and

statistical expertise [156]. In a comprehensive investigation by Gao and Khoshgoftaar [153], eight SBCV prediction models (including PR, ZIP, NBR, ZINBR, and 4 versions of Hurdel Regression Models) were explored to select the optimal model for estimating the software project quality. Afzal et al. [154] presented empirical results of genetic programming (GP) on three industrial projects, demonstrating that GP is a statistically significant model in terms of Pred(l=0.25). However, GP necessitated high setup times, experimentation in configuring control parameters, and involved complex algebraic expressions.

Ensemble methods tackle the SBCV prediction problem by combining multiple weak ML algorithms to form a robust regression model. Rathore and Kumar [89] conducted an empirical study on commonly used ensemble methods, including stacking, boosting, bagging, rotation forest, and random subspace ensemble methods, using three base learners: LM, multilayer perceptron (MLP), and DTR. They concluded that the performance of Random subspace was superior, while Stacking exhibited the weakest performance among all ensemble methods. Additionally, they developed two heterogeneous ensemble techniques employing the linear regression combination rule (LRCR) and the Gradient boosting combination rule (GBCR) for SBCV prediction. Their findings showed that the GBCR method outperformed LRCR, LR, MLP, GP, NBR, and ZIP across 11 DSs [77]. Wu et al. [78] revisited the impact of SBCV prediction models using seven regression algorithms, namely BRR, DTR, GBR, LR, KNN, MLP, and SVR, over 31 PROMISE DSs. They concluded that Gradient boosting regression (GBR) and BRR performed better than other models. Nevendra and Singh [157] introduced the AdaBoost.R-ET algorithm for predicting SBCV in software modules. They experimented with 15 DSs and compared the performance of AdaBoost.R-ET with various AdaBoost models, such as AdaBoost.R, AdaBoost.R-RF, AdaBoost.R-GB, and AdaBoost.R-XGB. The results showed that AdaBoost.R-ET was the most suitable model for SBCV prediction.

Rathore and Kumar [41,155] conducted a comparative analysis of neural network (NN) and genetic programming (GP) for predicting SBCV over 10 PROMISE DSs.

Their findings revealed that NN performed better than GP on small DSs, while GP outperformed NN on larger DSs. Qiao et al. [1] proposed a novel deep learning model for SBCV prediction and compared its performance with SVR, Fuzzy-SVR, and DTR on two DSs (MIS and KC2). The results demonstrated that their proposed deep learning model achieved a reduction in Mean Squared Error (MSE) ranging from 3% to 13% and an improvement in squared correlation coefficient ranging from 2% to 27% compared to the other models.

In a recent study, Yu et al. [71] conducted an extensive experimental analysis involving 12 commonly used regression algorithms, three ensemble algorithms, one metric selection technique, two resampling methods, and one parameter optimization. The findings revealed that regression algorithms may not be accurate enough to predict the exact number of bugs in software modules, as they exhibited high Mean Absolute Error (MAE) and low Pred(l=0.3) values. As an alternative, the authors suggested employing SBCV prediction models to rank software modules instead of attempting to predict the precise number of bugs. This ranking approach can be useful for evaluating software quality and facilitating maintenance decisions.

The primary limitation of the aforementioned supervised learning methods for SBCV prediction is their reliance on labeled DSs. The process of labeling DSs is complex and time-consuming, requiring the expertise of a software professional with at least 15 years of experience. To address this challenge and enable the creation of an SBCV prediction model using unlabelled DSs, we have proposed the MTB/MTBP technique in Chapter 5.

Based on the aforementioned discussion on supervised SBCV prediction models, it was observed that the majority of authors [1, 71, 75–78] utilized standard ML algorithms such as LM, SVR, DTR, KNN, NBR, GA, BRR, and MLP to compare the results of their proposed methods. Consequently, we have also employed these algorithms as baselines to compare the results of our proposed approach, MTBP.

2.2 Related Work on Unsupervised SBP

2.2.1 Classification-Based Unsupervised SBP

Various research studies have been conducted to build SBP on the unlabelled DS using ML, specifically clustering algorithms [65–69]. Li et al. [34] presented a wide systematic review on the use of unsupervised learning methods for SBP. They found that Fuzzy CMeans and Fuzzy SOMs performed best amongst the 14 families of unsupervised learning models. Zhong et al. [87] proposed an unsupervised learning system developed using K-means (KMS) and Neural-Gas clustering (NGC) techniques on NASA's DS KC2. An expert evaluated the cluster mean and some statistical data of the group to label the modules as defective or non-defective [87]. Later, the expert-specified approach was used by Zhong et al. [158] on the JM1 and KC2 DSs. It was found that the K-means clustering algorithm performed better than Neural-Gas-based clustering algorithm in terms of the overall error rate. However, the results of this approach were subjective and dependent on software engineer's experience and capability. A constraint-based semi-supervised SBP system was developed using the K-means algorithm. It was found that this approach was more helpful for an expert to label the modules as compared to other unsupervised-based clustering methods [159]. Abaei et al. [160] proposed an SDP model using a semi-supervised hybrid self-organizing map (HySOM) which minimizes the role of experts for identifying defect-prone modules. The experimental results showed improvement in false negative rate (80%) and overall error rate (60%) with NASA DSs [160]. Xu et al. [86] presented a framework comprising 40 unsupervised SBP models. They analyzed and compared 40 clustering models for SBP on 27 project versions. These approaches always needed a software expert to label the instances or create clusters based on some characteristics to assign label to instances. Thus, these techniques are not fully automated. So, to overcome these limitations, threshold-based unsupervised SBP is proposed, and related studies on threshold-based unsupervised SBP methods are mentioned separately in the next subsection.

2.2.1.1 Threshold-Based Unsupervised SBP

Boucher et al. [141] investigated ROC curve, VARL (Value of an Acceptable Risk Level), and Alves rankings as three threshold calculation methods that can be used for SBP. They constructed and investigated a threshold-based fault prediction model using each of the aforesaid three techniques on 12 public DSs. They compared the performance with supervised ML and clustering-based models and concluded that fault prediction model using ROC curve is the best-performing approach among the three threshold calculation techniques.

Catal et al. [115] proposed a clustering and threshold-based approach (two-stage approach). Later, they concluded that one stage approach (threshold-based) is better than the two-stage approach. They decided the threshold of the software metrics based on literature study, past fault-prone modules, and analysis of the previous version of software projects. The given instances will be faulty if the metric value is greater than their corresponding thresholds. Nam et al. [116] proposed the CLA/CLAMI model as a state-of-the-art technique based on thresholds as median of software metrics. Yang et al. [140] proposed ACL (Average,Clustering and Labeling) approach to overcome the CLA/CLAMI results. This approach made a small improvement in CLAMI performance. Later Yan et al. [117] proposed CLAMI+ techniques after modifying CLAMI approach.

Further, **C**lustering and **La**beling (CLA) has two steps: Clustering instances and Labeling instances in the cluster. These two steps help to label or predict all instances as faulty or non-faulty in unlabeled DSs based on the threshold (median value) of each metric. Further, CLAMI extends CLA by adding two more steps of data refinement. These two steps are **M**etric selection and **I**nstance selection. In addition, they prepare a test DS from original unlabeled DS using only selected metrics in previous steps and build a prediction model on training DS using ML techniques and tested on filtered DSs.

It is analyzed that CLA/CLAMI approach requires improvement in the performance of the model and stringent validation on more number of DSs. We mitigate the aforesaid gaps by proposing an approach that does not need any human intervention. It is completely automated to label and predict the faulty classes on unlabelled DSs. The proposed TCL/TCLP method has improved the state-of-the-art results and is validated on 28 public DSs. The originally considered DSs are suffering from a class imbalance problem. So, we have also used the SMOTE technique to balance the DS and have analyzed the performance of TCL/TCLP with both imbalanced and balanced DSs. The proposed TCL/TCLP is described in Chapter 4.

2.2.2 Regression-based Unsupervised SBP

Unsupervised techniques have gained significant attention among researchers for the development of SBP systems [34]. These approaches offer advantages such as low construction costs, simplicity, and the absence of the need for labeled DSs [124]. Numerous classification-based unsupervised SBP models have been created using unlabeled DSs [34]. In this context, our focus lies specifically on unsupervised SBP techniques developed using software metric thresholds. The related work on threshold-based unsupervised SBP is already given in Section 2.2.1.1.

As far as our knowledge extends, no previous research has introduced an unsupervised approach for predicting the number of bugs (SBCV) in each module of a software system. Therefore, we have put forth an unsupervised SBCV prediction model MTB/MTBP. From the aforementioned related research, we found that existing metric threshold calculation techniques did not consider software metrics distribution and skewness [48, 140, 161]. All the existing threshold-based unsupervised techniques were used to develop the classification-based SBP model. Therefore, we have proposed an unsupervised regression-based SBP technique based on the derived software metric threshold. The derivation of the software metric threshold also considers the distribution and skewness of software metrics. The proposed MTB/MTBP is described in Chapter 5.

2.3 Studies on Classification-Based SBP in FP

The basic concept of the functional paradigm languages, their characteristics, applications, and a brief introduction is already discussed in the introduction Section 1.3.4. The following paragraphs detail the software metrics and existing SBP approaches.

Many researches have been done in the field of SDP in object-oriented paradigm (OOP). SDP models are used to identify defective software modules using software metrics. Software metrics are the features used to characterize the performance and reliability of software projects. The software metrics used for SDP are Chidamber and Kemerer's (CK) metrics [162], McCabe [163], Halstead metrics [164], change metrics [165], developer micro interaction metrics [166], code smell metrics [167], web metrics [168], network metrics [169], cohesion metrics [170] etc. Appropriate software metrics are crucial for building the SDP model. However, not so many empirical studies have been conducted to analyze the impact of functional programming languages on software quality (in terms of defects) [125]. The most available and valuable OO SMs are CK metrics proposed by Chidamber and Kemerer [162]. Object-oriented and imperative languages use the SMs to predict the defects, and several of those metrics can be used in FP as well [171]. Sherriff et al. [84, 85] empirically showed the relevance of SMs on SDP in Haskell (FP). However, in recent years there has been little work to explore the relationship between metrics and functional languages. Abajirov [125] analyzed the use of SMs for comparing the quality of software written in FP languages and imperative ones.

Khanfor et al. [81] presented the practical impacts of functional programming in the field of software engineering. Pankratuis et al. [172] claimed that although functional programming code is more compact, it requires higher programming and debugging effort. Le et al. [173] introduced a new set of mutation operators to perform mutation testing for functional programming languages. They provided

MuCheek prototype tool[1] to perform mutation testing for Haskell [173]. The study of Manjhi and Chaturvedi [174] showed that the Radial basis function neural network shows a significantly good reusability estimate of software components in FP.

Detailed studies on classification-based unsupervised SBP models using ML, specifically clustering algorithms, are provided in Section 2.2.1. The studies on classification-based unsupervised SBP models using software metrics threshold are provided in Section 2.2.1.1. From the related research, we found that some selected SMs are quite relevant and useful for SDP in FP. Therefore, We have proposed an SM threshold-based ensemble approach UMV for SDP on unlabeled DSs. It is used to identify the defect-prone functions in Haskell packages. The proposed UMV is described in Chapter 6.

2.4 Preliminary

"A weak background is a deadly thing, so make the strong one." - Robert Henri

In this section, we provide some important background information about SBP. We discuss software metrics in Section 2.4.1, DSs in Section 2.4.2, performance measures in Section 2.4.3, statistical test in Section 2.4.4, baseline methods in Section 2.4.5. Section 2.4.6 mentions the hardware and software used for experimental work performed in this thesis.

2.4.1 Software Metrics

SBP models are used to identify buggy software modules using software metrics (SMs). SBP is a bridge between SMs and software bugs. Software metrics play a crucial role in describing the internal characteristics of source code [126]. So, SMs are extracted directly from the source code snippet and are used to measure the reliability, quality, and design of the source code. The metrics used in the construction of our SBP model are inspired by the SMs used in OOP and procedural

[1]https://bitbucket.org/osu-testing/mucheck.git

paradigms. There are diverse SMs used for SBP. The most widely used SMs for SBP are OOP CK metrics suit, proposed by Chidamber and Kemerer's (CK) [162]. This suit contains basically six object oriented software metrics viz. Coupling between objects (CBO), Depth of Inheritance Hierarchy (DIT), Number of Children per Classes (NOC), Lack of Cohesion of Methods (LCOM), Response for Class (RFC), Weighted Methods per Class (WMC). The other metrics suit are McCabe [163], and Halstead [164] SMs. Some other SMs are also employed for SBP, like change metrics [165], developer micro interaction metrics [166], code smell metrics [167], web metrics [168], network metrics [169], cohesion metrics [170], etc. The selection of appropriate SMs is crucial for building an effective SBP model.

The list of available SMs groups-wise is shown in Table 2.2. Software metrics are used to understand the internal structure of the source code, like size, coupling, cohesion, inheritance, and complexity. These SMs of five groups are primarily used in classification-based SBP. In our experiments, we have utilized these five group SMs for classification-based SBP in Chapters 3 and 4. MORPH and AEEEM group software metrics have been utilized to build the regression-based SBCV prediction model in Chapter 5. Further, the six source code software metrics [132, 171] viz. Indegree (IND), Outdegree (OUTD), Cyclomatic Complexity (CC), Type Signature Argument (TS), Branching Depth (BD), and Lines of Code (LOC) extracted/derived solely from a given function in Haskell code snippets and utilized in Chapter 6.

Additionally, MORPH group provides a list of a few CK metrics that are particularly useful for both binary classification and bug count vector prediction. The MORPH group contains twenty-one important SMs that are necessary for the functioning of SBP.

The full form of 20 SMs of the MORPH group is given as follows. The software metrics for the MORPH group are WMC (Weighted Methods per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling between Object Classes), RFC (Response for a class), LCOM and LCOM3 (Lack of Cohesion

in Methods), CA (Afferent Couplings), CE (Efferent Couplings), NPM (Number of Public Methods), LOC (Lines of Code), DAM (Data Access Metric), MOA (Measure of Aggregation), MFA (Measure of Functional Abstraction), CAM (Cohesion among methods of class), IC (Inheritance coupling), CBM (Coupling between methods), AMC (Average Method Complexity), Max_cc and Avg_cc (Max and Average McCabe's Cyclomatic Complexity) [175, 176].

The full form of 17 SMs of AEEEM group are CBO (Coupling between Object Classes), DIT (Depth of Inheritance Tree), fanIn (Number of other classes that reference the class), fanOut (Number of other classes referenced by the class), LCOM (Lack of Cohesion in Methods), NOC (Number of Children), numberOfAttributes (noa), numberOfAttributesInherited (noai), numberOfLinesOfCode (noloc), numberOfMethods (nom), numberOfMethodsInherited (nomi), numberOfPrivateAttributes (nopra), numberOfPrivateMethods (noprm), numberOfPublicAttributes (nopua), numberOfPublicMethods (nopum), RFC (Response for a class), WMC (Weighted method count) [177].

2.4.2 Datasets Description

There are a number of software bug datasets (DSs) available to software researchers. We have used the following two categories of DSs:

1. Public DSs (benchmark DSs) collected from the public repositories in Chapters 3, 4, and 5. These Chapters 3, 4, and 5 use object-oriented and procedural programming paradigm datasets.

2. Novel functional paradigm DSs created from Haskell packages in Chapter 6. We have considered the functional paradigm only in this Chapter 6.

2.4.2.1 Benchmark Datasets

In this section, we introduce the DSs employed in the experimental segments of this thesis. These DSs are widely accessible and are frequently utilized in numerous

TABLE 2.2: Group-wise list of original software metrics in different datasets

AEEEM (Actual Metrics)	Code
cbo	A1
dit	A2
fanIn	A3
fanOut	A4
lcom	A5
noc	A6
numberOfAttributes	A7
numberOfAttributesInherited	A8
numberOfLinesOfCode	A9
numberOfMethods	A10
numberOfMethodsInherited	A11
numberOfPrivateAttributes	A12
numberOfPrivateMethods	A13
numberOfPublicAttributes	A14
numberOfPublicMethods	A15
rfc	A16
wmc	A17

MORPH (Actual Metrics)	Code
wmc	M1
dit	M2
noc	M3
cbo	M4
rfc	M5
lcom	M6
ca	M7
ce	M8
npm	M9
lcom3	M10
loc	M11
dam	M12
moa	M13
mfa	M14
cam	M15
ic	M16
cbm	M17
amc	M18
max_cc	M19
avg_cc	M20

NASA (Actual Metrics)	Code
LOC_BLANK	N1
BRANCH_COUNT	N2
CALL_PAIRS	N3
LOC_CODE_AND_COMMENT	N4
LOC_COMMENTS	N5
CONDITION_COUNT	N6
CYCLOMATIC_COMPLEXITY	N7
CYCLOMATIC_DENSITY	N8
DECISION_COUNT	N9
DECISION_DENSITY	N10
DESIGN_COMPLEXITY	N11
DESIGN_DENSITY	N12
EDGE_COUNT	N13
ESSENTIAL_COMPLEXITY	N14
ESSENTIAL_DENSITY	N15
LOC_EXECUTABLE	N16
PARAMETER_COUNT	N17
HALSTEAD_CONTENT	N18
HALSTEAD_DIFFICULTY	N19
HALSTEAD_EFFORT	N20
HALSTEAD_ERROR_EST	N21
HALSTEAD_LENGTH	N22
HALSTEAD_LEVEL	N23
HALSTEAD_PROG_TIME	N24
HALSTEAD_VOLUME	N25
MAINTENANCE_SEVERITY	N26
MODIFIED_CONDITION_COUNT	N27
MULTIPLE_CONDITION_COUNT	N28
NODE_COUNT	N29
NORMALIZED_CC	N30
NUM_OPERANDS	N31
NUM_OPERATORS	N32
NUM_UNIQUE_OPERANDS	N33
NUM_UNIQUE_OPERATORS	N34
NUMBER_OF_LINES	N35
PERCENT_COMMENTS	N36
LOC_TOTAL	N37

SOFTLAB (Actual Metrics)	Code
total_loc	S1
blank_loc	S2
comment_loc	S3
code_and_comment_loc	S4
executable_loc	S5
unique_operands	S6
unique_operators	S7
total_operands	S8
total_operators	S9
halstead_vocabulary	S10
halstead_length	S11
halstead_volume	S12
halstead_level	S13
halstead_difficulty	S14
halstead_effort	S15
halstead_error	S16
halstead_time	S17
branch_count	S18
decision_count	S19
call_pairs	S20
condition_count	S21
multiple_condition_count	S22
cyclomatic_complexity	S23
cyclomatic_density	S24
decision_density	S25
design_complexity	S26
design_density	S27
normalized_cc	S28
formal_parameters	S29

Relink (Actual Metrics)	Code
AvgCyclomatic	R1
AvgCyclomaticModified	R2
AvgCyclomaticStrict	R3
AvgEssential	R4
AvgLine	R5
AvgLineBlank	R6
AvgLineCode	R7
AvgLineComment	R8
CountLine	R9
CountLineBlank	R10
CountLineCode	R11
CountLineCodeDecl	R12
CountLineCodeExe	R13
CountLineComment	R14
CountSemicolon	R15
CountStmt	R16
CountStmtDecl	R17
CountStmtExe	R18
MaxCyclomatic	R19
MaxCyclomaticModified	R20
MaxCyclomaticStrict	R21
RatioCommentToCode	R22
SumCyclomatic	R23
SumCyclomaticModified	R24
SumCyclomaticStrict	R25
SumEssential	R26

SBP research.

The 28 standard DSs obtained from 5 repositories SOFTLAB [27], AEEEM [25, 28], NASA [178], ReLink [179], and MORPH [180] are shown in Table 2.3. We selected only 5 NASA DSs with common software metrics as features [181]. All of these 28 DSs are commonly utilized in Chapter 3 and Chapter 4 to evaluate the classification-based SBP and compare it with the existing techniques.

Bug% of each dataset defines the percentage of instances/modules in a dataset having at least one bug. It is calculated using (2.1).

$$Bug\% = \frac{\text{Number of instances having at least one bug}}{\text{Total number of instances}} * 100 \tag{2.1}$$

TABLE 2.3: Classification based 28 Software projects collected from five groups

Groups	Dataset	Project ID	#Metrics	#Modules	Bug%	Granularity
AEEEM	Euinox	SP1	18	324	39.81	Class
	JDT	SP2		997	20.66	
	Lucene	SP3		691	9.26	
	ML	SP4		1862	13.16	
	PDE	SP5		1497	13.96	
SOFTLAB	AR1	SP6	30	121	7.44	Function
	AR3	SP7		63	12.70	
	AR4	SP8		107	18.69	
	AR5	SP9		36	22.22	
	AR6	SP10		101	14.85	
NASA	CM1	SP11	38	327	12.84	Function
	MW1	SP12		253	10.67	
	PC1	SP13		705	8.65	
	PC3	SP14		1077	12.44	
	PC4	SP15		1287	13.75	
MORPH	Ant-1.3	SP16	21	125	16.00	Class
	Arc	SP17		234	11.54	
	Camel-1.0	SP18		339	3.83	
	Poi-1.5	SP19		237	59.49	
	Redaktor	SP20		176	15.34	
	Skarbonka	SP21		45	20.00	
	Tomcat	SP22		858	8.97	
	Velocity-1.4	SP23		196	75.00	
	Xalan-2.4	SP24		723	15.21	
	Xerces-1.2	SP25		440	16.14	
ReLink	Apache	SP26	27	194	50.52	File
	Safe	SP27		56	39.29	
	Zxing	SP28		399	29.57	

Chapter 5 proposed MTB/MTBP to address the regression-based SBP. So, we have utilized 22 public DSs in Table 2.4 chosen from PROMISE (MORPH)[2] group [90, 150, 182] and Eclipse (AEEEM)[3] group [52, 177, 183]. The brief information about the DSs, like number of software metrics, number of modules, number of buggy modules, bug% are given in Table 2.4. The percentage of modules having five or more than five bugs is represented as 5+Bug%. Similarly, 7+Bug% represents the percentage of buggy modules that have 7 or more than 7 bugs. From this 7+Bug% analysis, we find that number of modules that have 7 or more than 7 bugs are less than 1% in all the software projects (except Camel-1.6, Lucene-2.4, Poi-2.5, Poi-3.0, and Equinox). We find that quite a small percentage of buggy modules with

[2] `https://github.com/klainfo/DefectData`
[3] `https://bug.inf.usi.ch/download.php`

TABLE 2.4: Regression-based 22 benchmark datasets collected from two groups

Groups	Datasets	#Metric	#Module	#Bugs	Bug%	5+Bug%	7+Bug%	9+Bug%
MORPH	Ant-1.6	20	351	92	26.21	1.42	0.57	0.28
	Ant-1.7	20	745	166	22.28	1.61	0.54	0.13
	Camel-1.4	20	872	145	16.63	1.83	0.80	0.34
	Camel-1.6	20	965	188	19.48	2.59	1.76	0.93
	Ivy-2.0	20	352	40	11.36	0.00	0.00	0.00
	Jedit-4.2	20	367	48	13.07	1.36	0.81	0.54
	Jedit-4.3	20	492	11	2.24	0.00	0.00	0.00
	Lucene-2.4	20	340	203	59.70	12.35	4.11	2.64
	Poi-2.5	20	385	248	64.40	2.85	1.29	0.51
	Poi-3.0	20	442	281	63.57	3.39	1.80	1.58
	Prop-4	20	8718	840	9.64	0.34	0.14	0.06
	Prop-5	20	8516	1299	15.25	0.39	0.16	0.07
	Tomcat	20	858	77	8.97	0.11	0.00	0.00
	Xalan-2.5	20	803	387	48.19	0.62	0.25	0.12
	Xalan-2.6	20	885	411	46.44	1.13	0.11	0.11
	Xereces-1.2	20	440	71	16.14	0.00	0.00	0.00
	Xereces-1.3	20	453	69	15.23	1.55	0.88	0.44
AEEEM	Equinox	17	324	129	39.81	2.16	1.54	0.62
	JDT	17	997	206	20.66	1.40	0.80	0.20
	Lucene	17	691	64	9.26	0.29	0.14	0.14
	Mylyn	17	1862	245	13.16	0.11	0.11	0.05
	PDE	17	1497	209	13.96	0.47	0.20	0.20

7 or greater than 7 bugs exist in many software projects. Hence, if the prediction result of MTB/MTBP with a maximum seven number of bugs is comparable to the supervised ML models then MTB/MTBP can be useful for unlabeled DSs. If MTB/MTBP model can predict up to 7 number of bugs in each module of a software project, then it can increase the software reliability and reduce the maintenance effort of software with a good margin. Since a quite small number of modules have 7 or more than 7 bugs, therefore MTB/MTBP approach can be used to predict SBCV in the maximum number of modules. For example, 0.57% of buggy modules of Ant-1.6 have 7 or more than 7 bugs. So, we can say 100%-0.57%=99.43% of modules can be directly and correctly predicted by MTB/MTBP.

NASA dataset: It is a widely utilized DS in the field of SBP and is freely

available for download[4]. PROMISE dataset: Another commonly used DS in SBP, it is accessible through the PROMISE repository[5] [184,185]. Eclipse dataset: Multiple versions of this DS are available for download[3]. Student dataset: Primarily designed for academic purposes and developed by students[3]. Open-Source dataset: This compilation includes various open-source software projects such as Xylan, Lucene, Ant, Apache, KDE, Gnome, Mozilla, Openoffice, Klac, Kpdf, Kspread, Firefox, and more. It is accessible for download[3]. Others: These DSs comprise private or industrial data, including commercial Java applications or DSs from the banking sector. The AEEEM dataset was denoted by D'Ambros et al. [28]. The name comes from the first letter of its five projects, i.e., **A**pache Lucene (LC), **E**quinox (EQ), **E**clipse JDT Core (JDT), **E**clipse PDE UI (PDE), and **M**ylyn (ML).

Chapter 6 proposed UMV to address the classification-based SBP on newly created FP DSs using Haskell packages. These DSs are part of the contribution, so, the FP DS creation and their description are provided in Section 6.2 of Chapter 6.

2.4.3 Performance Measures

In this thesis, two types of SBP models are developed. **1.** classification-based SBP; **2.** regression-based SBP. So, our work employed classification and regression-based performance measures in Section 2.4.3.1 and Section 2.4.3.2, respectevely.

2.4.3.1 Classification-Based Performance Measure

The three performance measures, viz. accuracy, f-measure (FM), and Matthew's correlation coefficients (MCC), are employed in Chapters 3, 4, and 6. The thesis considered a convention that the buggy module as positive and non-buggy modules as negative instances.

To compare the performance of the proposed approach and other benchmark techniques, we calculated accuracy using (2.2), FM using (2.4), and MCC using

[4] `https://github.com/klainfo/NASADefectDataset/tree/master/OriginalData/MDP`
[5] `http://promise.site.uottawa.ca/SERepository/`

TABLE 2.5: Confusion matrix to classify whether a class is buggy or not

		Predicted	
		NO	YES
Actual	NO	TN	FP
	YES	FN	TP

(2.5) as performance parameters to compare the results of proposed and other baseline methods. Table 2.5 shows the confusion matrix used to determine whether a class is buggy or not. Where TP (stands for true positive) is defined as that instance where both actual and predicted function values are YES. TN (stands for true negative) means both actual and predicted function values are NO. FP (stands for false positive) means that the actual function value is NO, but the predicted function value is YES. FN (stands for false negative) means the actual function value is YES, but the predicted function value is NO.

$$Accuracy = \frac{TN + TP}{TN + FP + FN + TP} \tag{2.2}$$

$$Precision = \frac{TP}{TP + FP}; Recall = \frac{TP}{TP + FN} \tag{2.3}$$

$$F - measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{2.4}$$

$$MCC = \frac{TP.TN - FP.FN}{\sqrt{((TP + FP).(TP + FN).(TN + FP).(TN + FN))}} \tag{2.5}$$

Accuracy and FM are the two most often used performance metrics for SBP models. Precision is the proportion of accurately predicted buggy modules to all buggy modules predicted. FM is defined as the harmonic mean of precision and recall (2.3) [186]. The proportion of correctly predicted buggy instances to all actual buggy instances is defined as recall. For an imbalanced DS, FM is preferred over accuracy [187]. In evaluating binary classifiers, MCC is a statistical metric for evaluating performance that provides a more reliable, accurate, and informative score. MCC produces a high score only when the SBP model demonstrates excellent

performance across all four categories: true positive, true negative, false positive, and false negative [188].

2.4.3.2 Regression-Based Performance Measure

From the literature, we have found that 18 out of 24 studies used MAE, MRE, Pred(l), RMSE, completeness, etc., to evaluate the performance of regression models. So, the three performance measures, viz. Mean Absolute Error, Mean Relative Error, and Prediction at level l Error or Pred(l)_Error, are employed in Chapters 5.

Mean Absolute Error (MAE): It is used to show how close your prediction is to the actual value. It is defined as the difference between the predicted value of bug count ($\bar{Y}_i$) and the actual value of bug count (Y_i). MAE is calculated using (2.6).

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|(\bar{Y}_i - Y_i)\right| \tag{2.6}$$

Mean Relative Error (MRE): It is used to measure how good a measurement is, relative to the size of the module being measured. n is the total number of modules. MRE is calculated using (2.7). In some cases the value of Y_i can be zero, so, to avoid divide by zero problems, we have added one to the denominator of MRE [153].

$$MRE = \frac{1}{n}\sum_{i=1}^{n}\frac{\left|(\bar{Y}_i - Y_i)\right|}{(Y_i + 1)} \tag{2.7}$$

Prediction at level l Error or Pred(l)_Error: Pred(l) is defined as the number of modules (c) in a DS for which the MRE value is less than or equal to a certain threshold value (l) [189]. MacDonell (1997) [190] recommended to keep the threshold value l less than or equal to 0.3 to contemplate an acceptable bug count prediction model. It is calculated using (2.9).

$$Pred(l) = \frac{c}{n} \tag{2.8}$$

$$Pred(l)\_Error = 1 - \frac{c}{n} \tag{2.9}$$

2.4.3.3 Boxplot Presentation

We have represented comparative results using boxplot in Chapters 3, 4, 5, and 6. The boxplot visualization is employed, providing a standardized representation of result distribution. Within the boxplot, a dark horizontal line represents the median value, while the rectangular box represents the interquartile range (IQR). The minimum and maximum results are denoted by the bottom and top edges of the box, respectively. Outliers are indicated by dots. A boxplot with a smaller IQR, a higher median value, and fewer outliers indicates a superior model in terms of results.

2.4.4 Statistical Test

For strong comparison, we employed many statistical tests to identify statistically significant differences in the overall performance of distinct methods. Wilcoxon signed-rank conducted in Chapters 3, 4, and 5. Nemenyi test and Cohen's D test conducted in Chapters 3, 4, 5, and 6.

Wilcoxon Signed-Rank Test: We utilized the Wilcoxon signed-rank test to conduct pairwise comparisons [191]. For the Wilcoxon signed-rank test, the null hypothesis H_0 states that if the p-value is greater than 0.05, then there is no significant difference between the techniques. The two techniques are significantly different if the p-value is less than 0.05 (alternate hypothesis H_a).

Nemenyi Test: The Nemenyi test [192] initially employs the Friedman test [193] as recommended by Demsar [194] to decide if there is a significant difference in the performance of the techniques. It then utilizes the Nemenyi test to group the techniques based on similar performance. Two techniques are considered significantly different if their mean scores exhibit a difference greater than the computed critical distance (CD). The CD is calculated by the number of techniques and DSs

used, as well as the critical value corresponding to a specific significance level (p-value). The CD is derived from the studentized range statistic and can be observed in standard statistics textbooks ([195]).

Friedman's Test: Friedman's test is a non-parametric statistical test employed for testing numerous hypotheses. It allocates scores to the techniques based on their performance on each DS individually. In this scoring scheme, the technique with the poorest performance is given a score of 1, the second worst technique gets a score of 2, and so on. In cases where there is a tie, mean scores are allocated [193].

Cohen's D Test: The p-value suggests only whether or not there is a statistically significant difference between the two models. On the other hand, the effect size can tell us how large this difference actually is. The appropriate way to calculate the effect size is through standardized mean differences. The mean difference increases in proportion to the effect size [196]. Cohen's D effect size is defined using (2.10). Where μ_{s1} and Ω_{s1} are the mean and standard deviation of sample 1 ($s1$).

$$\Phi = \frac{\mu_{s1} - \mu_{s2}}{\sqrt{(\Omega_{s1}^2 + \Omega_{s2}^2)/2}} \tag{2.10}$$

The assessment of effect size magnitude Φ is carried out using the thresholds defined in [197]. The significance is determined based on the thresholds specified as follows: $|\Phi| < 0.2$ is considered as 'negligible,' $|\Phi| < 0.5$ is considered as 'small,' $|\Phi| < 0.8$ is considered as 'medium,' and $|\Phi| > 0.8$ is considered as 'large' [197]. Furthermore, even if the p-value indicates a significant difference between the two models, the observed differences will be considered negligible if the effect sizes Φ of the two models do not differ by at least 0.2.

2.4.5 Baseline Methods

This section briefly discusses state-of-the-art techniques for different domains, classification-based SBP, and regression-based SBCV prediction. We compared the

performance of our proposed techniques with these benchmark/ baseline techniques.

Baseline Methods-1: As the first baseline methodology, we have chosen five supervised ML algorithms. These are K-nearest neighbor (KNN) [198], Naive Bayes (NB) [199], Support vector machine (SVM) [200], Random Forest (RF) [201], and C5.0 (C50) [202]. Then, we compared the results obtained using these conventional supervised algorithms independently with those obtained using the proposed technique and described in Chapter 3. The findings from this comparison were utilized to address RQ-1.

Baseline Methods-2: We have developed four SBPs using unsupervised ML algorithms, viz., KMS, NGC, MBC, HC, and three threshold-based methods CLAMI, CLAMI+, and ACL. We have compared the results of the proposed techniques with these four unsupervised algorithms and three threshold-based methods in Chapters 4. This comparison answers RQ-2.

Baseline Methods-3: As the third baseline methodology, we selected results from recent research articles already published and standard machine learning algorithms. We compared the effectiveness of the proposed technique with the results of best-performing algorithms in all Chapters 3, 4, 5, and 6. This comparison answers RQ-3.

Baseline Methods-4: As the fourth baseline methodology, we developed the proposed technique with sampling techniques, viz. ROSE, SMOTE, and RUS. We have compared the results of these three models to answer RQ-4.

Baseline Methods-5: We have compared the SBCV prediction performance of MTB/MTBP with the eight Standard ML models trained using labeled DSs, which are 8 supervised ML regression models. These ML models are Linear regression (LM), Multi-layer perceptron (MLP), Tree Models from Genetic Algorithms (GA), Decision tree regression (DTR), Support Vector Machines with RBF Kernel (SVR), K-nearest neighbor (KNN), Bayesian Ridge Regression (BRR) and Negative

binomial regression (NBR) and described in Chapter 5. This comparison answers RQ-5.

Baseline Mehtods-6: 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 DS and compare the results to answer the RQ-6. These are discussed in Chapter 6.

2.4.6 Hardware and Software Used

All experiments of this thesis are conducted on personal computer systems. The configuration of the processor used to implement our research work is a Core i7, 2.11 GHz Intel with 8 GB RAM and 256 GB SSD in a Windows environment. Majority of the baselines algorithms are implemented in Rstudio[6] with R programming language version 3.5.1. Particularly, we executed these algorithms (baseline approaches) using their default parameters by utilized the caret package from the R library [203].

This chapter studied the state-of-the-art approaches in the disciplines of SBP. It also examined the challenges and issues of these domains. This chapter also discussed the software metrics, benchmark databases, performance measures, statistical tests, and baseline methods utilized in the overall thesis. Finally, the hardware and software required to conduct the experiment were also explained.

[6] `https://www.rstudio.com/`