Chapter 4

BCV-Predictor: Cross-Version Defect Count Vector Predictor

4.1 Introduction

The bug count vector (BCV) is a column vector that consists of the information about bug number in every module, and the BCV-Predictor is the proposed method to predict such column vector (as discussed in section 1). In this chapter, we have formalized a problem statement and its novel solution. We anticipate the bugs count vector of the next version of a software system using information about all its previous versions. We have collected the information about software metrics used, bug details of each module, several versions of software and created a meta-information repository by concatenating different versions of the same software called meta-dataset. Meta-dataset feeds into the training process of the proposed approach. Early detection of the number of bugs in the modules can be beneficial in terms of allocating testing resources optimally and efficiently for each module. This chapter aims to build an approach called BCV-Predictor that can predict the number of bugs in each module of the upcoming software system version, which helps in reducing the testing effort. We have conducted our experiments over seven open-source software projects collected from the PROMISE data repository [215], and data consist of 20 various software metrics, one additional number of bugs information in every module. Accuracy, mean squared error, mean absolute error are used as performance evaluation metrics. Softwares nowadays are becoming more

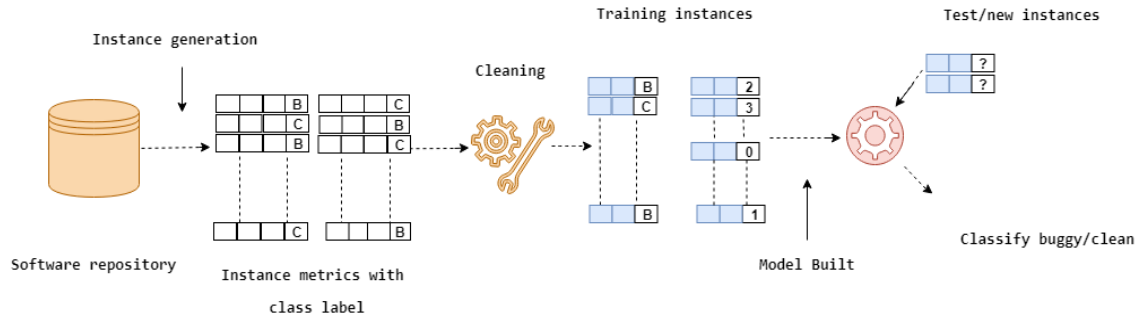


FIGURE 4.1: Basic architecture of software bug count prediction model.

sophisticated and bulky. The state-of-the-art methods are mostly based on classical machine learning (ML) based techniques. Large scale software system consists of an extensive number of software modules such as prop from the PROMISE repository. Traditional ML-based defect count prediction models are mostly inadequate on such projects and take huge training time. Deep learning (DL) architecture-based technique can conquer such challenges. To the extent of our knowledge, deep learning approaches for bug count prediction have not been explored. Moreover, the use of DL-based methods for the cross-version defect prediction [286] for the next version of software modules remained unexplored. Earlier, DL-based approaches were applied to software bug prediction; they were used to classify dependent variables into a buggy and clean classes. However, the use of DL-based methods for the regression problem in BCV prediction is never investigated. The regression problem is used to identify the density or quantity of the dependent variable, in our case number of bugs in each module. Whereas classifying buggy or non-buggy modules comes into a classification problem. Knowing the approximate number of bugs in the future version of the software will be utilitarian than knowing just a module is buggy or not [155]. It helps the software developer to prioritize the developing resources based on the number of bugs information. It also helps them to locate as many bugs as early and quickly. However, very limited work has been done in such aspects. We have framed three research queries (RQ) to justify the performance of the BCV-Predictor; we will address these RQs in section 4.3.7. The list of RQ is given below.
RQ-1: How much the BCV-Predictor is effective over all seven metadata?
RQ-2: Comparison of evaluation matrices of BCV-Predictor with baseline methodologies.
RQ-3: Training time comparison of the BCV-Predictor with baseline techniques.
The further structure of the chapter is in the following manner. In the next section,

we will explain problem statement. We will illustrate the proposed approach in section 4.3; further, we discuss results in section 4.3.7. Fig. 4.1 shows the underlying architecture of the software bug counts prediction model. Fig. 4.1 indicates the instances are generated from the software repository; then instances were labeled as "buggy" or "clean" according to their status. Data cleaning needs to be done to remove noisy instances and needs to extract relevant features. After that, the instances that have buggy labels replace a number of bugs, and the clean label replaces with 0. Then data is ready to train the machine learning [155], or statistical learning [71] based predictive model; the new instance with unknown labels is used to test the predictive model.

4.2 Problem statement

Our empirical study investigates to predict the bug count vector in the upcoming version of the software system. The set of software features (number of features) is the same across all versions of a software. The problem formulation is shown below.

- Let the software system S has n versions, where n is a positive integer.
- Let f is the number of features in each version, f is fixed for all versions of the same software system.
- Every version has a different number of modules, and buggy modules consist of a finite number of bugs.
- We will predict the bug count vector that consists of the number of bugs in each module of $(n+1)^{th}$ version of S.

$$S_i = (M_1, M_2, M_3........, M_j) \tag{4.1}$$

$$S_{n+1} = (M_1, M_2, M_3........, M_k) \tag{4.2}$$

For each module M_k of S_{n+1} no. of features $\in$ (1, 2, 3, ..., f). We consider it as a regression problem. Let maximum bug count in metadata of S is $\max(bug_S)$. The proposed approach will predict the bug count vector. The predicted BCV consists of bug value in each module of S_{n+1} from 0 to $\max(bug_S)$, and predicting it minimizes the testing effort.

Table 4.1: Meta-datasets description.

Project	No. of version	No. of modules	No. of buggy modules	% of buggy modules	Max no. of bugs (max(bug_S))
ant	5	1692	637	37.76%	10
camel	4	2784	1371	49.24%	28
jedit	4	1257	639	50.83%	45
prop	6	69554	8540	12.27%	37
xerces	4	1644	815	49.57%	62
xalan	4	3320	2210	66.65%	9
poi	4	1378	705	51.16%	20

4.2.1 Dataset description

We have used seven open-source projects from the PROMISE data repository [215] as shown in Table 2.4. Table 2.4 also reports that some of the data are imbalanced, i.e., highly skewed towards non -buggy instances. Wang et al. [266] suggest sampling techniques on defects datasets to avoid imbalance issues. We have used random over-sampling [3] techniques to circumvent this problem. We have built metadata using a different version of the software system. Table 4.1 shows the metadata description of datasets. As the table reports, ant and prop have 37.76% and 12.27% of buggy modules, respectively. In all meta-datasets, few instances have some number of bugs values, let's say x, and that x is very few in comparison of all instances, which causes an imbalance distribution of class x.

4.2.2 Software metrics

Software metrics are designed based on the code complexity and features of the object-oriented program. Table 2.2 includes the metrics related to all seven software systems. Software systems have five different metrics classes: Abstraction, Cohesion, Coupling, Complexity, and Encapsulation. A total of nineteen software metrics, all these metrics have been considered for experiments.

4.2.3 Meta-dataset collection process

Data collection is the procedure of assembling and measuring information on variables of interest. We make use of seven public projects and their various versions, as discussed in section 4.2.1. The overall collection of the metadata process is shown in Fig.4.2. Metadata is an aggregated emergence of all prior versions of the software.

We are illustrating this process using an example. The project ant has five different versions from version 1.3 to 1.7, as shown in Table 4.1. Every version of ant software has 20 software metrics (X) from f_1, f_2 to f_{20}, and one bug count vector (Y) as shown in Fig. 4.2.

$$Meta(ant_X) = ant(X_{1.3}) * ant(X_{1.4}) * .... * ant(X_{1.7}) \tag{4.3}$$

$$Meta(ant_Y) = ant(Y_{1.3}) * ant(Y_{1.4}) * .... * ant(Y_{1.7}) \tag{4.4}$$

$$Meta(ant) = Meta(ant_X) + Meta(ant_Y) \tag{4.5}$$

Ant($X_{1.3}$), ant($X_{1.4}$), ..., ant($X_{1.7}$) are various feature matrices of all five versions of the ant project, Whereas $bug_{1.3}$, $bug_{1.4}$, ..., $bug_{1.7}$ are bug count vector of these five versions, respectively. The feature matrix and bug count vector of metadata ant project represented as Meta(ant_X) and Meta(ant_Y) respectively as shown in Eq.4.3 & Eq.4.4. We concatenate various feature matrices and bug count vector sequentially (initial to the final version) and separately. Finally, accumulation of the feature matrix and bug count vector devise metadata of the software, as shown in Eq.4.5.

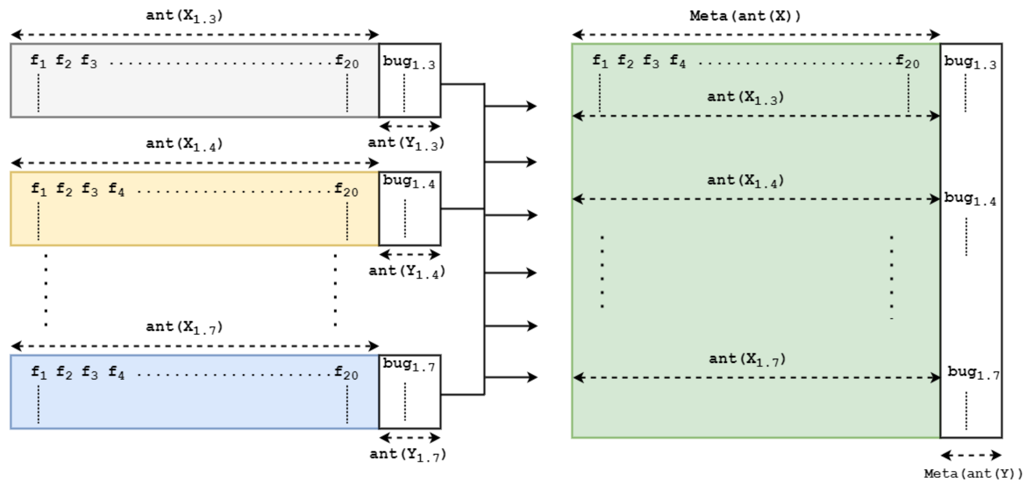


FIGURE 4.2: Meta-data collection process.

4.3 Proposed approach

In this section, we will discuss, first, the reason behind the launch of the new version of any software system, then illustrate the proposed approach and deep learning

architecture that addresses the problem statement. In the last, we will explain the performance measure that evaluates the proposed model.

Most software companies launch their newer version of the software system, which has more functionality than the previous version. Three main reasons why software companies launch the newer version of a software system are given below.

(a). Faster and additional functionality added from user feedback.

(b). New platform compatibility issues can occur with the old version.

(c). Corrective maintenance for the old version can be costly or cumbersome.

The prediction of the number of bugs in the upcoming version of a software system will be very efficient to schedule resources, and it minimizes testing effort as we have discussed in section 4.1. Fig. 4.3 shows the graphical abstract of our proposed work. The data instances are generated from the software repository; after data cleaning [199], the metadata generated from all prior versions of software as discussed in section 4.2 then data is ready for preprocessing according to the model.

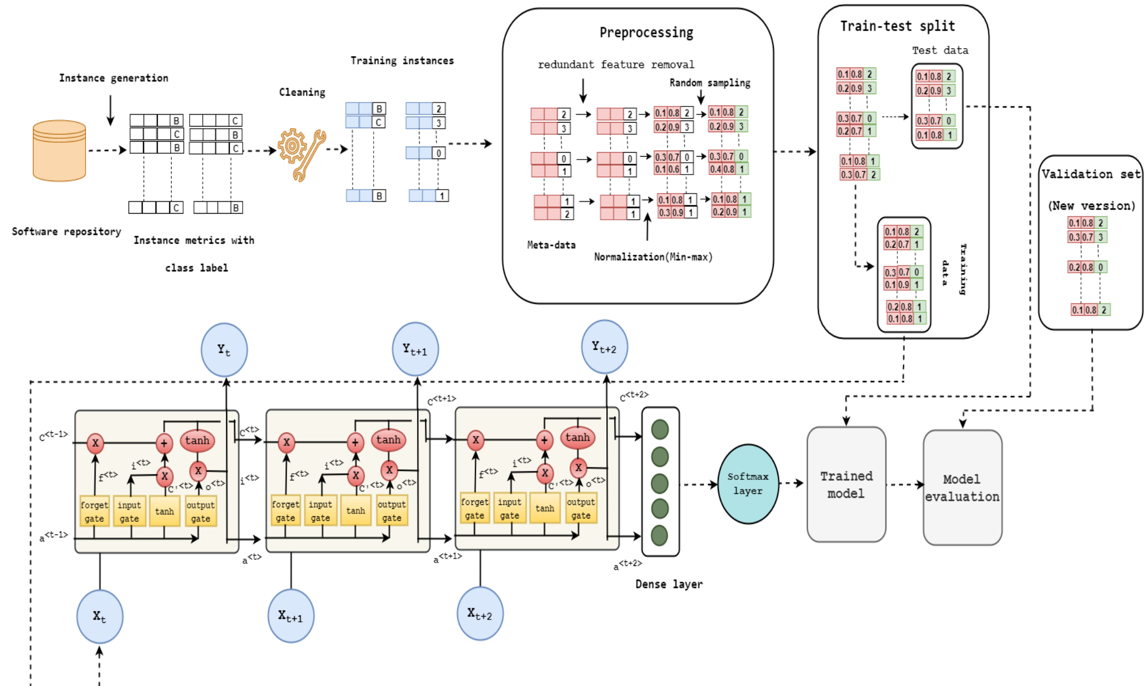


FIGURE 4.3: Graphical abstract of Proposed work.

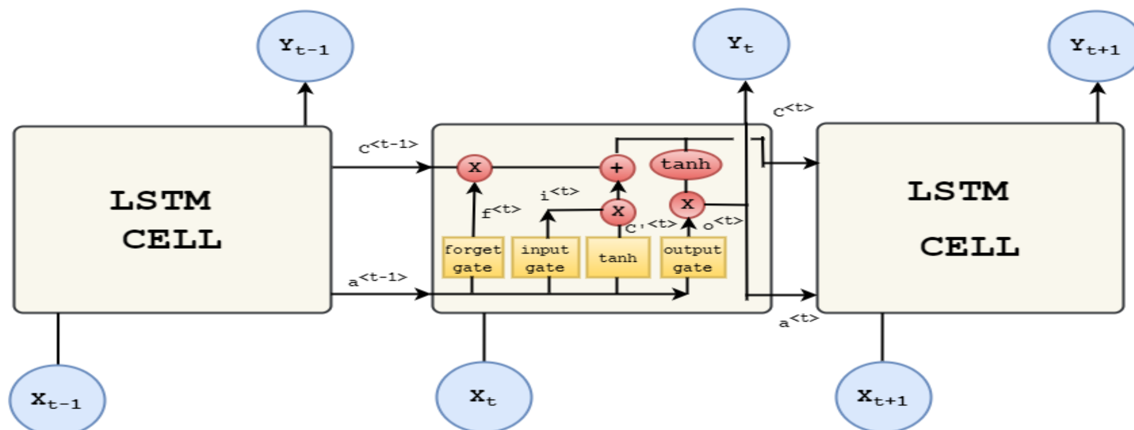


FIGURE 4.4: Basic architecture of LSTM network, and LSTM cell.

4.3.1 Preprocessing

We have used seven public PROMISE datasets, and then we generated metadata after concatenation of all versions of the same software system (shown in Table 4.1) as discussed in section 4.2.3. There are 20 columns features and 1 class label; the class label represents the number of bugs in each module. Feature vector needs to be normalized; we have used Min-max normalization [6] to scale the features between 0 to 1. Let x be the feature with instance i, min(x), and max(x) represent the minimum and maximum values of x. Then normalized value of x_i is z_i, which is calculated using Eq. 4.6. The preprocessing step is visible in the graphical abstract of the proposed work as shown in Fig. 4.3.

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{4.6}$$

As we have discussed in section 4.2.1, the meta-datasets have a different bug label, and it has a skewed distribution, which causes class imbalance problems [100]. Many researchers have suggested a solution to this issue. Oversampling and under-sampling of datasets [265] will avoid this challenge. We have applied a random over-sampling method [3] for meta-datasets.
Algorithm 2 is multi-label random oversampling [31] that we have used in a pre-processing step. The dataset and the percentage of imbalance of a class p are as input. Mean Imbalance Ratio (MIR) and Imbalance Ration per Label (IRL) were calculated during the cloning of minority labels. Algorithm 2 clones the minority

class according to their IRL and discard the label, which has high IRL; a complete explanation of this algorithm is given by Charte et al. [31].

4.3.2 Deep learning architecture

Recurrent Neural Networks (RNN) [217] are a type of neural network where the output from the previous step is feed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but when it is required to predict the next word of a sequential sentence, the previous words are needed, and hence there is a need to remember the previous words. It works fine when there are short sequences to train and finds it challenging when the sequence is large. It also suffers from a vanish gradient problem [93] and exploding gradient problem [187]. Long short term memory (LSTM) [94] overcomes these challenges. The heart of an LSTM network is it's a cell or, say, cell state, which provides a bit of memory to the LSTM so that it can remember the past or present sequences. We have used LSTM architecture in our proposed model as shown in Fig. 4.4. LSTM cell consists of three gates, "forget gate," "input gate," and "output gate," as given in Fig. 4.4. The gates of LSTM have a special ability to add or remove information to the cell. x_t, and y_t are input and output sequence at t time step. The w_c, w_i, w_f, w_o are the weight matrix for candidate cell, input gate, forget gate, and output gate respectively, whereas b_c, b_i, b_f, and b_o are bias value for candidate cell, input gate, forget gate, and output gate, respectively. Tanh is a activation layer, its value lies between [-1, 1], σ is activation function layer lies between [0, 1]. Step by step process of the LSTM cell is explained in section 2.2 (Eq. 2.3 to Eq. 2.8).

4.3.3 Proposed model

Long short term memory (LSTM) has the ability to remember, update & forget the past sequences, and it utilizes those sequences to predict the next sequences. We are trying to use this powerful ability of LSTM in the prediction of the bug count vector of the successive version software system. Metadata is feed into the LSTM network for training. We set the time step required in the LSTM network as t, for training the metadata of software S, as the number of all prior versions of S, i.e., t = n, to predict S_{n+1}. It will be easy for the model to learn the different functionality of all previous versions, and it can correctly predict the bug count vector on the same

functionality of the successive version.

Algorithm 3 represents the pseudo-code of the proposed method, S is the collection of a software system that has n versions. Let for every module of each version of S; the number of features is fixed, i.e., f. We assumed that the number of features in upcoming (S_{n+1}) is also f. The dataset related to each software S_1, S_2,...,S_n is d_1, d_2,...,d_n. The metadata $d_{M'}$ is the combination of all version of software system (S), $d_{M'}$ = d_1, d_2,....., d_n as discussed in section 4.2. Let X is a set of features, and Y is the target vector of $d_{M'}$. The instances of various features have different scaled values, so they need to be normalized. We have used min-max normalization, as discussed in 4.3.1 section. As we have discussed earlier, the meta-datasets (4.1) are imbalanced, so we have applied the random over-sampling method (see Algo. 2) in step 11 over feature vector X. The main objective to apply the sampling method is to avoid class imbalance problem as we have discussed in section 4.3.1.

After preprocessing the metadata splitted into two sections, training set, and testing set. We have used 70% as training data and 30% as testing data. We have also tried other variations of 60%—40%, 75%—25%, 80%—20% splitting in training and testing, respectively, but got optimal results over 70%—30% split.

X, Y are divided as $train_x$, $test_x$ & $train_y$, $test_y$ respectively. Fig. 4.3 shows the train-test split step. Then training data (X $\rightarrow$ $train_x$, & Y$\rightarrow$ $train_y$) feed into the LSTM layer with t time step, as t is the total number of the existing version of software system S, i.e., n. To avoid the chances of overfitting, we have used dropout regularization [236] method $\Delta = c$, a constant value for each layer of LSTM. Dropout is an efficient way to avoid overfitting by randomly dropout units along with their connections from the training set. At the training time, dropout samples from an exponential number of thinned networks. So, it is easy to validate approximate results from this thinned network. The values of Δ lie from 0 to 1. Let l be the index of the hidden layer from 1 to L, ζ_l represents the input vector for layer l, y^l is the output of layer l, w^l, b^l are weights and bias of layer l. f is the activation function.

$$\zeta_i^l = w_i^{l+1} * y^l + b_i^{l+1} \tag{4.7}$$

$$y_i^{l+1} = f(\zeta_i^{l+1}) \tag{4.8}$$

r^l is the vector of independent Bernoulli random variable, each of them has probability p. r^l is then sampled and multiplied element-wise with output layer y^l.

$$r_j^l \ Bernauli(p) \tag{4.9}$$

The thinned output is then used as an input to the next layer; this process repeats for every layer. All these processes are given in Eq. 4.7 to 4.12

$$\hat{y} = r^l * y^l \tag{4.10}$$

$$\zeta_i^{l+1} = w_i^{l+1} * \hat{y^l} + b_i^{l+1} \tag{4.11}$$

$$y_i^{l+1} = f(\zeta_i^{l+1}) \tag{4.12}$$

Now for epoch 1 to I, we have set *BatchSize* (batch size) = ϵ, where batch size [97] is a hyperparameter, which defines the number of samples to work through before updating the internal model. Till now no proper justification is available regarding value of batch size, we have used ϵ=128. Each LSTM layer computes C'$^{<t>}$, i^t, f^t, o$^{<t>}$, C$^{<t>}$, a$^{<t>}$ (as discussed in section 4.3.2).

$$\sigma(z_j) = \frac{e^{(z_i)}}{\sum_{k=1}^{K} e^{z_k}} \tag{4.13}$$

We have used Adam optimizer [125] to update network weights iterative based on training data. It combined both heuristic optimization method RMSProp [213] and Momentum [244] impedes search in direction of oscillations.

$$v_t = \beta_1 * v_{t-1} - (1 - \beta_1) * g_t \tag{4.14}$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2 \tag{4.15}$$

$$\Delta_{\omega_t} = -\eta \frac{v_t}{\sqrt{s_t + \hat{\epsilon}}} * g_t \tag{4.16}$$

$$\omega_{t+1} = \omega_t + \Delta_{\omega_t} \tag{4.17}$$

Here η is initial learning rate, g_t is gradient at time t, v is exponential average of gradient along with ω_t, s exponential average of square of gradient along with ω_t, and β_1 & β_2 are hyperparameters.

Model compute κ_q, which is a probability score of each bug q, for metadata d$_{M'}$. Every hidden unit computes the probability score, the softmax layer (Eq. 4.13) validates the computed probability score of the trained set with the validation set.

We have utilized Sparse categorical cross-entropy as a loss function, as shown in Eq. 4.18 to calculate the loss. When the target label is not a one-hot vector, then this

Algorithm 2: Multi label random over sampling

```
1  Inputs ← d, p                          /* Dataset & percentage of imbalance */
2  Output → Preprocessed dataset
3  CloneSample ← | d | / p*100
4  l ← DatasetLabel(d) // Datset label copied
5  MIR ← calculate mean imbalance ratio(d,l)
6  foreach label in l// packet of minority class
7   do
8      IRL ← calculate imbalance ration per label(d, l)
9      if IRL_i > MIR then
10         minPacket_{i++} ← Bag_i
11     end
12 end
13 foreach CloneSample >0                                   /* loop for clone */
14 Clone a random sample from each monority packet do
15     foreach minPacket_i in minPacket do
16         x←random(1,minPacket_i) CloneSample(min_Packet)
17         if IRL_minPacketi <=MIR then
18             min_Packet → min_Packeti
19         end
20         CloneSample // exclude from cloning
21     end
22 end
```

loss function is more preferable [45].

$$L(\hat{Y}, Y) = -\frac{1}{M'} \sum_{i=1}^{M'} [y_i \log \hat{y} + (1 - y_i) \log(1 - \hat{y}_i)] \tag{4.18}$$

y_i is the class label of ith module in the training set, $\hat{y}_i$ is the predicted class of the same module. The Y is an overall original class label, and $\hat{Y}$ is an overall predicted class of validation set.

For each module i, we have compared the target label y_i, with predicted value $\hat{y}_i$ and compute mean squared error, mean absolute error, and accuracy. Afterward, we finally compute overall loss L($\hat{Y}$, Y). The details of performance metrics are given in section 4.3.5.

4.3.4 Experimental arrangements

We have performed our experiments over the NVIDIA GPU server of 16GB RAM, CUDA version 10, NVIDIA-SMI 410.104. We have used anaconda version 3, Tensorflow, as a backend over a Keras library. Additionally, we have used Numpy as a linear algebra library, Pandas, & Sklearn for data interpretation. Iblearn, Scipy, & Seaborn for sampling methods and Matplotlib for data visualization.
It is an evolutionary process to tune the hyperparameters, and we have tested different values of various hyperparameters to achieve optimal results. For every metadata, we have used three LSTM hidden layers and one fully connected layer. The first, second, and third hidden layers have 100, 80, and 60 hidden units. Whereas 0.2 dropout rate in all three LSTM layers. In most of the meta-datasets, we have used 128 batch sizes. All the experiments were conducted over 450 to 500 epochs. The time step is the same as the number of versions of the software, the validation split is 0.2, and the rest of the parameters are set as default values of WEKA tool [74].

4.3.5 Performance metrics

To evaluate the performance of the BCV-Predictor, we have used MAE, MSE, and accuracy performance measures; the brief illustration of these performance matrices are given in section 2.4. We have considered both overall, and validation set performance metrics.

4.3.6 Baseline methods

We feed all the seven meta-datasets into eleven state-of-the-art techniques and evaluate their performance. The performance of Linear Regression (LR), Multilayer Perceptron (MLP) [153], Instance-based learning (IBK) [5], Additive Regression (AR) [272], M5rules [201, 56], M5P [201], Bagging [241], Gaussian Process (GP) [230], Decision Stump [96], Random Forrest [28], and Regression by Decentralization (RD) [65] are compared with BCV-Predictor shown in Table 4.3 to Table 4.5. Most of these methods have been widely used in software bug count prediction and other estimation/prediction applications.
We have considered a similar preprocessing scenario as BCV-Predictor for all other

Algorithm 3: Proposed algorithm

```
1  S ← S_1, S_2,....,S_n                              // Software S has n versions
2  for each M_j of S_i do
3  |  f is fixed      // number of features in each module is fix and positive
   |   interger
4  end
5  S_{n+1} version of S also contains f features.
6  d_{M'} ← data instance of S                         // meta-detaset from S_1 to S_n
7  d_{M'} = d_1, d_2,....., d_n
8  X← all features of d_{M'}                                      // feature vector
9  Y← target labels of d_{M'}                                      // target vector
10 Scaling_{MinMax}(X)  // Normalization of each feacture of X, using Eq. 4.6
11 Sampling_{random}(X,Y)              // random oversampling, using Algo. 2
12 X→ train_x, test_x & Y→ train_y, test_y                        // train test split
13 sequential_{LSTM}(l, train_x, train_y, Δ = c, time-step = n)   // trainning set feeded
    to LSTM with l hidden units and fix droupout rate
14 foreach epoch I, BatchSize = ε do
15 |  for each sequencial_{LSTM} layer do
16 |  |  C'^{<t>} u^t , f^t, o^{<t>}, C^{<t>}, a^{<t>}              // from Eq. 2.3 to 2.8
17 |  |  κ_q = P_{score}(q) of d_{M'}                           // score of each bug q
18 |  end
19 |  Optimize_{adam}(Ŷ, Y)            // update weight from Eq. 4.14 to 4.17
20 |  Compute σ(z_j) = e^{(z_i)} / Σ_{k=1}^{K} e^{z_k} for each hidden unit   // compute softmax
   |   function
21 |  Compare(κ_q, κ_Q)                           // compared over validation set
22 |  Compute L(Ŷ, Y) = - 1/M' Σ_{i=1}^{M'}[y_i log ŷ + (1-y_i)log(1 − ŷ_i)]  // Loss calculate
23 |  Calculate MSE(y, ŷ), MAE(y, ŷ), Accuracy(y, ŷ).
24 end
25 Evaluate M using MAE(Y, Ŷ), MSE(Y, Ŷ), Accuracy((Y, Ŷ)).  // model evation
    after I epoch
```

baseline methods to compare their performance with the proposed model. We used 100 batch sizes for all the methods. Poly kernel, and seed as 1 in GP. Ridge is 1.0E-8 in LR. Learning rate 0.3, momentum 0.2, training time 500, and validation threshold 20 in LR. KNN as 1, linear NN search as nearest neighbor search algorithm in IBK. Seed 1, RepTree as a base classifier in bagging. J48 as a base classifier, univarientEqualFrequncyHistogram as an estimator, ten bins in RD. Bag size percent as 100, max depth 10, seed 1 in RF. We set the rest of the parameters/hyperparameters as the default value of the WEKA tool [74] of all the eleven techniques.

4.3.7 Results and explanations

For the sake of diversity and unbiasedness, we have conducted each experiment 30 times and taken the mean of each performance metric and training time over every seven projects. We evaluated the model's performance using MSE, MAE, & accuracy of over the complete model and over the validation set. Table 4.2 represents the MSE, MAE, Accuracy, Val_{mse}, Val_{mae}, and $\text{Val}_{accuracy}$ values produced by the BCV-Predictor.

Table 4.2: Performance of BCV-Predictor.

Data	**MSE**	**MAE**	**Accuracy(%)**	**Val**$_{mse}$	**Val**$_{mae}$	**Val**$_{accuracy}$**(%)**
ant	2.979	0.453	87.70	2.925	0.450	86.44
camel	1.279	0.880	88.24	1.3039	0.886	88.00
jedit	3.33	0.52	84.86	3.98	0.56	79.52
prop	0.71	0.22	87.67	0.84	0.22	87.61
xerces	4.715	1.679	91.95	4.865	1.709	91.44
xalan	2.751	0.440	66.41	2.716	0.436	66.87
poi	2.68	0.90	55.12	5.61	1.15	49.85

4.3.7.1 Justification of RQ-1

The MSE and MAE of the complete model over all the meta-datasets are shown in the first and second columns of Table 4.2. The lowest MSE value is of prop, i.e., 0.71, followed by camel with 1.279 MSE value. After that, poi, xalan, ant, jedit, and xerces have lower MSE values of 2.68, 2.751, 2.979, 3.33, 4.715, respectively. The increasing order of MAE values of prop, xalan, ant, jedit, camel, poi, and xerces projects are 0.22, 0.440, 0.453, 0.52, 0.880, 0.90, and 1.679, respectively. Fig. 4.7(b) shows the overall MSE value of all metadata. Xerces has the highest pick, whereas prop has the lowermost curve in the bar graph. The rest of the projects have moderate height in the bar graph, which represents a moderate loss. Fig.4.7(a) shows the bar graph of the overall MAE value produced by the BCV-Predictor. The lowest pick is of prop project, whereas the highest is of xerces software project, indicating a maximum loss. Jedit and xalan also have high MAE loss. Ant and xalan have almost approx equal MAE values.
Table 4.2 indicates that the accuracy of xerces is highest with 91.95%, followed by camel, ant, prop, jedit, xalan, and poi with 88.24%, 87.70%, 87.67%, 66.41%, and 55.12% respectively. Whereas the accuracy over validation set ($\text{Val}_{accuracy}$) for ant, camel, jedit, prop, xerces, xalan, and poi are 86.44%, 88%, 79.52%, 87.61%, 91.44%,

66.87%, and 49.85% respectively. Fig.4.7 represents the bar graph of MAE, MSE, and accuracy of BCV-Predictor over every project. Fig.4.7(c) shows the bargraph of overall accuracy in all seven projects. Xerces has the highest pick plot, and poi has lowest pick plot.

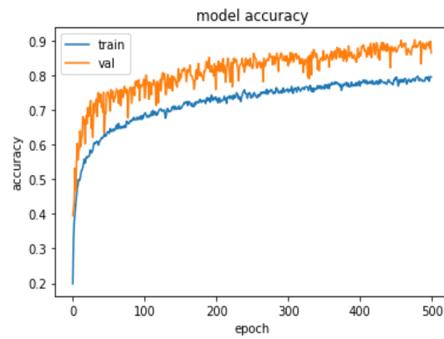
The MSE over validation set (Val_{mse}) is shown in fifth column of Table 4.2. The minimum Val_{mse} is of prop metadata, which is 0.22. The increasing order of Val_{mse} for rest of the meta-datasets i.e., xalan, ant, jedit, camel, poi, and xerces are 0.436, 0.450, 0.56, 0.886, 1.115, and 1.709 respectively.

The sixth column of Table 4.2 reflects the MAE over validation set (Val_{mae}). Similar to MAE, the Val_{mae} follows the same increasing order. The prop metadata has the lowest Val_{mae} value, i.e., 0.22. Then after Val_{mae} values for xalan, ant, jedit, camel, poi, and xerces are 0.436, 0.450, 0.56, 0.886, 1.15, and 1.709 respectively.

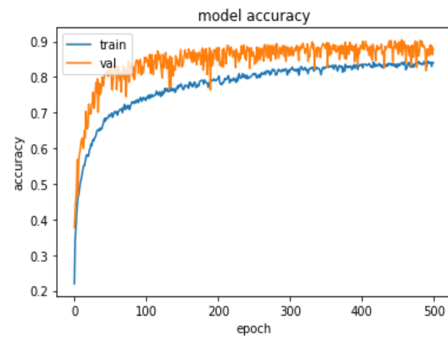
The accuracy over validation set ($\text{Val}_{accuracy}$) is shown in the last column of Table 4.2. Similar to overall accuracy, the validation set accuracy also follows a similar order and approximate equal values. The maximum accuracy is produced for xerces metadata, i.e., 94.44%. The accuracy of camel, prop, ant, jedit, xalan, and poi have values 88.00%, 87.61%, 86.44%, 79.52%, 66.87%, and 49.85% respectively. Fig. 4.7(c) shows the accuracy produced by the BCV-Predictor over all meta-datasets.

The accuracy over both the training set and validation set of the proposed model over 500 epochs is shown in Fig. 4.5. The blue and orange lines in all sub-figures from Fig. 4.5(a) to Fig. 4.5(g) represents accuracy plot over the train set and validation set till 500 epoch. In Fig. 4.5(a), the accuracy over validation set is approx 85%-87% whereas over the training set, it is 79% to 81% for ant metadata. Model is stable over ant project because the variation in overall accuracy is negligible after 200 epoch. In Fig. 4.5(b), accuracy on the validation set lies between 86% to 88%, but for the train set, it lies between 79% to 82% from epoch 400 to 500 for camel metadata, the model is stable over camel software. Fig .4.5(c) shows for jedit metadata, the accuracy is linear till 400 epoch; it is because of lesser data instances, the model is moderately stable over this project. As prop metadata has sufficient data to train, the validation accuracy is from 0.8760 to 8465, whereas the train set accuracy is from 0.88775 to 0.8780. The suggested model is highly stable over prop software projects. Fig. 4.5(e) reports that the validation and train set accuracy varying with the epoch of xerces metadata. Still, validation set accuracy is always more than train set accuracy, and the validation accuracy lies between 85% to 91%. BCV-Predictor is stable over xerces; it has a sufficient number of instances in the

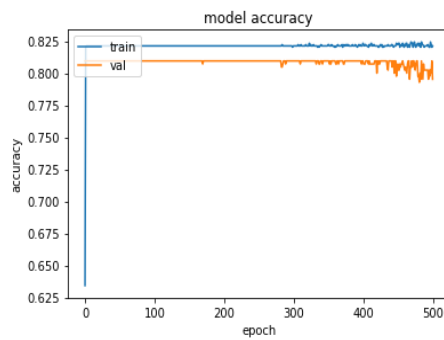
training set. Fig. 4.5(f) shows the accuracy over the validation set is varying from 0 to 500 epoch, unlike train set accuracy, validation set accuracy ranging from 61% to 70% after 400 epoch of xalan metadata, the proposed model is unstable due to lack of training instances. The train set accuracy is higher than the validation set accuracy in poi metadata, as shown in Fig. 4.5(g), both train and validation set accuracy varying throughout the epochs (1 - 500), the validation set accuracy after 400 epoch lies between 45% to 55%. Poi has very few meta-data instances that lead to an unstable model. In Fig. 4.6, we have aggregated the plot between epoch and loss. Here blue and orange lines indicate loss over the train set and validation set, respectively. When the train set curve and validation set curve diverges imply, the model is over-fitted. Whereas when the curves converge or overlapping are good fitted curves and model circumvent overfitting problem. In Fig. 4.6(a), Fig. 4.6(b), Fig. 4.6(e), & Fig. 4.6(f) we can see the loss over the train set of ant, camel, xerces and xalan metadata respectively, are lesser than the loss over validation set in every epoch. Whereas in Fig. 4.6(c), Fig. 4.6(d), & Fig. 4.6(g), the loss of jedit, prop, poi metadata, respectively, have more validation set loss, than train set loss. The loss over validation data for ant metadata (Fig. 4.6(a)), lies from 0.25 to 0.5 after 400 to 500 epoch. Validation loss over camel metadata (Fig. 4.6(b)) lies from 0.25 to 0.55 after 400 epoch. The validation loss for jedit moderately varies with epoch but still more than train loss, which lies from 0.5 to 0.75 from 200 to 500 epoch. The validation and train set losses for prop metadata is overlapping from 0 to 220 epoch, as Fig. 4.6(d) reports, both lie between 0.450 to 0.475. Unlike in xerces metadata, the training loss is always more than validation loss and lies from 0.2 to 0.55 after 100 epochs. More variation on validation loss of xalan metadata, as shown in Fig. 4.6(f), the validation loss lies from 1.2 to 0.4 after 400 epoch. Lastly, the validation loss gradually increases with epoch because of lesser data instances, and it lies between 1.25 to 1.75 after 100 epochs. The loss curve of ant, camel, prop, xalan, and xerces are good fitted curves; it concludes that BCV-Predictor avoids overfitting problem. The loss curve of jedit is also diverging, as shown in Fig. 4.6(c), but the difference is negligible, so it also lies under a good fitted curve, so it also circumvents overfitting. The loss curve of poi project starts diverging, and the loss difference is too high, as given in Fig. 4.6(g), so it is suffering from overfitting.



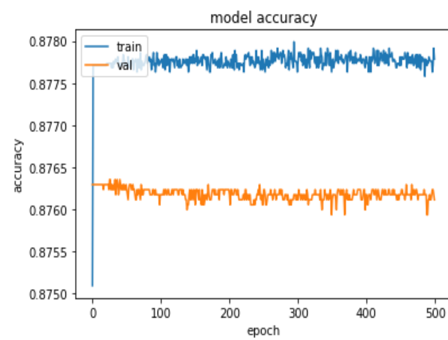
(a) Accuracy of ant metadata.



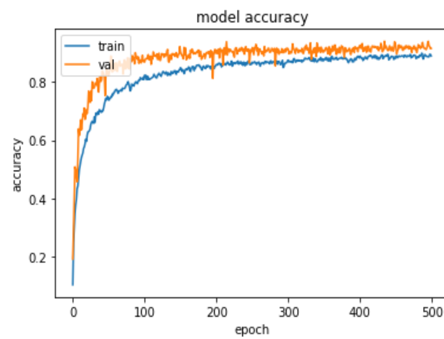
(b) Accuracy of camel metadata.



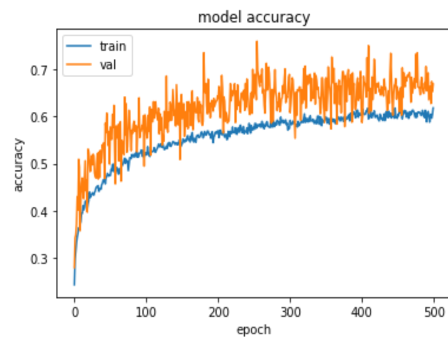
(c) Accuracy of jedit metadata.



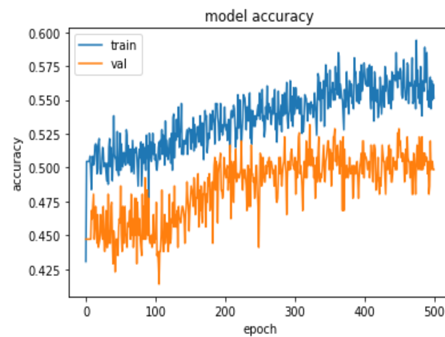
(d) Accuracy of prop metadata.



(e) Accuracy of xerces metadata.



(f) Accuracy of xalan metadata.



(g) Accuracy of poi metadata.

FIGURE 4.5: Accuracy of all seven metadata.

Table 4.3: MAE comparison of BCV-Predictor with other techniques.

Data	**LR**	**MLP**	**IBK**	**AR**	**M5rules**	**M5P**	**Bagging**	**GP**	**DS**	**RF**	**RD**	**BCV-Pred.**
ant	0.51	0.58	0.47	0.45	0.44	0.44	0.47	0.52	**0.43**	0.47	0.51	0.44
camel	0.73	1.1	**0.70**	0.77	0.73	0.73	0.74	0.74	0.79	0.71	0.83	0.88
jedit	0.75	0.71	0.67	0.78	0.75	0.75	0.76	0.69	0.83	0.66	0.79	**0.52**
prop	0.34	0.69	0.31	0.33	0.34	0.34	0.33	1.28	0.32	0.31	0.37	**0.22**
xerces	1.49	1.59	1.41	1.49	1.41	1.49	1.39	1.48	1.49	**1.37**	1.51	1.67
xalan	0.68	0.69	0.71	0.67	0.68	0.64	0.62	0.68	0.71	0.64	0.63	**0.44**
poi	0.96	1.13	1.11	1.01	0.94	0.93	0.95	0.96	1.1	0.99	1.1	**0.90**

Table 4.4: MSE comparison of BCV-Predictor with other techniques.

Data	**LR**	**MLP**	**IBK**	**AR**	**M5rules**	**M5P**	**Bagging**	**GP**	**DS**	**RF**	**RD**	**BCV-Pred.**
ant	0.604	0.702	0.98	0.683	**0.518**	0.653	0.687	0.655	0.790	0.654	0.686	2.977
camel	2.23	3.23	3.56	3.81	2.54	2.61	2.50	3.21	2.48	3.84	2.91	**1.227**
jedit	3.80	3.73	4.85	3.65	3.81	3.73	3.62	3.57	3.67	3.51	3.60	**3.33**
prop	0.76	0.85	0.98	0.78	0.79	0.77	0.79	2.54	0.789	0.77	0.78	**0.71**
xerces	10.23	24.3	10.22	10.6	10.25	12.65	11.97	10.88	11.96	10.13	11.93	**4.71**
xalan	0.75	0.84	1.33	0.76	0.95	0.73	**0.72**	0.71	0.82	0.78	0.85	2.74
poi	2.95	6.26	3.25	3.17	2.84	2.94	2.73	2.99	3.58	2.96	3.18	**2.68**

Table 4.5: Accuracy comparison of BCV-Predictor with other techniques.

Data	**LR**	**MLP**	**IBK**	**AR**	**M5rules**	**M5P**	**Bagging**	**GP**	**DS**	**RF**	**RD**	**BCV-Pred.**
ant	82.54	76.12	73.26	75.29	84.37	83.98	83.71	83.99	81.24	84.51	82.27	**87.7**
camel	83.65	72.33	74.35	77.62	85.04	83.10	84.19	85.12	80.87	85.37	83.13	**88.24**
jedit	78.54	81.12	68.82	70.38	79.63	78.76	79.87	81.31	78.85	81.22	80.1	**84.52**
prop	76.32	72.21	68.21	70.67	84.09	82.68	83.52	84.68	80.31	85.02	82.64	**87.67**
xerces	82.33	80.12	74.32	76.85	86.97	86.53	87.16	87.92	84.12	88.58	86.81	**91.65**
xalan	71.21	67.21	62.58	67.76	70.22	70.11	71.21	72.31	68.13	**72.72**	70.64	66.41
poi	72.68	69.55	63.73	65.91	74.10	73.19	74.25	75.33	71.11	**75.37**	73.42	55.12

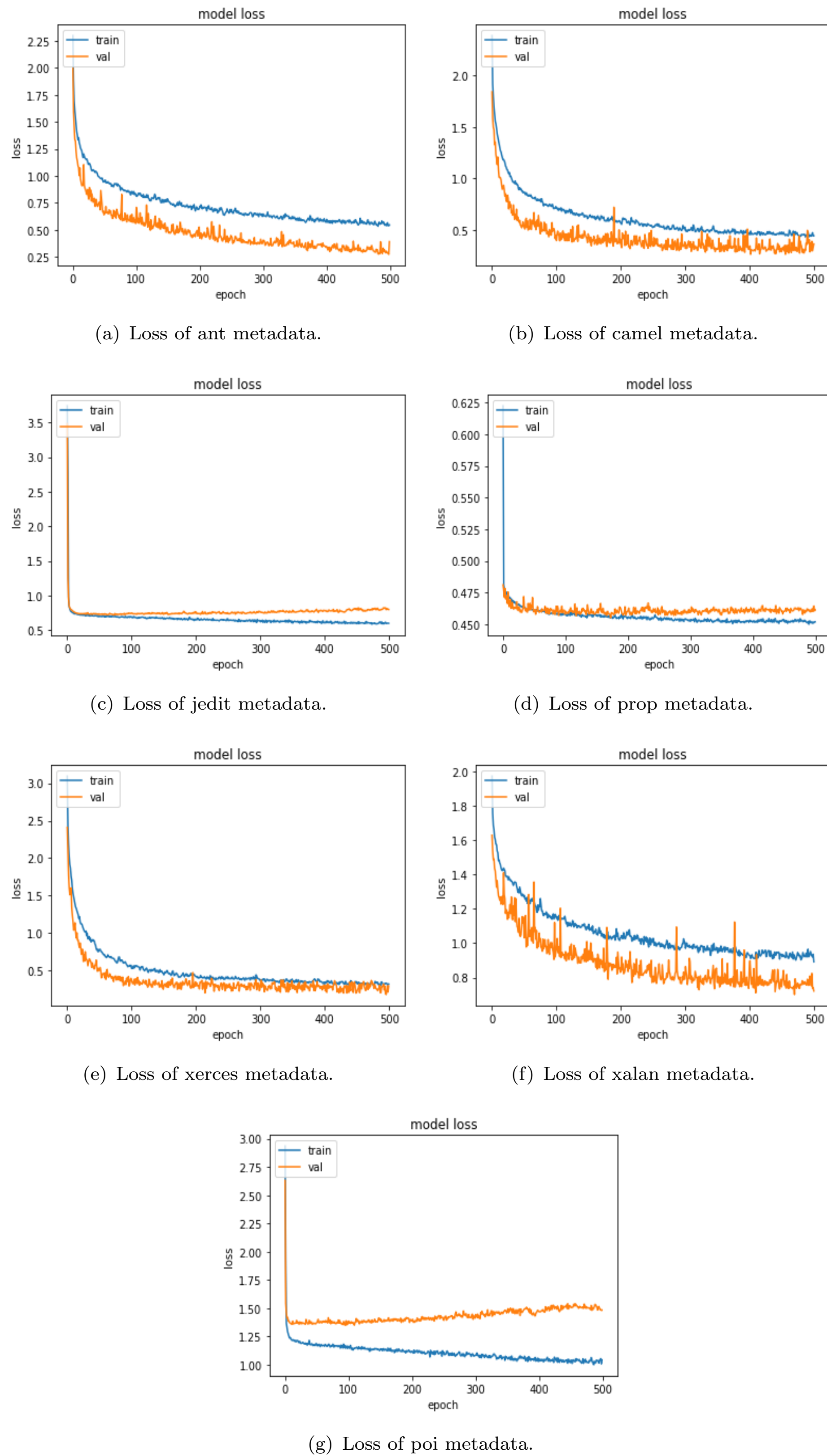
4.3.7.2 Justification of RQ-2

Table 4.3 compares the MAE values of various techniques with BCV-Predictor. As shown in Table 4.3, four out of seven meta-datasets have lesser MAE values compared with other state-of-the-methods. The MAE values of BCV-Predictor over jedit, prop, xalan, and poi metadata have low values compared with other baseline techniques. The lowest MAE of ant, camel, and xerces projects are processed by DS, IBK, and RF models, respectively. Table 4.4 compares the MSE values of different models with our proposed approach. The MSE produced by the BCV-Predictor on camel, jedit, prop, xerces, and poi meta-datasets is less than other approaches. M5rules and Bagging have the lowest MSE for ant and xalan projects, respectively. The accuracy of the proposed model is high over five meta-datasets. The accuracy of BCV-Predictor over ant, camel, jedit, prop, and xerces metadata has more elevated than the rest of the other eleven techniques, as shown in Table 4.5. In contrast, RF surpasses the accuracy over BCV-Predictor for xalan and poi meta-datasets.

4.3.8 Insightful discussion

We have conducted experiments over seven meta-datasets; few of them do not have enough data instances for finer training purposes. As Table 4.1 indicates, poi, jedit, xerces, and ant metadata have 1378, 1257, 1644, and 1694 data instances. Deep learning methods are hunger of data instance, as instances are more, training of a learning model becomes healthier, and it leads to satisfactory results [38] with lesser chances of over-fitting. In Fig. 4.5(c), we can see the validation set and train set accuracy is almost unvarying with respect to epoch after 300 epoch. Although slight variation started; it is because of very lesser instances in the validation set. A similar effect is also spotted in Fig. 4.6(c), the validation loss somehow increases concerning the epoch. If BCV-Predictor has more training data, better accuracy can be achieved.

As we observe in Table 4.2 and Fig. 4.5(g), the accuracy of poi metadata is 55.12%, and on the validation set, it is 49.85%, which is not that effective. It is mainly due to the lack of data points in the training set. High loss occurs during training and validation (see Fig. 4.6(g)). Data imbalance [266] also plays a crucial role in such a scenario. Xalan is also facing a similar problem, as we can see in Table 4.2, the accuracy of xalan is 66.41%, and MSE is 2.751. That indicates a high loss during



(a) Loss of ant metadata.

(b) Loss of camel metadata.

(c) Loss of jedit metadata.

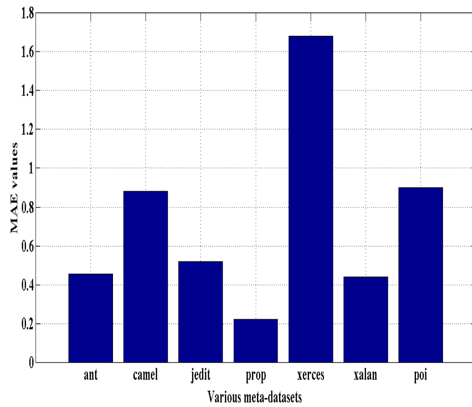
(d) Loss of prop metadata.

(e) Loss of xerces metadata.

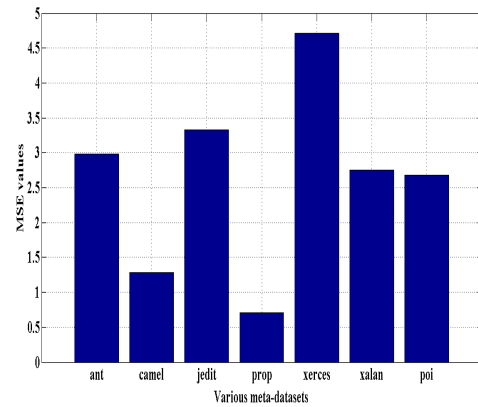
(f) Loss of xalan metadata.

(g) Loss of poi metadata.

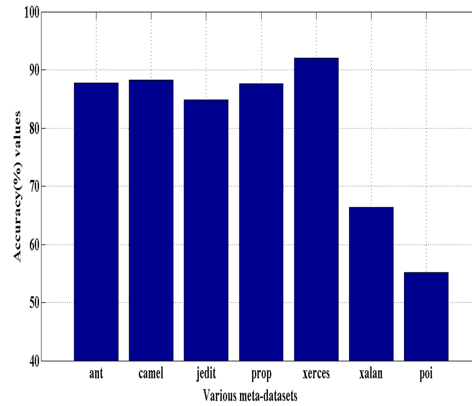
FIGURE 4.6: Loss of all seven metadata.



(a) Bar-graph of MAE over all meta-datasets.



(b) Bar-graph of MSE over all meta-datasets.

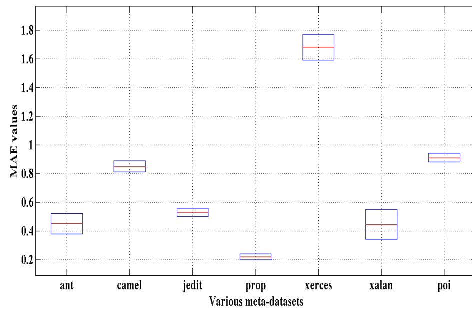


(c) Bar-graph of Accuracy over all meta-datasets.

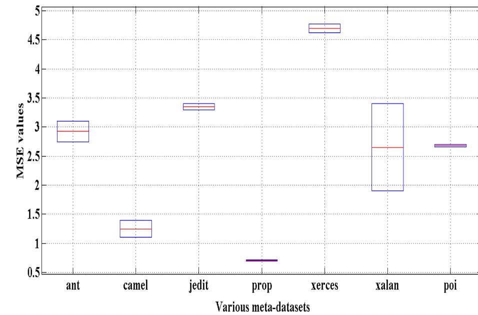
FIGURE 4.7: Bar-graph of MAE, MSE, & accuracy over all meta-datasets.

prediction and misses the correct bug number in the bug count vector. Fig. 4.6(f), we can see the loss over the train set is to 1, i.e., between 0.8 to 1 for the validation set.

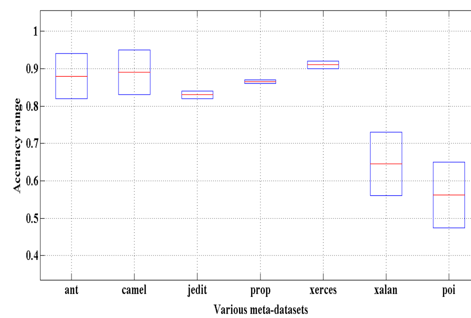
The stability of the model is a principal aspect; to state the stability of the BCV-Predictor. As we discussed earlier that we had conducted every experiment 30 times, the boxplot shows the range of each performance metric as shown in Fig.4.8 during experiments. We considered 500 epoch because the loss versus accuracy graph in Fig. 4.6 is almost stable after 300 epoch; before that loss is moderate for xalan and poi metadata. Fig.4.8(a) shows that excluding xalan all the other six metadata have almost no variation in MAE values, which indicates a stable model. Whereas some disparity is visible in xalan, which is acceptable. Similarly, the only xalan has variation in MSE after 300 epoch, as shown in Fig. 4.8(b), which makes lesser



(a) Boxplot of MAE over all meta-datasets.



(b) Boxplot of MSE over all meta-datasets.



(c) Boxplot of Accuracy over all meta-datasets.

FIGURE 4.8: Boxplot of MAE, MSE, & accuracy over all meta-datasets after 300 epoch.

stable. In Fig.4.8(c), the range of accuracy is shown after 300 epoch, its easily visible that jedit, prop, and xerces, have a tiny difference that implies high stability. In comparison, xalan and poi have some variations that specify small-scale instability, which can be acceptable. BCV-Predictor is a moderately unstable model over xalan metadata, whereas it is stable over the rest of the six meta-datasets. Small training set in xalan software is the reason for instability.
A list of factors that affect the performance of the BCV-Predictor is given below.

(i) Lack of training instances, which causes miss in accurate prediction of target bug number.

(ii) Skewed distribution of bugs in a different module.

(iii) There is always a scope of hyperparameters tuning to enhance the performance of the deep learning model.

4.3.8.1 Justification of RQ-3

Table 4.6 listed the training time of BCV-Predictor and other eleven methodologies. Time is given in seconds, μ (micro) second, and millisecond (ms). The training time of BCV-Predictor is more from all the methods except GP and MLP. Prop has the most massive training set. The training time for prop is 17sec+43μ sec, which is significantly less than RF and MLP. This implies when the software project is large, the BCV-Predictor outperforms compared with other techniques. The complex structure of BCV-Predictor causes of long training time. The proposed model is beneficial for large software systems; in such cases, it can efficiently predict the bug count vector with less computational cost and high accuracy.

4.3.9 Non-Parametric test

We performed the Wilcoxon Signed-Rank Test [275], a non-parametric test, to testify the significant results of the actual bug versus the predicted bug in the module by BCV-Predictor. First, we collected the actual bug count of an existing version of the software and compared them with our predicted bug count vector of the same version. Let S be a software that has n different version, so we created metadata using a concatenation of the n-1 version. Feed that metadata into BCV-Predictor for training and to predict the bug count vector of nthversion. Then compare the predicted bug count vector with existing nth version of S. In Table 4.7, we have aggregated the actual bugs (y) and predicted bugs ($\hat{y}$) of a different module of a nth software version. The non-parametric tests make no assumptions about the distribution of asses samples. The null hypothesis is that the median difference between the pairwise samples, i.e., actual and predicted number of bugs, is zero. We used a two-tailed test with $\alpha = 0.05$, indicating 95% of the confidence level. The mean difference of the actual and predicted bugs is 16.54, the sum of the rank is 405, the z-value is -4.6226, and the p-value is < 0.00001. The critical value for W at $N = 28$ ($p < 0.05$) is 101. The test concluded that the null hypothesis is significant at $\alpha = 0.005$.

Table 4.6: Training time (second) of various models compared with BCV-Predictor.

Data	**LR**	**MLP**	**IBK**	**AR**	**M5rules**	**M5P**	**Bagging**	**GP**	**DS**	**RF**	**RD**	**BCV-Pred.**
ant	0.01	3.33	0.04	0.05	0.18	0.2	0.11	7.51	0.01	0.53	0.05	4s+465μs
camel	0.11	5.59	0.01	0.11	0.63	0.71	0.44	29.59	0.05	1.36	0.27	2s+1μs
jedit	0.01	3.39	0.05	0.04	0.28	0.14	0.18	7.61	0.08	0.54	0.03	6s+6ms
prop	0.3	187.32	0.03	3.02	10.03	9.29	16.44	738.6	0.32	49.18	8.14	17s+435μs
xerces	0.01	3.15	0.09	0.04	0.32	0.17	0.07	7.2	0.07	0.04	0.04	3s+3ms
xalan	0.14	5.78	0.01	0.11	1.29	0.5	0.02	48.64	0.07	1.63	1.21	8s+899μs
poi	0.01	2.33	0.09	0.04	0.37	0.14	0.09	4.16	0.06	0.61	0.06	5s+6ms

TABLE 4.7: Actual and predicted values of bugs in different samples.

Meta-data	**Actual bug(y)**	**Predicted bug($\hat{y}$)**
	1	1
	2	1
ant	4	3
	5	4
	10	8
	4	4
	10	9
camel	13	11
	17	15
	28	25
	2	2
	7	6
jedit	17	15
	23	16
	45	31
	12	11
	15	13
	27	24
prop	20	19
	37	32
	3	3
	4	4
xerces	11	7
	30	21
	62	45
	3	3
	4	3
xalan	7	5
	8	6
	9	7
	2	2
	3	2
poi	11	6
	19	11
	20	12

4.4 Summary

Predicting the bug count vector of the upcoming version of the software system will reduce testing effort and the cost of software development. It also helps the project manager to suitably allocate developers and testers in software development. We have collected different versions of a software system and concatenate the versions to make metadata. We proposed a novel architecture, i.e., bug count vector predictor (BCV-Predictor), that predicts the bug count vector in the next version of a software system. The meta-datasets are used to train BCV-Predictor. LSTM deep learning architecture is used as a learning algorithm in BCV-Predictor and the number of versions as a time step. After performing experiments over seven meta-datasets, we found that LSTM is very effective in such predictions application. As we get high accuracy over most of the meta-datasets. We have applied dropout regularization and multi-label random oversampling to avoid overfitting and class imbalance problems, respectively. Out of seven, five meta-datasets have more than 80% accuracy, which is sufficiently high. Six out of seven and five out of seven metadata have MAE less than 0.9 and MSE less than 3, respectively. We compared the performance of

BCV-Predictor with eleven other state-of-the-art techniques. We found that BCV-Predictor produces higher accuracy on ant, camel, jedit, prop, and xerces compared to baseline methods. Over four out of seven and five out of seven metadata, the BCV-Predictor has lesser MAE and MSE values, respectively, compared with other state-of-the art methods. BCV-Prediction is more efficient over extensive software system; it takes lesser computational cost than baseline methods and gives better results.

In a real application, it is rare to get sufficient training data from the newer software project. So the SDP model produces poor performance over new projects; it comprises many diverse software projects released for defect prediction. In the next work, we employed projects of diverse software systems to overwhelm such issues.