Chapter 5

An Unsupervised SBCV Prediction Technique Based on Selected Software Metrics

This chapter focuses on the third contribution of this thesis detail in Section 1.7.3. Section 5.1 demonstrates the proposed work to develop the regression-based unsupervised SBP. The experimental setup is given in Section 5.2. Section 5.3 covers the experimental results and analysis obtained using the proposed technique. Section 5.4 reports the explainability and interoperability, and Section 5.5 reports the threat to validity associated with the chapter. Section 5.6 focuses on the conclusion and future scope of this chapter. Detailed related work is provided in Section 2.2.2.

In a growing enterprise, the size of software systems continuously expands to meet user requirements, resulting in proportionate increases in complexity [1]. Subsequently, a higher likelihood of software bugs that obstruct meeting the software's requirements and specifications, potentially leading to software failures and reduced reliability [3]. Consequently, the software quality assurance team allocates substantial resources, primarily software testers, to activities such as code review, software testing, and bug detection, accounting for approximately 80% of the total software

cost and considerable time [4]. So, SBP provides a solution by enabling developers and testers to focus on modules with higher bug counts and help to optimize test resource allocation [119]. However, predicting bug counts is more advantageous than solely identifying buggy modules, aiding software managers in decision-making and timely software releases, thereby mitigating potential losses or issues related to software bugs [12, 71, 222]. Software Bug Count Vector (SBCV) prediction technique is a machine learning model that predicts the number of bugs or bug count in a module or code area [44].

Numerous SBCV prediction models have been developed based on labeled datasets, employing various machine learning algorithms such as linear regression, multi-layer perceptron, decision tree regression, k-nearest neighbor, support vector regression, deep neural network, and ensemble regression [1,71,77,118]. However, these models' effectiveness diminishes when applied to new software projects due to disparities between the distributions of training and testing datasets, necessitating labeled data for model training [34]. The challenge of collecting and labeling software datasets with bug count information further emphasizes the need for an unsupervised SBCV prediction model, prompting the proposal of the MTB/MTBP model designed for unlabeled datasets. This model encompasses five critical steps: unsupervised metric selection, threshold derivation, bug count vector generation, machine learning model construction, and bug count vector prediction.

Fig. 5.1 demonstrates all the steps involved to implement the regression-based unsupervised SBCV prediction model MTB/MTBP. Fig. 5.1 consists of five key blocks viz. unlabeled input datasets creation, metric selection, metric threshold derivation, last step of MTB (bug count generation), and last two steps of MTBP (building machine learning models and predicting bug count vectors). Other blocks show performance evaluation of MTB/MTBP and statistical analysis of results predicted by MTB/MTBP.

This empirical study aims to introduce a novel technique, MTB/MTBP, that

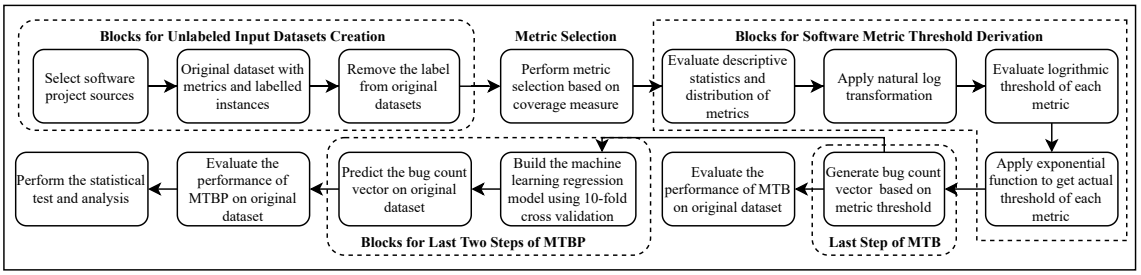


FIGURE 5.1: Step-by-step block diagram illustrating the development process of the proposed SBCV prediction model (MTB/MTBP)

can predict SBCV on unlabeled datasets in an automated manner. In this experimental study, the MTB/MTBP technique reaches promising prediction performance in terms of average mean absolute error (MAE) of 0.45, average mean relative error (MRE) of 0.20, and average prediction error at level l=0.3 (Pred(l)_Error) of 0.29 (for performance parameters definition see Subsection 5.2.3). In addition, MTB/MTBP technique outperforms or gives comparable performance to the supervised machine learning models with statistical significance. These promising results show that our MTB/MTBP technique is potentially suitable for SBCV prediction on new software projects.

5.1 Proposed Approach: MTB/MTBP

The proposed approach MTB/MTBP is an unsupervised software bug count vector prediction model based on an unlabeled bug dataset. The key idea of our proposed technique is to select the effective software metrics using the unsupervised metric selection (UMS) technique and assign a bug count value to each module based on the derived software metric threshold. MTB accomplishes this key idea using the following key steps: 1. Unsupervised **M**etric selection (UMS); 2. **T**hreshold derivation; and 3. **B**ug count vector (BCV) generation based on the threshold. These three steps avail to label all instances with bug count values in unlabeled datasets. Furthermore, MTBP stands for MTB **P**lus. MTBP has two additional steps: 4. Building machine learning models; and 5. Predicting bug count values or

number of bugs in each module of software projects. The following five subsections describe each step of MTB/MTBP.

5.1.1 Unsupervised Metric Selection (UMS) Method

We have used the space-filling concept [223], also known as coverage measure, to select a small informative subset of software metrics to reduce redundancy in the dataset. The available software bug dataset is high dimensional and has redundancy in the software metrics. High dimensional datasets consume much time in building the machine learning model and cause the curse of dimensionality problems. Gao et al. [91] suggested that top $\lceil log_2 m \rceil$ software metrics among the ranked features (total m metrics) are sufficient to build the prediction model [92]. In the literature, two types of software metric selection techniques are available, supervised and unsupervised. We are working on unsupervised technique, so, we have introduced an unsupervised metric selection technique based on coverage measure [223]. This technique has not been explored up to now for software metric selection and software bug prediction. A smaller coverage value indicates that the hypercube is well filled, coverage measure value of zero or near zero shows that data points are distributed as a regular grid or uniformly distributed in the data space. To identify the uniformity in a hypercube, point-to-point uniformity measures based on the distance between pairs of points are used. The best subset gives the smallest coverage measure. Since, the coverage measure is based on pairwise distances, if the data points are large in number then it requires more time and memory to calculate the coverage measure of each subset of dataset. Coverage measure (Ψ) based on point-to-point uniformity measure is defined as follows:

Let there be n number of classes/instances/modules and m number of metrics in a software project (SP). Hence a metric vector X_k is defined using (5.1).

$$X_k = \{x_i\}_{i=1}^{n} \quad \forall \quad k = 1, 2, ..., m \tag{5.1}$$

Then we can compute the minimum Euclidean distance δ between the instance value x_i and any other instance value x_j using (5.2).

$$\delta_i = \min_{i \neq j} |x_i - x_j| \tag{5.2}$$

Then the coverage measure Ψ is computed using (5.3).

$$\Psi = \frac{1}{\overline{\delta}} \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(\delta_i - \overline{\delta}\right)^2} \tag{5.3}$$

Where $\overline{\delta}$ is the mean of δ_i and computed using (5.4).

$$\overline{\delta} = \frac{1}{n} \sum_{i=1}^{n} \delta_i \tag{5.4}$$

For a perfectly uniform mesh, $\delta_1 = \delta_2 = \delta_3 = ... = \delta_n = \overline{\delta}$. Hence, $\Psi = 0$. Thus a smaller value of Ψ shows that the mesh is more uniform. Minimum euclidean distance between points has been used to measure the quality coverage of values. The value of Ψ is directly proportional to the distance between the data points. A smaller value of Ψ shows that the design is closer to a regular grid.

For all subsets of metrics, Ψ is computed using (5.3). Using search forward selection techniques (SFS) the best subset of metrics has the smallest Ψ value. SFS is used to add metrics in a step-by-step way on the basis of the obtained coverage value. The best subset of metrics produces the smallest coverage measure value. This approach does not require any parameter or threshold value.

The pseudo-code for the aforementioned unsupervised metrics selection technique is presented in Algorithm 7. This algorithm takes an unlabeled dataset $X_k[x_i]_{i=1}^{n}$ as input and produces the output (metric ID M_{id} and coverage measure CD) for each subset of software metrics.

Fig. 5.2 shows the coverage measure value versus added metrics generated using

Algorithm 7: An unsupervised metric selection technique

Input: X, a $n \times m$ matrix representing n instances and m metrics
Output: $CD[l]$, an array storing the minimum Ψ values
Output: $M_{id}[l]$, an array storing metric IDs corresponding to minimum Ψ values

```
CD[l] ← new array ;            // Initialize array for minimum Ψ values
M_id[l] ← new array ;                  // Initialize array for metric IDs
for l ← 1 to m do
    δ_i ← new array of size n ;                    // Store minimum distances
    for i ← 1 to n do
        δ_i[i] ← ∞
        for j ← 1 to n do
            if i ≠ j then
                distance ← 0
                for k ← 1 to m do
                    distance ← distance + (X[i,k] − X[j,k])^2
                end
                distance ← sqrt(distance)
                if distance < δ_i[i] then
                    δ_i[i] ← distance
                end
            end
        end
    end
    mean_delta ← 0
    for i ← 1 to n do
        mean_delta ← mean_delta + δ_i[i]
    end
    mean_delta ← mean_delta/n
    sum_variance ← 0
    for i ← 1 to n do
        sum_variance ← sum_variance + (δ_i[i] − mean_delta)^2
    end
    Ψ ← 1/mean_delta × sqrt(sum_variance/n)
    CD[l] ← Ψ ;                          // Save the minimum value of Ψ
    M_id[l] ← l ;                    // Save the corresponding metric ID
end
return CD, M_id
```

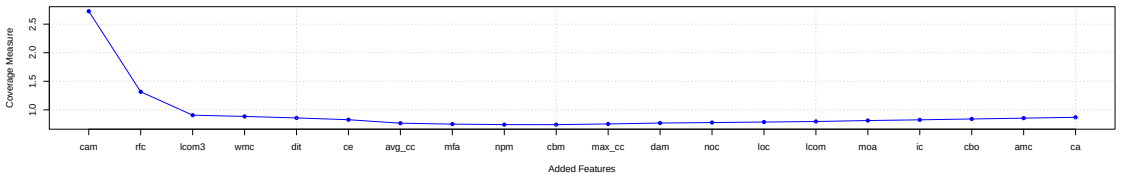


FIGURE 5.2: Coverage measure versus added metrics for Ant-1.6 dataset

Algorithm 7 for Ant-1.6 dataset. The cam metrics has minimum coverage measure of 2.7, then when rfc is added, we get a reduced coverage measure of 1.3 and so on. The minimum value of the coverage measure is 0.74 for the subset of cam (2.7), rfc (1.3), lcom3 (0.9), wmc (0.88), dit (0.85), ce (0.82), avg_cc (0.76), mfa (0.74), npm (0.74), and cbm (0.74)). But, when we further add more metrics like max_cc (0.75), dam (0.77), ... ca (0.86) then the coverage measure value starts increasing. So, we have ranked these software metrics based on the coverage measure obtained using UMS algorithm and have selected the top p software metrics to build the MTB/MTBP model.

5.1.2 Threshold Calculation

After selecting the effective software metrics, the threshold calculation of each software metric is an important step. The threshold of each software metric helps to label the instances with the number of bugs (bug count value). In order to calculate the appropriate threshold, a number of metric threshold calculation techniques are available [47, 48, 140]. We have used the natural logarithmic transformation method to calculate the metric threshold [47]. In this natural log transformation method, firstly, we apply natural log transformation to reduce the skewness of software metrics using (5.5), then the addition of the mean using (5.6) and standard deviation using (5.7) of each metric is used as the logarithmic threshold of each metric calculated using (5.8). We calculate the actual metric threshold (T_m) by taking the

exponential transformation using (5.9) of the logarithmic threshold vector ($T_m^{'}$).

$$Log(x) = ln(x+1) \tag{5.5}$$

The logarithmic value of zero is not defined. So, to apply the log transformation to the software metric, we have added one to the original dataset as (x+1) using (5.5).

$$\mu_m = \frac{\sum_{i=1}^{n} x_i}{n} \tag{5.6}$$

$$\Omega_m = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu_m)^2}{n-1}} \tag{5.7}$$

$$T_m^{'} = \mu_m + \Omega_m \tag{5.8}$$

$T_m^{'}$ is the logarithmic threshold of transformed data. To calculate the actual threshold of software metrics, T_m can be calculated using (5.9).

$$T_m = exp(T_m^{'}) \tag{5.9}$$

5.1.3 Software Bug Count Vector (SBCV) Generation

After calculating the threshold of each metric, the bug count vector generation is the core idea. The bug count vector is generated using the following bug proneness concept.

Bug proneness concept: Higher complexity in the source code causes more bug proneness [177, 224, 225]. The bug count value assigned to each instance is based on the fact that a higher metric value causes the instance to be bug-prone [176, 226, 227].

Accordingly, the aforesaid bug proneness concept is utilized here as follows: if the software metric value is greater than the corresponding threshold value, then it may cause the module or instance to be bug-prone.

TABLE 5.1: BCV generation using software metric threshold example

(a) Actual metric value						(b) Binary metric value					BCV
Inst.	M1	M2	M3	M4	M5	M1	M2	M3	M4	M5	
I1	11	18	106	1	9	1	0	0	0	1	2
I2	4	13	76	4	4	0	0	0	0	0	0
I3	1	20	101	8	11	0	0	0	1	1	2
I4	8	26	185	7	5	1	1	1	0	0	3
I5	9	40	119	11	7	1	1	1	1	1	5
I6	3	28	95	6	4	0	1	0	0	0	1
I7	7	19	145	9	10	1	0	1	1	1	4
Thr.	4.75	21.29	109.86	7.16	5.95	4.75	21.29	109.86	7.16	5.95	

Example: Let us consider a Dataset $X_k[x_i]_{i=1}^n$ for $i = 1, 2, .., n$ instances and $k = 1, 2, .., m$ metrics. Firstly, we have selected top p metrics out of m metrics using Algorithm 7 (Subsection 5.1.1). Now we have reduced dataset $X_k[x_i]_{i=1}^n$ for $i = 1, 2, .., n$ instances and $k = 1, 2, .., p$ metrics. Then we calculated the threshold vector $TV[1:p]$ using (5.9). Thereafter, based on calculated $TV[1:p]$, we generated bug count vector $BCV[1:n]$ on the basis of bug proneness concept. For example, Table 5.1a has seven instances (Inst.), five selected metrics (p), and a corresponding threshold (Thr.). Using Table 5.1a, a binary metric value Table 5.1b and then bug count vector (BCV) is generated. Binary metric value table consists of 0 or 1. Zero represents the metric value is less than the corresponding threshold, and one shows the metric value is greater than the corresponding threshold value. Later, we added all the binary values instance-wise to generate the bug count value for each instance.

Thus, software metric selection, threshold calculation, and bug count vector generation for the unlabelled dataset are successfully completed. These three steps are known as MTB (**M**etric selection, **T**hreshold derivation, and **B**ug count vector generation) model. The pseudo-code of the proposed approach MTB is written as Algorithm 8. The additional two steps are explained in the next two subsections.

Algorithm 8: To implement MTB (**M**etric selection **T**hreshold calculation and **B**CV generation) technique

Input: A dataset $X_k[x_i]_{i=1}^n$ for $i = 1, 2, .., n$ instances and $k = 1, 2, .., m$ metrics
Output: A bug count vector $BCV[1:n]$
Performance of MTB as MAE, MRE and Pred(l)_Error
Algorithmic Steps:
Step 1: Select top p metrics out of m metrics using Algorithm 7 (Section 5.1.1)
Step 2: Perform threshold calculation of each software metric (Section 5.1.2)
for $k \leftarrow 1$ **to** p **do**
  Apply natural log transformation $ln(X_k[x_i]_{i=1}^n + 1)$ using (5.5)
  Apply mean function $mean(\mu_k)$ using (5.6)
  Apply standard deviation function $SD(\Omega_k)$ using (5.7)
  Find logarithmic threshold $T_k^{'} \leftarrow mean(\mu_k) + SD(\Omega_k)$ using (5.8)
  Apply exponential function $exp(T_k^{'})$ to find actual threshold of software metrics T_k or $TV[1:p]$ using (5.9)
end
Step 3: Now evaluate $BCV[1:n]$ based on $TV[1:p]$; $BCV[i]$ is the total number of metrics values higher than the corresponding metrics threshold in i^{th} module (Section 5.1.3)
for $i \leftarrow 1$ **to** n **do**
  $temp \leftarrow 0$
  for $k \leftarrow 1$ **to** m **do**
    if $X_k[x_i] > T_m[k]$ **then**
      $temp = temp + 1$
  end
  $BCV[i] \leftarrow temp$
end
Step 4: Evaluate the performance of MTB in terms of MAE, MRE, Pred(l)_Error.

5.1.4 Machine Learning Model Building

Initially, we have an unlabelled dataset $X_k[x_i]_{i=1}^n$ for $i = 1, 2, .., n$ instances and $k = 1, 2, .., m$ metrics. After applying the MTB approach, we have a dataset $X_k[x_i]_{i=1}^n$ of $i = 1, 2, .., n$ instances and $k = 1, 2, .., p$ metrics with a label bug count value ($BCV[1:n]$) for each instance. Now, we have used the robust linear regression (RLM) algorithm [228] with 10-fold cross-validation to train the model with $X_k[x_i]_{i=1}^n$ for $k = 1, 2, .., p$ and $BCV[1:n]$. Machine learning training is done using (5.10).

$$Train\_Model \leftarrow train_{RLM}(X_k[x_i]_{i=1}^n, BCV[1:n]) \tag{5.10}$$

In the above equation, $train_{RLM}$ represents a function that trains the model using the RLM algorithm on a provided input dataset $X_k[x_i]_{i=1}^n$ and its corresponding labels $BCV[1:n]$. The resulting model is saved as "$Train\_Model$" and subsequently used for predicting the bug count on the original datasets.

5.1.5 Software Bug Count Vector (SBCV) Prediction

After getting a trained machine learning model using linear regression ($Train\_Model$), we pass the original dataset with selected metrics $X_k[x_i]_{i=1}^n$ for $k = 1, 2, .., p$ as input to the trained model to predict the precise number of bugs in each instance of the original dataset. The trained model predicts the BCV using (5.11).

$$Predict\_BCV \leftarrow predict(Train\_Model, X_k[x_i]_{i=1}^n) \tag{5.11}$$

In the above equation, the function "$predict$" utilizes the "$Train\_Model$" and dataset X_k to predict the bug count in each module, which is then saved in $Predict\_BCV$. Subsequently, the three performance parameters are evaluated based on the predicted number of bug values and the actual number of bug values, as described in Subsection 5.2.3.

The main goal of this proposed approach MTBP with MTB is to utilize the unsupervised metric selection technique to find the effective software metrics and then use the threshold of software metrics to generate the bug count vector. After getting bug count vector using MTB, we build the machine learning model (MTBP) using $X_k[x_i]_{i=1}^n$ and BCV. Then, we predict the performance of MTBP model (machine learning model) on the original dataset. The pseudo-code of the proposed approach MTB/MTBP is written as Algorithm 9.

Algorithm 9: To implement MTBP (**M**etric selection **T**hreshold calculation, **B**CV generation and **P**lus) technique

Input: A dataset $X_k[x_i]_{i=1}^n$
Output: Bug count prediction for each module of dataset $X_k[x_i]_{i=1}^n$
Algorithmic Steps:
Step 1: Perform metrics selection using Algorithm 7
Step 2: Calculate software metrics threshold vector $TV[1:p]$ and generate bug count vector $BCV[1:n]$ using Algorithm 8
Step 3: Perform machine learning model building using robust linear regression (RLM) algorithm with 10-fold cross-validation (Section 5.1.4)
$Train\_Model \leftarrow train_{RLM}(X_k[x_i]_{i=1}^n, BCV[1:n])$
Step 4: Perform bug count vector prediction on an original dataset
$Predict\_BCV \leftarrow predict(Train\_Model, X_k[x_i]_{i=1}^n)$
Step 5: Evaluate the performance of MTBP as MAE, MRE, Pred(l)_Error.

5.2 Experimental Design

In this section, we provide in-depth details of the framed RQ in Section 5.2.1 and the benchmark datasets in Section 5.2.2. The performance measures and the statistical test performed are given in Section 5.2.3.

5.2.1 Research Questions

To guide our experimental analysis, we have formulated the following two research questions. Subsequently, we strive to address and find answers to these questions.

RQ1: Is the prediction performance of MTB/MTBP approach comparable to that of standard supervised machine learning models?
To answer RQ1, we have compared the SBCV prediction performance of MTB/MTBP with the eight Standard machine learning models trained using labeled datasets, which are 8 supervised machine learning regression models. These machine learning 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). By employing the caret package from the R library [203], we executed these algorithms (baseline approaches) using their default parameters. If MTB/MTBP's performance aligns with that of conventional supervised machine learning models, its viability for application in new software projects involving unlabeled datasets becomes evident.

RQ2: What is the effect of selected software metrics on the performance of MTB/MTBP approach and standard supervised machine learning models?
To answer this question, we have compared the SBCV prediction performance of MTB/MTBP and eight supervised regression models with varying numbers of selected metrics. The software metrics are selected using the unsupervised metric selection (UMS) approach (Subsection 5.1.1). These selected metrics are used in both MTB/MTBP as well as supervised learning approaches, and then a comparative analysis of their performance is made.

5.2.2 Datasets

Our objective is to predict the bug count value in each software module. Therefore, we specifically opt for datasets that include information about the number of bugs. The PROMISE repository (MORPH) is a widely utilized source of datasets for predicting software bug count value [71, 77, 150]. The repository comprises numerous software projects, some with varying versions. We have selected the datasets using the following criteria.

1. To ensure an adequate software project size, we have chosen datasets with a module count exceeding 300 [71, 77, 150].

2. Among the projects selected based on the aforementioned criteria, some exhibit varying numbers of versions. To maintain experimental control, we have specifically chosen only the last two versions of datasets from each project.

We have listed 22 datasets in Table 2.4 chosen from PROMISE (MORPH)[1]

[1]https://github.com/klainfo/DefectData

group [90,150,182] and Eclipse (AEEEM)[2] group [52,177,183]. The brief information about the datasets, like number of software metrics, number of modules, number of buggy modules, and bug% are given in Table 2.4 and description given in Section 2.4.2.1.

5.2.3 Performance Measures

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, we have selected the three performance parameters, viz. MAE, MRE, and Pred(l)_Error to compare the performance of MTB/MTBP and the baseline machine learning algorithms. The detail is provided in Section 2.4.3.2. We have also utilized the Wilcoxon signed rank test [191] to conduct a statistical comparison of the models, relying on the derived p-values. We have used Cohen's D test to find the effect size among all the models. Nemenyi test [192], followed by Friedman test [193] suggested by Demsar [194] is used to find the mean score of all the models over 22 datasets, and the mean scores are represented using critical diagrams. The details are provided in Section 2.4.4.

5.3 Experimental Results

In this section, we present the experimental outcomes of the proposed approach MTB/MTBP (unsupervised model) and eight standard algorithms (supervised models). These results are reported in terms of MAE, MRE, and Pred(l)_Error over 22 datasets to answer the RQs.

Table 5.2 presents the models' performance value, and Fig. 5.3 shows their comparative performance via boxplot. Additionally, a right-upper triangle of Table 5.3 presents the effect size (using Cohen's D test), and left bottom triangle of Table 5.3 presents p-value (using Wilcoxon signed-rank test) among all machine learning

[2]https://bug.inf.usi.ch/download.php

models, and Fig. 5.4 presents the critical diagram (using Nemenyi test) for all the models to answer RQ1. Fig. 5.5 shows the graph between the performance of the models with the selected number of metrics to answer RQ2.

5.3.1 Result Analysis for RQ1

RQ1: Is the prediction performance of MTB/MTBP approach comparable to that of standard supervised machine learning models?

The results in terms of MAE, MRE, and Pred(l)_Error of unsupervised model MTB/MTBP and 8 supervised machine learning models are shown in Table 5.2. In terms of average error MAE, SVR (0.30), and KNN (0.40) perform better than MTBP (0.45). Whereas in terms of MRE, and Pred(l)_Error, MTBP model (0.20, 0.29) surpasses all the other machine learning models except SVR (0.15, 0.17). Hence, the average performance of SVR is the best, followed by MTBP in terms of MRE and Pred(l)_Error. For performance in terms of average MAE, the decreasing order of the models is SVR (0.30), KNN (0.40), MTBP (0.45), LM (0.46), GA (0.46), BRR (0.46), DTR, (0.47) NBR (0.47), MLP (0.48) and MTB (0.56). Out of these, the average MAE performance of LM, GA, BRR are same (0.46), and average MAE performance of DTR and NBR are same (0.47). So, MLP and MTB are the poorest-performing models. Regarding average MRE and Pred(l)_Error performance, the top-performing models are SVR, MTBP, MLP, KNN, and GA.

The boxplot representations of the proposed unsupervised model MTB/MTBP and 8 supervised machine learning models are shown in Fig. 5.3. From Fig. 5.3, it can be concluded that the median performance of MTBP (0.34) and MLP (0.34) in terms of MAE is better than all models except SVR (0.25). The distribution of MAE value over 22 datasets is also compact as compared to all other models except SVR. The median performance of MTBP in terms of MRE (0.17) is better than all models except SVR (0.13) and MLP (0.13). The performance of MTBP in terms of median Pred(l)_Error (0.24) stands at the fourth position. Further, we present

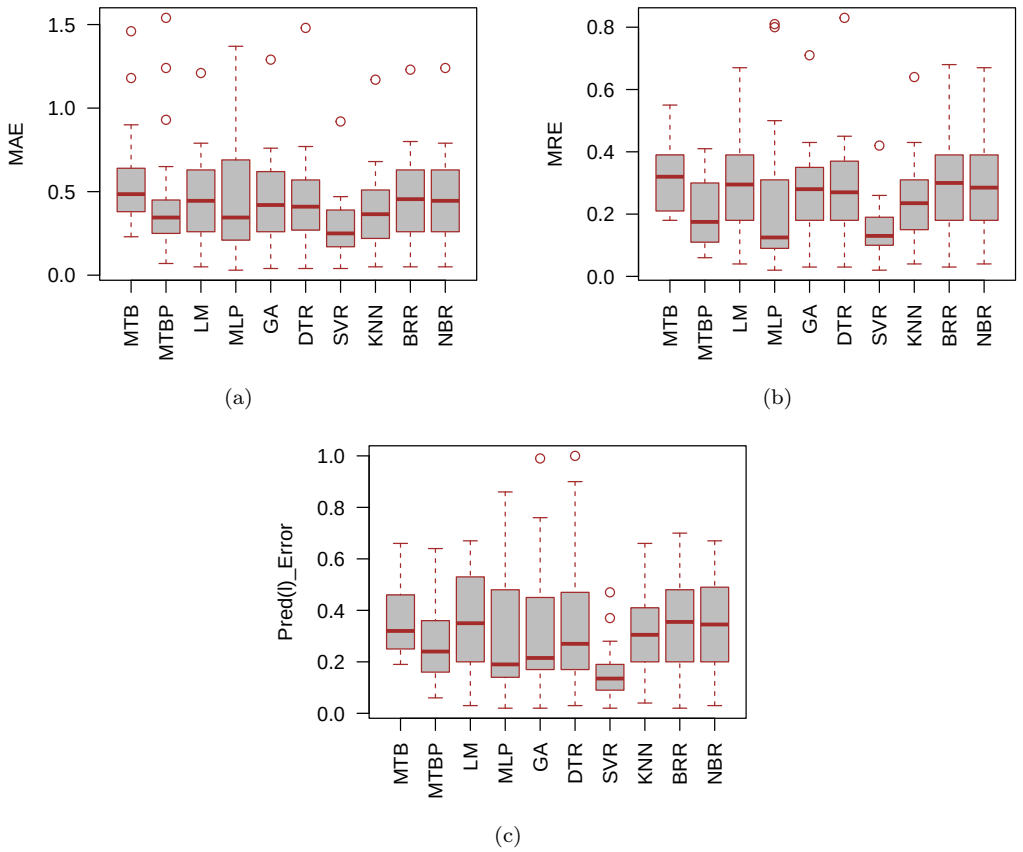


FIGURE 5.3: Boxplot performance of MTB/MTBP and baseline models in terms of (a) MAE, (b) MRE, and (c) Pred(l)_Error

results based on statistical tests (Wilcoxon signed-rank test, Cohen's D test, and Nemeny test) to find the significance of the proposed model compared to baseline models.

The effect size (shown in a right-upper triangle) using Cohen's D test and p-value (shown in a left-bottom triangle) using Wilcoxon signed-rank test among all the models are shown in Table 5.3. Based on the p-value ($>= 0.05$), the performance of MTBP is significantly equivalent to LM (0.30), MLP (0.59), GA (0.53), DTR (0.26), KNN (0.63), BRR (0.27), and NBR (0.22) in terms of MAE. Also, the performance of MTBP is significantly different from SVR (0.00) only in terms of MAE. Based on effect size, the standardized mean difference of MTBP is negligible ($\Phi < 0.2$) as compared to LM (0.02), MLP (0.09), GA (0.00), DTR (0.05), KNN (0.16), BRR

(0.03), and NBR (0.04). So, SVR (0.55) reports a medium ($\Phi < 0.8$) effect size with MTBP. Similar observation can be seen in terms of MRE and Pred(l)_Error. Finally, the comparative results of all the models across 22 datasets are visualized using the critical diagram.

The critical diagram generated using the Nemenyi test is shown in Fig. 5.4. The p-value of 0.000 using the Friedman test shows that there are significant differences among the models. The critical distance of 2.888 was calculated using Nemenyi test, which was used to divide the models among a low-performing group and a high-performing group. The performance of the two approaches is considered significantly different if their mean scores differ by more than the calculated critical distance (CD).

TABLE 5.2: Results of MTB/MTBP and 8 machine learning models in terms of MAE, MRE, and Pred(l)_Error

Groups	Datasets	Mean Absolute Error (MAE)										Mean Relative Error (MRE)										Prediction at level l=0.3 error (Pred(l)_Error)									
		MTB	MTBP	LM	MLP	GA	DTR	SVR	KNN	BRR	NBR	MTB	MTBP	LM	MLP	GA	DTR	SVR	KNN	BRR	NBR	MTB	MTBP	LM	MLP	GA	DTR	SVR	KNN	BRR	NBR
MORPH	Ant-1.6	0.49	0.41	0.46	0.50	0.47	0.47	0.39	0.37	0.47	0.53	0.30	0.18	0.30	0.16	0.30	0.30	0.20	0.23	0.32	0.31	0.33	0.27	0.33	0.26	0.22	0.22	0.25	0.29	0.38	0.35
	Ant-1.7	0.39	0.36	0.46	0.97	0.45	0.48	0.34	0.41	0.47	0.46	0.20	0.14	0.30	0.81	0.29	0.31	0.18	0.26	0.31	0.28	0.24	0.23	0.36	0.86	0.21	0.20	0.19	0.31	0.36	0.31
	Camel-1.4	0.42	0.35	0.54	0.34	0.53	0.57	0.39	0.49	0.55	0.52	0.21	0.11	0.36	0.10	0.34	0.37	0.17	0.31	0.36	0.33	0.27	0.17	0.56	0.17	0.20	1.00	0.16	0.46	0.57	0.49
	Camel-1.6	0.48	0.41	0.63	0.57	0.64	0.66	0.45	0.57	0.63	0.63	0.22	0.13	0.41	0.31	0.41	0.42	0.19	0.35	0.40	0.41	0.29	0.20	0.64	0.48	0.99	0.90	0.19	0.49	0.64	0.64
	Ivy-2.0	0.38	0.21	0.22	0.23	0.25	0.23	0.12	0.22	0.22	0.23	0.33	0.16	0.15	0.07	0.16	0.16	0.07	0.12	0.15	0.15	0.25	0.16	0.20	0.17	0.17	0.17	0.07	0.18	0.20	0.20
	Jedit-4.2	0.33	0.23	0.33	0.26	0.35	0.29	0.24	0.29	0.33	0.33	0.21	0.09	0.23	0.09	0.19	0.19	0.12	0.18	0.23	0.23	0.22	0.14	0.26	0.15	0.15	0.28	0.12	0.28	0.26	0.26
	Jedit-4.3	0.24	0.07	0.05	0.03	0.04	0.04	0.04	0.05	0.05	0.05	0.23	0.06	0.04	0.02	0.03	0.03	0.02	0.04	0.03	0.04	0.19	0.06	0.03	0.02	0.02	0.03	0.02	0.04	0.02	0.03
	Lucene-2.4	1.46	1.54	1.21	1.37	1.29	1.48	0.92	1.17	1.23	1.24	0.39	0.40	0.67	0.80	0.71	0.83	0.42	0.64	0.68	0.67	0.60	0.60	0.67	0.74	0.76	0.66	0.47	0.66	0.70	0.67
	Poi-2.5	1.18	1.24	0.79	1.24	0.76	0.71	0.36	0.68	0.80	0.79	0.41	0.41	0.48	0.41	0.35	0.43	0.19	0.43	0.49	0.48	0.66	0.64	0.42	0.64	0.52	0.54	0.18	0.39	0.44	0.42
	Poi-3.0	0.90	0.93	0.66	0.91	0.67	0.77	0.47	0.64	0.69	0.66	0.34	0.36	0.39	0.38	0.38	0.44	0.20	0.36	0.39	0.39	0.61	0.64	0.47	0.64	0.45	0.45	0.25	0.41	0.48	0.47
	Prop-4	0.45	0.25	0.24	0.15	0.23	0.24	0.17	0.21	0.24	0.24	0.37	0.18	0.17	0.06	0.15	0.16	0.10	0.14	0.17	0.17	0.25	0.18	0.18	0.10	0.10	0.12	0.09	0.18	0.18	0.18
	Prop-5	0.29	0.20	0.35	0.23	0.35	0.35	0.25	0.31	0.35	0.35	0.18	0.09	0.24	0.09	0.24	0.24	0.13	0.21	0.24	0.24	0.23	0.16	0.32	0.17	0.40	0.24	0.15	0.30	0.32	0.32
	Tomcat	0.23	0.14	0.20	0.16	0.23	0.21	0.13	0.18	0.20	0.20	0.19	0.07	0.14	0.06	0.13	0.14	0.07	0.12	0.14	0.14	0.20	0.11	0.17	0.13	0.12	0.12	0.08	0.16	0.17	0.17
	Xalan-2.5	0.64	0.65	0.66	0.64	0.63	0.66	0.46	0.55	0.66	0.66	0.37	0.31	0.45	0.50	0.43	0.45	0.26	0.37	0.45	0.45	0.50	0.52	0.66	0.58	0.75	0.63	0.37	0.53	0.63	0.62
	Xalan-2.6	0.73	0.64	0.66	0.70	0.62	0.57	0.42	0.45	0.66	0.66	0.42	0.30	0.43	0.26	0.40	0.37	0.22	0.28	0.44	0.43	0.54	0.47	0.65	0.46	0.45	0.47	0.28	0.32	0.69	0.63
	Xerces-1.2	0.36	0.26	0.43	0.38	0.41	0.40	0.28	0.36	0.44	0.43	0.21	0.10	0.29	0.24	0.28	0.27	0.14	0.24	0.30	0.29	0.28	0.16	0.53	0.22	0.20	0.24	0.16	0.33	0.16	0.53
	Xerces-1.3	0.48	0.34	0.48	0.33	0.35	0.42	0.25	0.39	0.47	0.48	0.29	0.14	0.31	0.10	0.23	0.27	0.12	0.24	0.30	0.31	0.31	0.19	0.39	0.15	0.23	0.32	0.10	0.31	0.39	0.39
AEEEM	Equinox	0.54	0.45	0.53	0.69	0.50	0.52	0.20	0.51	0.54	0.55	0.31	0.21	0.33	0.23	0.32	0.33	0.13	0.30	0.33	0.33	0.40	0.36	0.44	0.40	0.37	0.37	0.12	0.43	0.44	0.46
	JDT	0.51	0.31	0.40	0.35	0.43	0.39	0.20	0.36	0.40	0.40	0.39	0.17	0.26	0.12	0.28	0.25	0.11	0.22	0.26	0.26	0.46	0.26	0.34	0.21	0.18	0.26	0.12	0.28	0.35	0.34
	Lucene	0.59	0.32	0.23	0.21	0.23	0.23	0.12	0.20	0.23	0.23	0.54	0.26	0.16	0.13	0.16	0.16	0.07	0.13	0.16	0.16	0.45	0.30	0.17	0.09	0.10	0.10	0.06	0.16	0.17	0.17
	Mylyn	0.50	0.26	0.26	0.20	0.26	0.27	0.15	0.22	0.26	0.26	0.43	0.18	0.18	0.10	0.18	0.18	0.08	0.15	0.18	0.18	0.41	0.25	0.20	0.13	0.36	0.35	0.09	0.20	0.20	0.20
	PDE	0.64	0.42	0.33	0.21	0.30	0.36	0.20	0.28	0.33	0.34	0.55	0.32	0.23	0.08	0.21	0.25	0.11	0.18	0.23	0.23	0.41	0.32	0.26	0.14	0.18	0.14	0.11	0.23	0.26	0.20
	Average	0.56	0.45	0.46	0.48	0.46	0.47	0.30	0.40	0.46	0.47	0.32	0.20	0.30	0.23	0.28	0.30	0.15	0.25	0.30	0.29	0.37	0.29	0.38	0.31	0.33	0.36	0.17	0.32	0.36	0.37

TABLE 5.3: Effect size using Cohen's D (shown in Right Upper Triangle) and p-value using Wilcoxon signed-rank test (shown in left-bottom triangle) among all the models

	MAE										MRE										Pred(l)_Error									
	MTB	MTBP	LM	MLP	GA	DTR	SVR	KNN	BRR	NBR	MTB	MTBP	LM	MLP	GA	DTR	SVR	KNN	BRR	NBR	MTB	MTBP	LM	MLP	GA	DTR	SVR	KNN	BRR	NBR
MTB	-	0.31	0.35	0.21	0.37	0.29	1.04	0.56	0.33	0.33	-	1.14	0.21	0.51	0.33	0.18	1.76	0.60	0.19	0.22	-	0.48	0.04	0.27	0.22	0.06	1.60	0.37	0.02	0.01
MTBP	0.00	-	0.02	0.09	0.00	0.05	0.55	0.16	0.03	0.04	0.00	-	0.78	0.19	0.65	0.72	0.50	0.43	0.78	0.77	0.00	-	0.47	0.11	0.16	0.30	0.87	0.16	0.40	0.42
LM	0.02	0.30	-	0.08	0.02	0.03	0.73	0.23	0.02	0.02	0.79	0.00	-	0.34	0.11	0.01	1.26	0.34	0.01	0.01	0.90	0.03	-	0.28	0.23	0.09	1.39	0.36	0.06	0.05
MLP	0.04	0.59	0.60	-	0.10	0.05	0.65	0.26	0.06	0.06	0.04	0.89	0.00	-	0.25	0.33	0.49	0.09	0.35	0.33	0.05	0.86	0.03	-	0.04	0.17	0.80	0.01	0.23	0.24
GA	0.01	0.53	0.34	0.72	-	0.05	0.69	0.20	0.04	0.04	0.35	0.00	0.03	0.01	-	0.11	1.12	0.22	0.12	0.10	0.08	0.92	0.12	0.71	-	0.13	0.83	0.04	0.18	0.19
DTR	0.03	0.26	0.91	0.51	0.31	-	0.70	0.24	0.02	0.01	0.63	0.00	0.77	0.00	0.03	-	1.14	0.32	0.00	0.02	0.10	0.46	0.18	0.33	0.65	-	0.97	0.19	0.04	0.04
SVR	0.00	0.00	0.00	0.00	0.00	0.00	-	0.51	0.74	0.75	0.00	0.02	0.00	0.06	0.00	0.00	-	0.91	1.26	1.25	0.00	0.00	0.00	0.00	0.00	0.00	-	1.19	1.28	1.35
KNN	0.00	0.63	0.00	0.26	0.00	0.00	0.00	-	0.24	0.25	0.08	0.03	0.00	0.08	0.00	0.00	0.00	-	0.35	0.33	0.12	0.34	0.00	0.33	0.88	0.72	0.00	-	0.29	0.31
BRR	0.02	0.27	0.04	0.60	0.16	0.83	0.00	0.00	-	0.00	0.87	0.00	0.10	0.00	0.01	0.49	0.00	0.00	-	0.03	0.75	0.05	0.48	0.07	0.18	0.31	0.00	0.00	-	0.01
NBR	0.03	0.22	0.17	0.51	0.25	0.70	0.00	0.00	0.82	-	0.86	0.00	0.42	0.00	0.05	0.90	0.00	0.00	0.05	-	1.00	0.07	0.09	0.04	0.16	0.23	0.00	0.00	0.09	-

The blue stick with a red dot shows the high-performing group, and the blue stick with the black dot presents the low-performing group. The models are arranged in decreasing order of their performance. This means the model with the minimum mean score is the best model among all the models. From Fig. 5.4, it can be concluded that SVR is the best regression model as compared to all the models in terms of MAE mean score (1.50), MRE mean score (1.84), and Pred(l)_Error (1.20). MTBP belongs to the high-performing group in terms of MRE mean score (3.93) but shows lower performance than SVR and MLP. In terms of Pred(l)_Error, all models belong to the low-performing group except SVR, but MTBP performs better than all models except SVR and MLP. Overall, considering the mean scores of MAE, MRE, and Pred(l)_Error (4.75, 3.93, 4.68), MTBP holds the third position among MTB and eight other machine learning models. However, there is room for improvement in the Pred(l)_Error performance of the MTBP model.

5.3.2 Result Analysis for RQ2

RQ2: What is the effect of selected software metrics on the performance of MTB/MTBP approach and standard supervised machine learning models?

We implemented MTB/MTBP and baseline models with different numbers of software metrics (5, 6, 7, 9, 11, 14, 17, 20). The performance of models corresponding to different numbers of metrics is depicted in Fig. 5.5. Notably, the MTBP model using five selected metrics outperforms all regression models except SVR. However, as we increase the number of selected metrics beyond 5, the performance of MTBP starts to decline. When number of selected metrics is 7, the performance of MTBP becomes comparable to the baseline regression models in terms of MAE, MRE, and Pred(l)_Error. We have observed that most datasets contain a maximum of 7 bugs. Hence, we can deduce that MTBP is adequate to achieve performance levels comparable to existing standard machine learning models. Additionally, the UMS method plays a crucial role in selecting effective metrics. However, it is worth noting that MTBP with UMS may not accurately predict bug counts greater than seven.

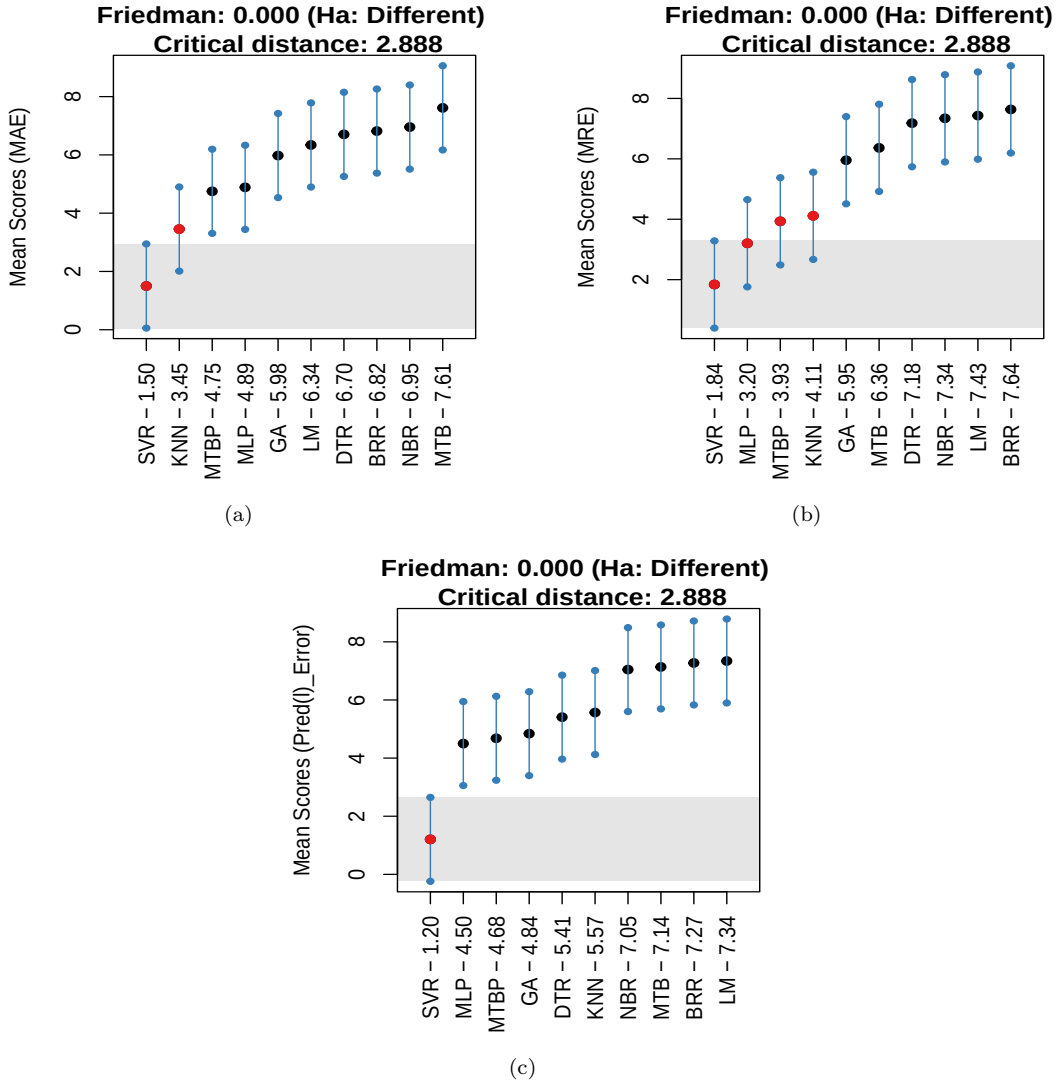
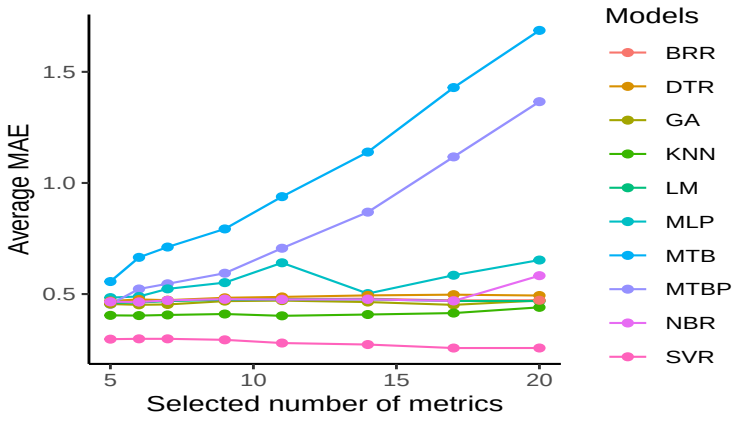
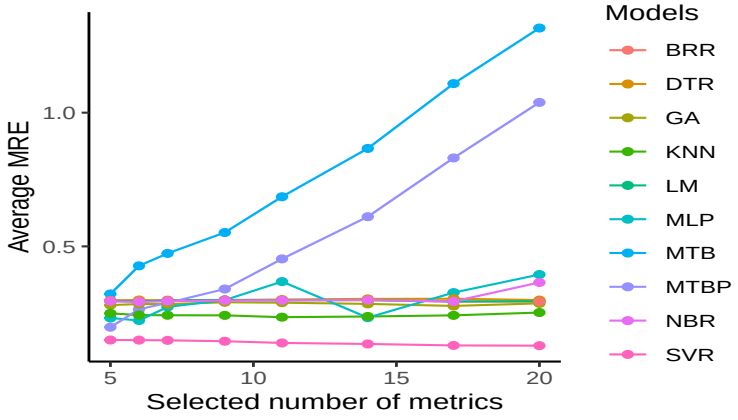


FIGURE 5.4: Performance analysis of MTB/MTBP and baseline models using Post Hoc Nemenyi Test in terms of mean score (a) MAE, (b) MRE, and (c) Pred(l)_Error

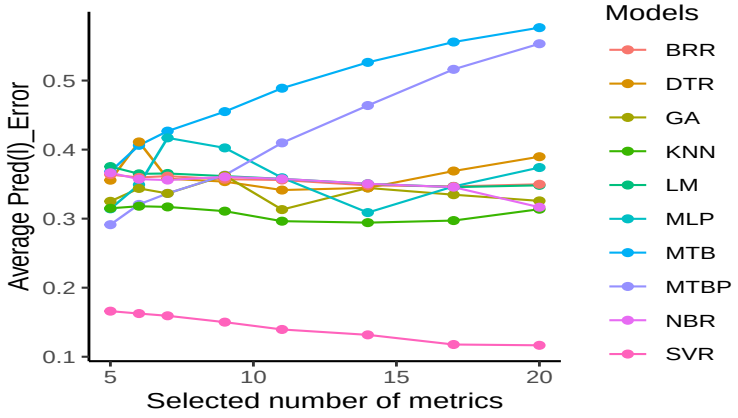
Our empirical observation finds that one metric causes one bug, which is true only for up to five metrics. After that, it would not be true (Fig. 5.5). In the future, further analysis can be conducted to address and overcome this limitation.



(a)



(b)



(c)

FIGURE 5.5: Average performance analysis of MTB/MTBP and other baselines with the number of selected metrics conducted on 22 datasets in terms of (a) MAE, (b) MRE, and (c) Pred(l)_Error

5.4 Explainability and Interpretability of SBCV Prediction Models

In this section, we have added some interpretations of the experimental results and some implications for practitioners and researchers.

5.4.1 Software Metrics Selected

The list of 20 software metrics of MORPH group and 17 software metrics of AEEEM group are shown in Table 5.4. The details about software metrics are given in Section 2.4.1.

Gao et al. [220] suggested that minimum $\lceil log_2 m \rceil$ metrics are sufficient to build the software bug prediction models. So, Initially, we have selected a minimum of five ($\lceil log_2 20 \rceil$) for m=20 software metrics. Then, we performed an experimental analysis by selecting a different number of metrics group-wise. Based on experimental observation, we found that the proposed approach performs better when selecting minimum $\lceil log_2 m \rceil$ metrics and declines beyond $\lceil log_2 m \rceil$, as shown in Fig. 5.5.

Table 5.4 also presents the top selected metrics using UMS from each dataset to build the SBCV prediction models. We found that cam, lcom3, mfa, rfc, and ce are the most common software metrics for datasets of MORPH group. Similarly, fanOut, nomi, and nopra are the most common software metrics for datasets of AEEEM group. Hence, these common software metrics are effective for building the SBCV prediction models.

5.4.2 Performance Analysis of MTB/MTBP and Baseline Models with respect to Bug %

To analyze the performance of MTB/MTBP and other baseline models concerning bug percentage (2.24%-64.40%), we have constructed line charts depicting the relationship between bug percentage and performance measures, as shown in

TABLE 5.4: Top selected metrics from each dataset after UMS and name of all metrics used group-wise

Group	Datasets	Top 5 selected metrics	MORPH Metrics	AEEEM Metrics Used
MORPH	Ant-1.6	cam, rfc, lcom3, wmc, dit	wmc	cbo
	Ant-1.7	lcom3, cam, rfc, ce, mfa	dit	dit
	Camel-1.4	mfa, lcom3, cam,ce, avg_cc	noc	fanIn
	Camel-1.6	mfa, lcom3, cam, ce, avg_cc	cbo	fanOut
	Ivy-2.0	cam, rfc, lcom3, ce, max_cc	rfc	lcom
	Jedit-4.2	cam, lcom3, ce, rfc, dam	lcom	noc
	Jedit-4.3	cam, rfc, lcom3, moa, mfa	ca	numberOfAttributes (noa)
	Lucene-2.4	mfa, lcom3, cam, ce, .dam	ce	numberOfAttributesInherited (noai)
	Poi-2.5	cam, lcom3, dam, mfa, rfc	npm	numberOfLinesOfCode (noloc)
	Poi-3.0	lcom3, cam, rfc, mfa, dam	lcom3	numberOfMethods (nom)
	Prop-4	amc, rfc, ce, mfa, cbo	loc	numberOfMethodsInherited (nomi)
	Prop-5	cam, ce, avg_cc, lcom3, mfa	dam	numberOfPrivateAttributes (nopra)
	Tomcat	cam, lcom3, avg_cc, rfc, mfa	moa	numberOfPrivateMethods (noprm)
	Xalan-2.5	lcom3, cam, ce, cbm, amc	mfa	numberOfPublicAttributes (nopua)
	Xalan-2.6	lcom3, cam, ce, cbm, rfc	cam	numberOfPublicMethods (nopum)
	Xereces-1.2	mfa, cam, npm, lcom3, dam	ic	rfc
	Xereces-1.3	lcom3, cam, npm, cbo, dam	cbm	wmc
AEEEM	Equinox	noloc, cbo, fanOut, nomi, nom	amc	
	JDT	nomi, cbo, nopua, fanOut, nopra	max_cc	
	Lucene	wmc, nomi, fanOut, nopum, nopra	avg_cc	
	Mylyn	noa, noprm, nomi, fanOut, nopum		
	PDE	wmc, nomi, fanOut, nom, nopra		

Figs. 5.6a, b, and c. Our findings reveal that the error of SBCV prediction models increases as the bug percentage in software project datasets increases. Up to 10% increase in bug percentage, all baseline models exhibit a linear increase in error. While beyond that point, the models' prediction errors become non-linear. Interestingly, the MTB/MTBP model also follows a similar pattern. Hence, we can conclude that the performance behavior of MTB/MTBP is analogous to the existing baseline models with respect to varying bug %.

The software projects Lucene-2.4 (59.70%), Poi-2.5 (64.40%), Poi-3.0 (63.57%), and Xalan-2.5 (48.19%) exhibit the highest bug percentages. Our analysis indicates that, on these datasets, MTB performs better than MTBP in terms of MAE. However, when compared to other baseline models, the performance of MTB/MTBP is relatively poor on these datasets in terms of MAE. As a result, we can infer that MTB/MTBP is likely to be more effective on software projects with bug percentages

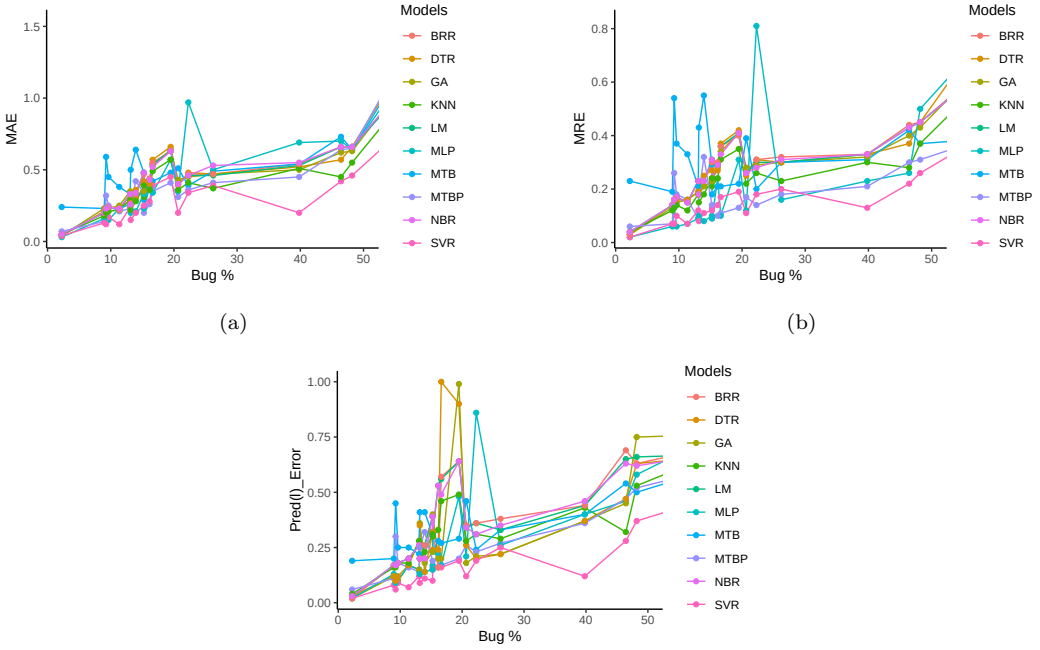


FIGURE 5.6: Performance analysis of MTB/MTBP and baseline models with respect to bug % in 22 datasets in terms of (a) MAE, (b) MRE, and (c) Pred(l)_Error

lower than 48%. If the software is developed in a good and controlled environment, then such software is likely have fewer bugs. Also, if the initial phases of Software Development Life Cycle have been explored thoroughly with proper documentation then coding is likely to be bug free [229]. So, MTB/MTBP can be applied to such software. Software developed by naïve programmers will likely have more bugs, and MTB/MTBP will not be helpful

5.4.3 Performance Analysis of MTB/MTBP and Baseline Models with respect to Software Projects Size

To analyze the performance of MTB/MTBP and other baseline models with respect to software project size (351-8718 modules), we have constructed line charts depicting the relationship between project size and performance measures, as shown in Figs. 5.7a, b, and c. Our findings reveal that most of the software projects used have less than 1,000 modules. The figures show that the performance of the

SBCV models is not dependent on the size of the software projects. All baseline models exhibit an increase/decrease in error versus the size of the software projects. Interestingly, the MTB/MTBP model also follows a similar pattern. Hence, we can conclude that the performance behavior of MTB/MTBP is analogous to the existing baseline models with respect to varying software project sizes.

The software projects Prop-4 (8718 modules) and Prop-5 (8516 modules) are the largest among those used in the study. Interestingly, the performance of MTB/MTBP on Prop-4 is the worst as compared to all the baseline models, while on Prop-5, it performs the best as compared to all baseline models. This indicates that the performance of MTB/MTBP is comparable to other standard supervised ML techniques on small size projects and better on one of the large size project (Prop-5). We have implemented MTB/MTBP on three larger-size real projects, Eclipse 2.0, 2.1, and 3.0, and we find that MTB/MTBP performs better than standard ML algorithms in terms of MAE and MRE (Table B.5).

5.4.4 Usage Scenario and Limitations of MTB/MTBP

MTB/MTBP reported better or equivalent performance as compared to the baseline models when considering only those modules where the maximum number of bugs is seven (Fig. 5.5). MTBP is anticipated to be more effective for software projects with bug percentages below 48% (Table 5.2). On the other hand, when bug % exceeds 48%, MTB outperforms MTBP. Therefore, it becomes intriguing to investigate the specific contexts or situations in which it would be preferable to use MTB over MTBP, and vice versa. We have left this exploration for future work. MTB/MTBP can be used with any size of the software project. This means software project size does not affect the performance of MTBP. SVR algorithm with radial basis function (RBF) kernel is the best supervised SBCV prediction model amongst those considered. A limitation of MTB/MTBP is that its bug prediction capability is constrained by the maximum number of software metrics selected for the prediction.

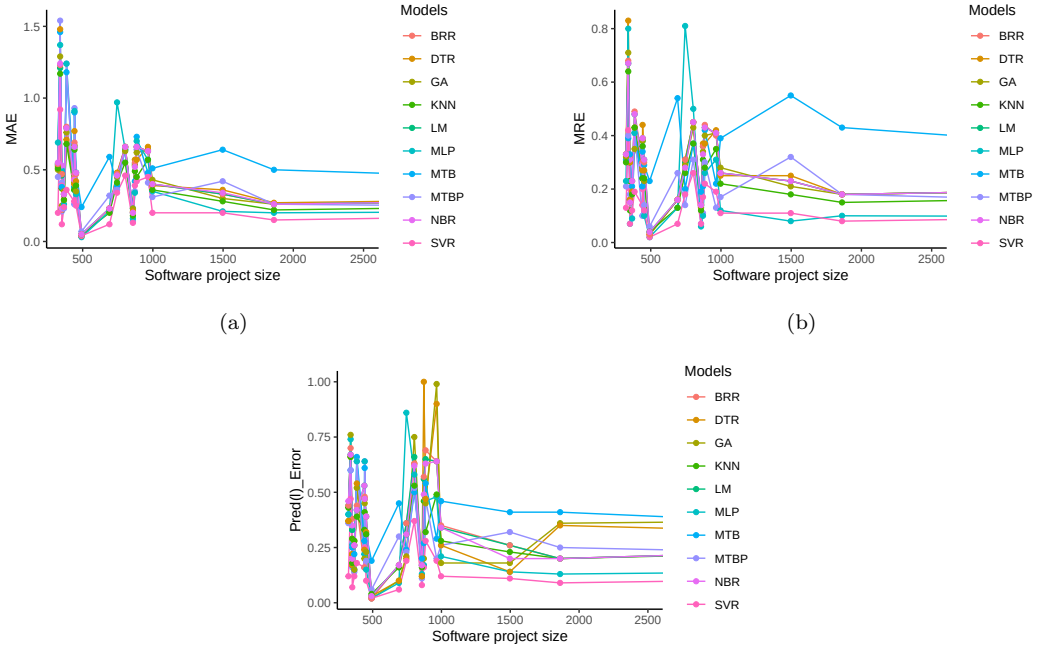


FIGURE 5.7: Performance analysis of MTB/MTBP and baseline models with respect to software projects size in terms of (a) MAE, (b) MRE, and (c) Pred(l)_Error

If we select a maximum number of metrics p, then we can only predict a maximum number of p bugs using MTBP.

5.5 Threat to Validity

The proposed approach is implemented with full consideration. The following risks may be associated with the proposed work.

Internal validity: The proposed MTB/MTBP and standard algorithms are validated using the 16 software bug project datasets collected from the two groups viz. MORPH and AEEEM. The datasets are consistently collected from public repositories; however, it cannot be definitively claimed that these datasets are completely accurate.

Construct validity: The MTB/MTBP predicts bug count vectors based on selected metrics. The performance of the MTBP model is greater as compared to the standard machine learning models when the maximum number of bugs in the datasets is 7. This is a potential threat to the MTBP model. All the standard algorithms are implemented with the default parameters of the standard library of R programming.

External validity: The proposed MTBP model was validated on only class/file level granularity. The MTBP may be useful for other types of granularity as well like methods, and functions. We have used only three performance parameters to validate the performance of MTBP.

5.6 Conclusion and Future Work

Overall, the proposed approach MTB/MTBP demonstrates superior or equivalent performance to standard machine learning models, except for SVR, in terms of average MAE, MRE, and Pred(l)_Error across all datasets (Table 5.2). The effect size analysis indicates that the standardized mean difference of MTBP is negligible ($\Phi < 0.2$) compared to the baseline models, and the p-value analysis shows that MTBP significantly outperforms most of the baseline models (Table 5.3).

When comparing the performance using boxplots (Fig. 5.3) over the 22 datasets, MTBP is either comparable or better than the majority of supervised regression models. In terms of mean MAE, MTBP ranks third, trailing behind SVR and KNN. With respect to the mean MRE score, MTBP ranks third and belongs to the high-performing group, along with SVR, MLP, and KNN. In terms of Pred(l)_Error mean score, MTBP falls into the low-performing group, but it still outperforms GA, DTR, KNN, NBR, MTB, BRR, and LM.

The performance of MTB/MTBP is highly influenced by the number of selected

software metrics using the unsupervised metric selection (USM) technique. By selecting up to a maximum of 7 software metrics, we achieve superior or equivalent performance compared to supervised regression models in terms of MAE, MRE, and Pred(0.3)_Error for accurately predicting up to 7 bugs (Fig. 5.5).

In conclusion, the proposed MTB/MTBP is a unique technique, and its performance comparable to existing machine learning methods. Future work will focus on addressing the limitations of MTB/MTBP (see Section 5.4.4).