Knowledge-Based Systems 197 (2020) 105924

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

BCV-Predictor: A bug count vector predictor of a successive version of the software system

Sushant Kumar Pandey [*], Anil Kumar Tripathi

Department of Computer Science and Engineering, Indian Institute of Technology (BHU), Varanasi, India

ARTICLE INFO

Article history:
Received 29 February 2020
Received in revised form 5 April 2020
Accepted 16 April 2020
Available online 18 April 2020

Keywords:
Number of bugs prediction
Bug count vector
Deep learning
Regression
Long short term memory (LSTM)

ABSTRACT

Predicting the number of bugs in a software system intensifies the software quality and turns down the testing effort required in software development. It reduces the overall cost of software development. The evolution of hardware, platform, and user requirements leads to develop the next version of a software system. In this article, we formulate a problem and its novel solution, i.e., we are considering the prediction of the bug count vector of a successive version of a software system. After predicting the bug count vector in the next version of a software, the developer team leader can adequately allocate the developers in respective fault dense modules, in a more faulty dense module, more number of developers required. We have conducted our experiment over seven PROMISE repository datasets of different versions. We build metadata using a concatenation of different versions of the same software system for conducting experiments. We proposed a novel architecture using deep learning called BCV-Predictor. BCV-Predictor predicts the bug count vector of the next version software system; it is trained using metadata. To the best of our knowledge, no such work has been done in these aspects. We also address overfitting and class imbalance problem using random oversampling method and dropout regularization techniques. We conclude that BCV-Predictor is conducive to predicting the bug count vector of the next version of the software. We found five out of seven meta datasets reaches to more than 80% accuracy. In all seven meta datasets, Mean Squared Error (MSE) lies from 0.71 to 4.715, Mean Absolute Error (MAE) lies from 0.22 to 1.679, MSE and MAE over validation set lie between 0.84 to 4.865, and 0.22 to 1.709 respectively. We also compared the performance of BCV-Predictor with eleven baselines techniques and found the proposed approach outperform on most of the meta-datasets.

1. Introduction

Software reliability is the probability of failure-free software operation for a specified period in a specified environment. Software reliability is a salient feature of any software system. Bugs in the software system produce unanticipated behaviors; even in the worst scenario, it can cause substantial economic loss. During software development, detecting the number of bugs reduces enormous testing efforts and prevents such economic losses. Software developer wants to provide bugs free software product, so he/she wants to detect every bug in the software system. Software bug prediction (SBP) methods [1,2] are helpful to the developer to detect every possible fault-prone modules as early as possible. A software module is a manageable, deployable, composable, natively reusable, stateless unit of software that provides a concise interface to consumers, it is mainly chunk of software. For example dapper.net encapsulates database access. It has an API to access its functionality. It is a single file that can plugged in a source tree to be built. SBP builds models using mining software repositories (bug tracking, version control) and utilizes the models to recognize potential bug modules in new systems.

Knowing the modules of a software system that are more likely to be bug-prone before the testing phase can help in allocating testing resources optimally and efficiently. Detecting only buggy/non-buggy modules [3] leads to information loss about existing bugs in each module. It causes ineffective testing resource allocation; there should be more testing effort required in those modules which have more number of bugs. Software bug count prediction intends to solve such kind of problems, and its objective is to identify fault-prone modules by using software metrics [4] before starting the testing phase. The early estimating of the number of bugs in the modules can be beneficial in terms of effort required in each module.

Research on software bug count prediction [5,6] is recently growing faster due to the development of large and complex software systems. There are few recent articles [5,7] are published on a similar issue. Researchers and software practitioners have proposed many models and algorithms using various statistical methods and machine learning techniques. Chen et al. [5]

* Corresponding author.
E-mail addresses: sushantkp.rs.cse16@iitbhu.ac.in (S.K. Pandey), aktripathi.cse@iitbhu.ac.in (A.K. Tripathi).

https://doi.org/10.1016/j.knosys.2020.105924
0950-7051/© 2020 Published by Elsevier B.V.

compared the performance of the supervised and unsupervised method of defect count prediction. Similarly, Rathore et al. [6] applied linear and non-linear heterogeneous ensemble learning techniques to predict a number of faults in the software system. They only considered module-level bug count prediction. None of them considered the bug count vector prediction over the complete software system for the upcoming version. The software project's dataset consists of many instances, each instances represents a specific module of software project. Every column vector of these datasets represents various software metrics such a line of code (LOC), Number of children (NOC) etc. The last column of the dataset is a column vector, it is also known as bug count vector. The bug count vector gives information about bug value in each software module, it represents as y_i. Let x_{ij} represents item in the dataset, then i indicates the module and j represents software metric. The y_i is the value of bug for ith instance or module. For example, the project ant-1.3 of PROMISE repository [8] has 125 modules; the last column is a bug information vector also called as bug count vector, it gives bug information of each of the 125 modules.

Deep learning architectures are capable to solve most of the real life problem. Hand writing recognition [9], Finger print recognition [10] many more. Recently deep learning architecture were applied over feature extraction [11] in SBP that outperforms over baselines methods. Deep learning methods are so powerful for future sequence prediction after training using current and past sequences. It also can generate a predicted sequence. Till now, deep learning is not applied over bug count prediction. Our empirical study suggests that deep learning can also apply to bug count vector prediction and can be effective over the baseline technique. The two main reasons for the concatenation of different versions of the same software are, first, the software features are the same as all previous versions in our experimental data. Second, it also increases the data size for model training. After the concatenation of all versions of the same software system makes the training more robust and effective, the detailed discussion will be done in later sections.

The bug count vector (BCV) is a column vector that consists of the information about bug number in every module, and the BCV-Predictor predicts that a column vector. In this article, we have formalized a problem and its novel solution. We are anticipating the bugs count vector of the next version of the software system using information about its all previous versions. We have collected the information about software metrics that are used, bug details of each module, several versions of software, and created a meta-information repository by concatenation of the different versions of the same software called meta-datasets. Meta-datasets feed into the training of the proposed approach. The early detection of the number of a bug in the modules can be beneficial in terms allocating testing resources optimally and efficiently in each module.

The objective of this article is to build a BCV-Predictor that can predict bugs in each module of the upcoming version of a software system, and it reduces the testing effort. We have conducted our experiments over seven open-source software system collected from PROMISE data repository [8], 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. The softwares nowadays are becoming more sophisticated and bulky. The state of the art methods are mostly based on classical machine learning (ML) based technique. Large scale software system that consists of an extensive number of software modules such as prop from PROMISE repository, the 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 over such challenges. To the extent of our knowledge, for software bug count prediction, the use of deep learning approaches for bug count has not been explored. Moreover, the use of DL based methods for the cross-version defect prediction [12] in the next version software modules remained unexplored. Earlier, DL based approaches applied to software bug prediction [11], here they were used to classify given dependent variable into the 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 classification buggy or non-buggy module 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 [1]. It helps the software developer to prioritize the developing resources based on the number of bugs information. It also helps him/her to locate as much as bugs as early and quickly. However, very limited work has been done in such aspects. According to our empirical study, the following are the research gaps that should be considered.

(i) Predicting the BCV in the upcoming version of a software system will alert the developer to examine the probable fault-prone modules and predicting number of bugs, before the testing phase minimizes the enormous testing effort for different modules. Very limited works were done in such aspects.
(ii) Information regarding all prior versions of the software project will be helpful to build better BCV estimator for the upcoming version of the same software. Considering the BCV prediction as a regression problem has still limited research.
(iii) Our empirical study reported that classical ML-based bug count prediction models are inefficient over large software systems, and it also takes high computational cost.
(iv) Deep learning architecture is capable of solving many real-life problems, recently few deep learning-based SBP models [2] surpass the performance over classical ML-based models. So, it can also be useful in BCV prediction.

After our empirical studies, we found deep learning-based models can be very efficient in terms of performance and computational cost over software BCV prediction, a few of the baseline methods [13,14] have high computational costs over an extensive software system. The article makes the following contributions.

1. Regression problem identification and the objective function to predict the BCV of the upcoming version of software systems.
2. We develop a novel architecture named BCV-Predictor that effectively predicts the BCV for the upcoming version of the software system, and we compared the performance with 11 baseline methods. It also addresses CIB and overfitting problem. To the best of our knowledge, the first deep learning architecture based model in software bug count prediction.
3. We performed extensive experiments on a total of 31 different versions of 7 software projects. Metadata collection process using these projects, a total of seven metadata projects formation.
4. Sharing all the sources code along with meta-datasets over GitHub at [15]

We have framed three research queries (RQ) to justify the performance of BCV-Predictor; we will address these RQs are Section 4. The RQ is given below.
RQ-1: How much the BCV-Predictor is effective over all seven metadata?

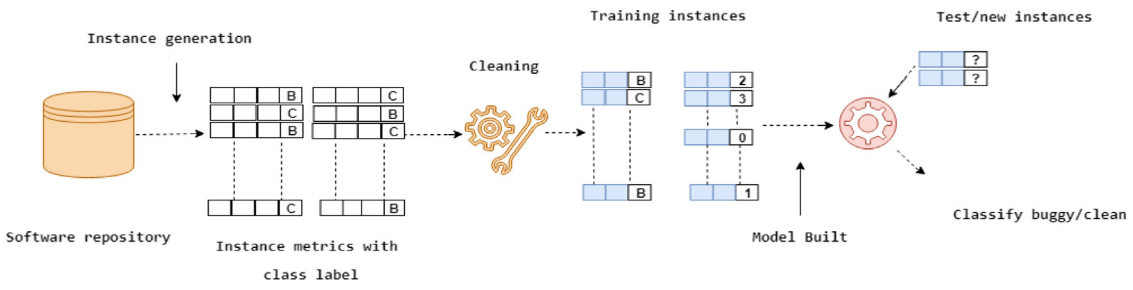


Fig. 1. Basic architecture of software bug count prediction model.

Table 1
Dataset description.

Project	version	No. of modules	No. of buggy modules	% Buggy modules	Max no. of bugs
ant	ant-1.3	125	33	16.00%	3
	ant-1.4	178	47	22.47%	3
	ant-1.5	293	35	10.92%	2
	ant-1.6	351	92	26.21%	10
	ant-1.7	745	166	22.28	10
camel	camel-1.0	339	14	3.83%	2
	camel-1.2	608	216	35.53%	28
	camel-1.4	872	146	16.63	17
	camel-1.6	965	118	19.48%	28
jedit	jedit-3.2	272	90	31.01%	45
	jedit-4.0	306	75	24.51%	23
	jedit-4.1	312	78	25.32%	23
	jedit-4.2	367	48	13.08%	10
prop	prop-v1	18471	2731	14.78%	37
	prop-v2	23014	2425	10.53%	27
	prop-v3	10275	1181	11.49%	11
	prop-v4	8718	839	9.62%	22
	prop-v5	8416	1298	15.42%	19
	prop-v6	660	66	10.00%	4
xerces	xerces-1.2	440	115	16.14%	4
	xerces-1.3	453	68	15.23%	30
	xerces-1.4	588	429	72.95%	62
	xerces-init	163	78	47.85%	11
xalan	xalan-2.4	723	156	15.21%	7
	xalan-2.5	803	531	48.19%	9
	xalan-2.6	885	625	46.44%	9
	xalan-2.7	909	898	98.78%	8
poi	poi-1.5	237	139	58.64%	20
	poi-2.0	314	37	11.78%	2
	poi-2.5	385	248	64.44%	11
	poi-3.0	442	281	63.57%	19

RQ-2: Comparison of evaluation matrices of BCV-Predictor with baseline methodologies.
RQ-3: Training time comparison of the BCV-Predictor with all 11 baseline techniques.

The further structure of the article is in such a manner that in the next section we will explain the background information and problem definition. Then we will illustrate the proposed approach in Section 3 after that we discuss the results in Section 4. Finally, we have discussed the related work in Section 5.

2. Background and problem statement

In this section, we will provide premises information about SBP and several bug prediction approaches. After that, we will explain the problem statement and objective function of the proposed work.

2.1. Software bug prediction

Software fault is a condition that may cause the software to fail to perform its required function, or it is a discrepancy between the actual and expected result that causes a failure. The bug is the terminology used by the tester to indicate fault. Software bug prediction is an assisting activity of the software testing phase, and it detects the modules that are fault-prone and requires extensive testing effort. Predicting the number of bugs in the software system is one step ahead of SBP. Fig. 1 shows the architecture of the software bug counts the prediction model. We can see in Fig. 1 that instances are generated from 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 which have buggy label replace with a number of bugs, and clean label replaces with 0. Then data is ready to train the machine learning [1] or statistical learning [16] based predictive model; the new instance with unknown labels is used to test the predictive model.

2.2. Problem statement

Our empirical study investigates to predict the bug count vector in the upcoming version of the software system. The number

Table 2
Meta-datasets description.

Project	No. of version	No. of modules	No. of buggy modules	% of buggy modules	Max no. of bugs max($bugs_S$)
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

Table 3
Metrics description.

Metric class	Metric name	Explanation
Abstraction	DIT	Depth of inheritance tree
	NOC	Number of children
	MFA	Measure of functional abstraction
Cohesion	LCOM	Lack of cohesion methods
	LCOM3	Lack of cohesion in methods
	CAM	Cohesion among methods of class
Coupling	CBO	Coupling between object classes
	RFC	Response for class
	CA	Afferent couplings
	CE	Efferent couplings
	IC	Inheritance coupling
Complexity	LOC	Line of codes
	WMC	Weighted methods per classes
	NPM	Number of public methods
	AMC	Average methods complexity
	Max_cc	Max McCabe's cyclomatic complexity
	Avg_cc	Average McCabe's cyclomatic complexity
	MOA	Measure of aggregation
Encapsulation	DAM	Data access metric

of software features are same in all versions of software. The problem formulation are 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.

$$S_i = (M_1, M_2, M_3........, M_j) \tag{1}$$

$$S_{n+1} = (M_1, M_2, M_3........, M_k) \tag{2}$$

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($bugs_S$). 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($bugs_S$) and predicting it minimizes the testing effort.

2.3. Dataset description

We have used seven open-source projects from PROMISE data repository [8] as shown in Table 1. Table 1 also reports that some of the data are imbalance, i.e., highly skewed towards non-buggy instances. Wang et al. [17] suggest sampling techniques on defects datasets to avoid imbalance issues. We have used random oversampling [18] techniques to circumvent this problem. We have built metadata using a different version of the software system. Table 2 shows the metadata description of datasets, we can see 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 lets say x, and that x is very few in comparison of all instances, which causes imbalance distribution of class x.

2.4. Software metrics

Software metrics are designed based on the code complexity, and features of the object-oriented program, Table 3 includes the metrics related to all seven software systems. Software systems have five different metrics classes, which are Abstraction, Cohesion, Coupling, Complexity, and Encapsulation. A total of nineteen software metrics, all these metrics have considered for the experiments.

2.5. 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 2.3. The overall collection of the metadata process is shown in Fig. 2. Metadata is a mainly 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 given in Table 2.3. 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. 2.

$$Meta(ant_X) = ant(X_{1.3}) * ant(X_{1.4}) * .... * ant(X_{1.7}) \tag{3}$$

$$Meta(ant_Y) = ant(Y_{1.3}) * ant(Y_{1.4}) * .... * ant(Y_{1.7}) \tag{4}$$

$$Meta(ant) = Meta(ant_X) + Meta(ant_Y) \tag{5}$$

Ant($X_{1.3}$), ant($X_{1.4}$), …, ant($X_{1.7}$) are various feature matrix of all five versions of ant. Whereas $bug_{1.3}$, $bug_{1.4}$, …, $bug_{1.7}$ are bug count vector of all five versions of the ant project. The feature matrix and bug count vector of metadata ant project represented as Meta(ant_X) and Meta(ant_Y) respectively as shown in Eqs. (3) & (4). We are just concatenating various feature matrix and bug count vector sequentially (initial to final version) and separately. Finally, accumulation of the feature matrix and bug count vector devise a metadata of an ant software, as shown in Eq. (5).

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 a 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 of the 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 the 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 1. Fig. 4 shows the graphical abstract of our proposed work. The data instances are generated from software repository; after data cleaning [19], the metadata generate from all prior version of software as discussed in Fig. 2 then data is ready for preprocessing according to the model.

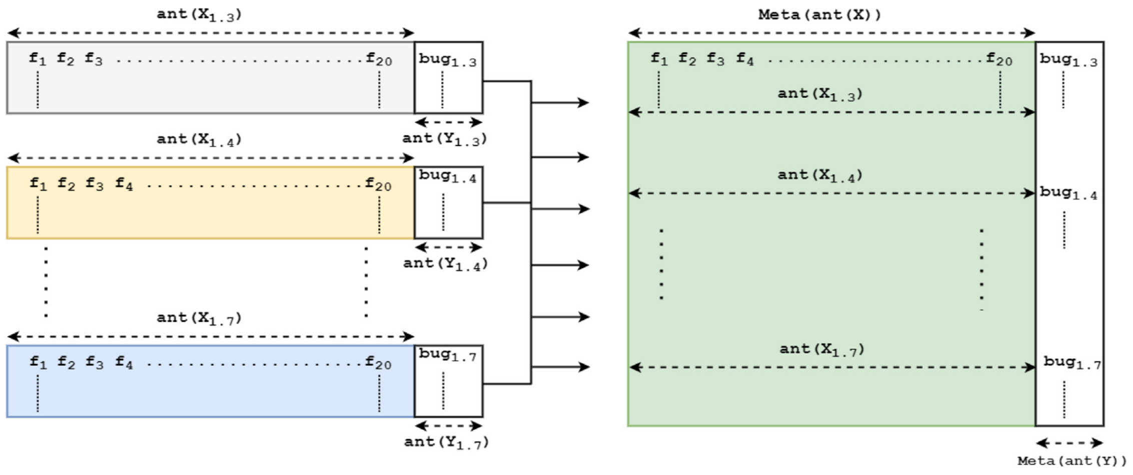


Fig. 2. Meta-data collection process.

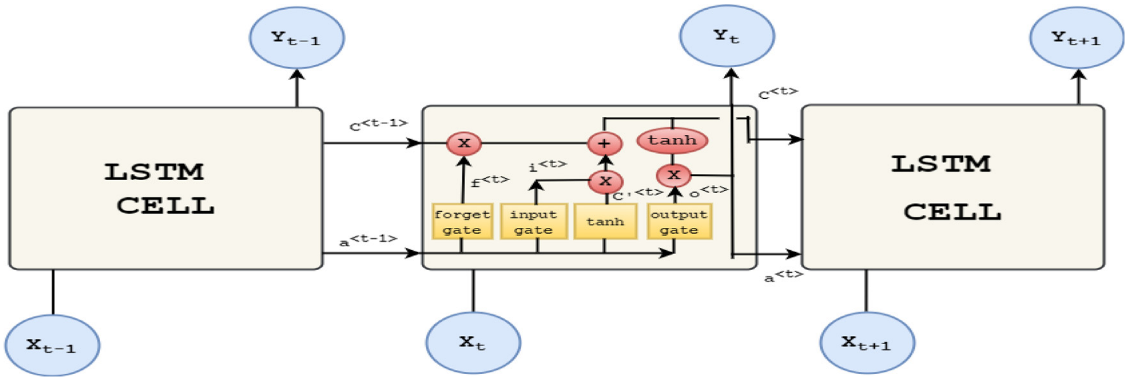


Fig. 3. Basic architecture of LSTM network, and LSTM cell.

3.1. Preprocessing

We have used seven public PROMISE datasets, as shown in Table 1, and then we generated metadata after concatenation all versions of the same software system (shown in Table 2) as discussed in Section 2.5. 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 [20] 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. (6). The preprocessing step is visible in the graphical abstract of the proposed work as shown in Fig. 4.

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{6}$$

As we have discussed in Section 2.3, the meta-datasets have a different bug label, and it has a skewed distribution, which causes class imbalance problems [21]. Many researchers have suggested a solution to this issue. Oversampling and under-sampling of datasets [22] will avoid this challenge. We have applied a random over-sampling method [18] for meta-datasets.

Algorithm 1 is multi-label random oversampling [23] that we have used in a preprocessing step. The dataset and the percentage of imbalance of a class p are as an input. Mean Imbalance Ratio (MIR) and Imbalance Ratio per Label (IRL) calculated during the cloning of minority labels. Algorithm 1 clone the minority class according to their IRL and discard the label, which has high IRL; complete explanation of this algorithm is given by Charte et al. [23].

3.2. Deep learning architecture

Recurrent Neural Networks (RNN) [24] are a type of neural network where the output from the previous step are fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like 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 find challenging when the sequence is large. It also suffers from a vanish gradient problem [25] and exploding gradient problem [26]. Long short term memory (LSTM) [27] overcomes from 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 3. LSTM cell consists of three gates, "forget gate", "input gate", and "output gate" as given in Fig. 3. 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 below.

(i) The first step of LSTM is to decide which information needs to be flush away from cell state. This determine by sigmoid

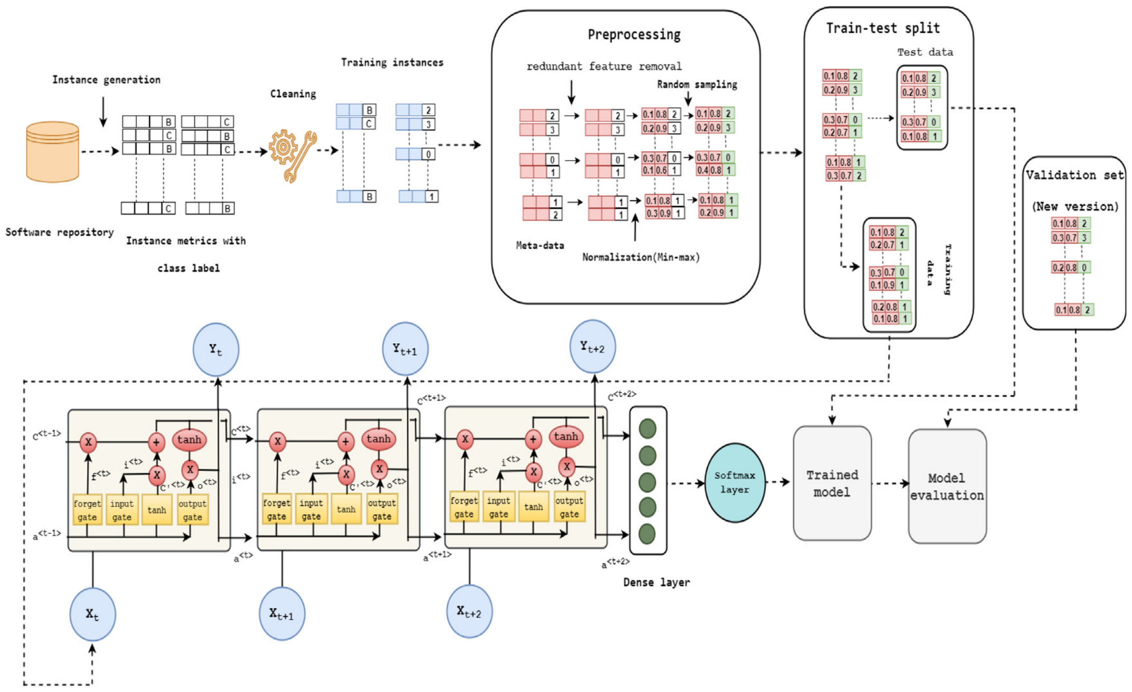


Fig. 4. Graphical abstract of proposed work.

function of forget gate (f^t). As it takes new input sequences from x_t and previous cell output sequence i.e., from $a^{\langle t-1\rangle}$.

$$f^t = \sigma(w_f[a^{t-1}, x^t] + b_f) \tag{7}$$

(ii) The next step decides which information needs to be stored in the cell state. It has two segments; first sigmoid layer decides which values will need to update, second tanh layer create a vector for the new candidates values, $C'^{\langle t\rangle}$ can be added to the state. After that, it combines these two and update ($i^{\langle t\rangle}$) the state.

$$i^t = \sigma(w_i[a^{t-1}, x^t] + b_i) \tag{8}$$

$$C'^{\langle t\rangle} = tanh(w_c[a^{\langle t-1\rangle}, x^t] + b_c) \tag{9}$$

(iii) After that, it update the old cell state, $C^{\langle t-1\rangle}$, into the new cell state $C^{\langle t\rangle}$.

$$C^{\langle t\rangle} = i^t * C'^{\langle t\rangle} + f^t * C^{\langle t-1\rangle} \tag{10}$$

(iv) In the final step, it decides what output needs to be provided. It first runs the sigmoid layer that decides which output will be provided, and then it pulls a cell state to tanh. It triggers that output sequence, which had agreed to be output.

$$o^t = \sigma(w_o[a^{t-1}, x^t] + b_o) \tag{11}$$

$$a^{\langle t\rangle} = o^t * tanh * C^{\langle t\rangle} \tag{12}$$

3.3. Proposed model

Long short term memory (LSTM) has an 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 are feed into to the LSTM network for training. We set the time step required in 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 2 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), $d_{M'} = d_1, d_2, \ldots, d_n$ as discussed in Fig. 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 3.1 section. As we have discussed earlier, the meta-datasets 2 are imbalanced, so we have applied the random over-sampling method (see Algo. 1) in step 11 over feature vector X. The main objective to apply sampling method is to avoid class imbalance problem as we have discussed in Section 3.1.

After preprocessing the metadata split 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. The Fig. 4 shows the train-test split step. Then training data (X → $train_x$, & Y→ $train_y$) feed into 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 [28] 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 network. 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.

$$\zeta_i^l = w_i^{l+1} * y^l + b_i^{l+1} \tag{13}$$

$$y_i^{l+1} = f(\zeta_i^{l+1}) \tag{14}$$

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

$$r_j^l \ Bernauli(p) \tag{15}$$

The thinned output then used as a input to next layer, this process repeats for every layer. All these processes are given in Eq. (13) to (18)

$$\hat{y} = r^l * y^l \tag{16}$$

$$\zeta_i^{l+1} = w_i^{l+1} * \hat{y}^l + b_i^{l+1} \tag{17}$$

$$y_i^{l+1} = f(\zeta_i^{l+1}) \tag{18}$$

Now for epoch 1 to I, we have set *BatchSize* (batch size) = ϵ, where batch size [29] 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 $\epsilon = 128$. Each LSTM layer computes $C'^{\langle t \rangle}$, i^t, f^t, $o^{\langle t \rangle}$, $C^{\langle t \rangle}$, $a^{\langle t \rangle}$ (as discussed in Section 3.2).

$$\sigma(z_j) = \frac{e^{(z_i)}}{\sum_{k=1}^{K} e^{z_k}} \tag{19}$$

Algorithm 1: 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
```

We have used Adam optimizer [30] to update network weights iterative based in training data. It combined both heuristic optimization method RMSProp [31] and Momentum [32] impedes search in direction of oscillations.

$$v_t = \beta_1 * v_{t-1} - (1 - \beta_1) * g_t \tag{20}$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2 \tag{21}$$

$$\Delta_{\omega_t} = -\eta \frac{v_t}{\sqrt{s_t + \hat{\epsilon}}} * g_t \tag{22}$$

$$\omega_{t+1} = \omega_t + \Delta_{\omega_t} \tag{23}$$

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. (19)) validates the computed probability score of the trained set with the validation set.

We have utilized Sparse categorical cross-entropy loss function, as shown in Eq. (24) to calculate loss. When the target label not a one-hot vector, then this loss function is more preferable [33].

$$L(\hat{Y}, Y) = -\frac{1}{M'} \sum_{i=1}^{M'} [y_i \log \hat{y} + (1 - y_i) \log(1 - \hat{y}_i)] \tag{24}$$

y_i is the class label of ith module in 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 3.5.

3.4. Experimental arrangements

We have performed our experiments over the NVIDIA GPU server of 16 GB 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. First, second, and third hidden layers have 100, 80, and 60 hidden units, respectively. Whereas 0.2 dropout rate in all three LSTM layers. In most of the meta-datasets, we have used 128 batch size. All the experiments were conducted over 450 to 500 epochs. The time step is the same as the number of versions of the software, validation split is 0.2, and the rest of the parameters are set as default value.

3.5. Performance metrics

To evaluate the performance of the BCV-Predictor, we have used four performance measures, the brief illustration of all performance matrices are given below.

Mean Absolute Error (MAE): It measures the mean magnitude of the error in a prediction set, without considering its direction of prediction. It is the mean over the testing sample of the absolute difference between the actual observation. Here every individual difference has equal weights. We have also considered MAE over validation set as Val_{mae}.

$$MAE = \frac{1}{n} \sum_{i=1}^{n} | y_i - \hat{y}_i | \tag{25}$$

Mean Squared Error (MSE): It is the overall sum of the data points, and the square of the difference between the predicted and actual target variables, divided by the number of data points.

Algorithm 2: 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 integer
4  end
5  $S_{n+1}$ version of S also contains f features.
6  $d_{M'}$ ← data instance of S // meta-dataset 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 feature of X, using Eq. (6)
11 Sampling$_{random}$(X,Y)    // random oversampling, using Algo. 1
12 X→ train$_x$, test$_x$ & Y→ train$_y$, test$_y$  // train test split
13 sequential$_{LSTM}$(l, train$_x$, train$_y$,$\Delta$ = c, time-step = n)
   // training set feeded to LSTM with l hidden units and fix dropout rate
14 foreach epoch I, BatchSize = $\epsilon$ do
15     for each sequential$_{LSTM}$ layer do
16         $C'^{<t>}$ $u^t$ , $f^t$, $o^{<t>}$, $C^{<t>}$, $a^{<t>}$    // from Eq. (9) to (12)
17         $\kappa_q$ = $P_{score}$(q) of $d_{M'}$    // score of each bug q
18     end
19     Optimize$_{adam}$($\hat{Y}$, Y) // update weight from Eq. (20) to (23)
20     Compute $\sigma(z_j) = \frac{e^{(z_i)}}{\sum_{k=1}^{K} e^{z_k}}$ for each hidden unit
       // compute softmax function
21     Compare($\kappa_q$, $\kappa_Q$) // compared over validation set
22     Compute $L(\hat{Y}, Y) = -\frac{1}{M'}\sum_{i=1}^{M'}[y_i \log \hat{y} + (1-y_i)\log(1-\hat{y}_i)]$
       // Loss calculate
23     Calculate MSE(y, $\hat{y}$), MAE(y, $\hat{y}$), Accuracy(y, $\hat{y}$).
24 end
25 Evaluate M using MAE(Y, $\hat{Y}$), MSE(Y, $\hat{Y}$), Accuracy((Y, $\hat{Y}$)).
   // model evaluation after I epoch
```

We have measured overall MSE and MSE over validation set denoted as Val$_{mse}$.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{26}$$

Accuracy: It is the fraction of corrected predictions over total predictions. If more is the accuracy more accurately the model is predicting. Its value lies between 0 to 1, where 1 indicates all the correct prediction, 0 means no correct prediction, and 0.5 means random predictions occur. We have considered the overall accuracy and accuracy over validation set as denoted by val$_{acc}$.

$$Accuracy = \frac{NumberofCorrectedPredictions}{TotalNumberofPredictions}. \tag{27}$$

4. Results

In this section, we will report the results of our experiment over various performance metrics (discussed in Section 3.5). Apart from these, we will also address the research queries that we have framed in Section 1. We will also explain the insightful discussion about our work.

Table 4
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

4.1. 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 performance of the model using MSE, MAE, & accuracy of over the complete model and also over the validation set. Table 4 represents the MSE, MAE, Accuracy, Val$_{mse}$, Val$_{mae}$, and Val$_{accuracy}$ values produced by BCV-Predictor.

4.1.1. Justification of RQ-1

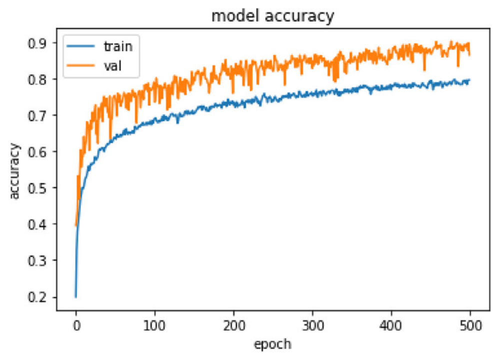
The MSE and MAE of the complete model over all the meta-datasets are shown in the first and second columns of Table 4. The lowest MSE value is of prop, i.e., 0.71, followed by camel with 1.279 MSE value. After that poi, xalan, ant, jedit, and xerces have lower MSE with values 2.68, 2.751, 2.979, 3.33, 4.715 respectively. The increasing order of MAE values of the 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. 7b shows the overall MSE value of all metadata. Xerces has the highest pick, whereas prop has a lowermost pick in the bargraph. The rest of the projects have moderate hight in the bargraph represents moderate loss. Fig. 7a shows the bargraph of overall MAE value that produced by BCV-Predictor. The lowest pick is of prop project, whereas the highest pick is of xerces software projects indicates maximum loss. Jedit and xalan also have high MAE loss. Ant and xalan have almost approx equal MAE values.

As shown in Table 4, the accuracy of xerces is highest with 91.95%, then of 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 (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. The Fig. 7 shows the bargraph of MAE, MSE, and accuracy of BCV-Predictor. Fig. 7c shows the overall accuracy bargraph of all seven projects. Xerces has highest pick plot and poi has lowest pick plot.

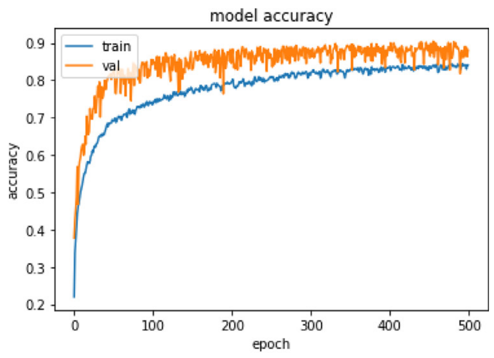
The MSE over validation set (Val$_{mse}$) is shown in fifth column of Table 4. The minimum Val$_{mse}$ is of prop metadata, which is 0.22. The increasing order of Val$_{mse}$ for rest of the meta-datasets i.e., xalan, ant, jedit, camel, poi, and xerces are 0.436, 0.450, 0.56, 0.886, 1.115, and 1.709 respectively.

The sixth column of Table 4 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.

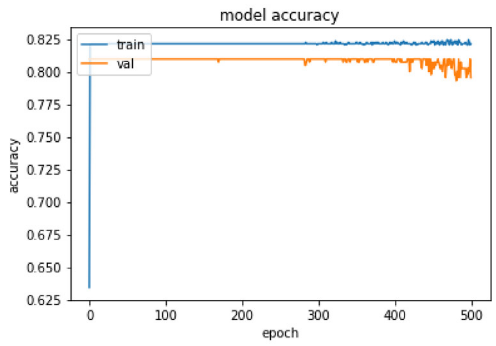
The accuracy over validation set (Val$_{accuracy}$) is shown in the last column of Table 4. Similar to overall accuracy, the validation set accuracy also follows the similar order and approximate equal values. The xerces metadata have maximum accuracy, i.e., 94.44%. The accuracy of camel, prop, ant, jedit, xalan, and poi have values 88.00%, 87.61%, 86.44%, 79.52%, 66.87%, and 49.85% respectively. The Fig. 7c shows the accuracy of BCV-Predictor over all meta-datasets.



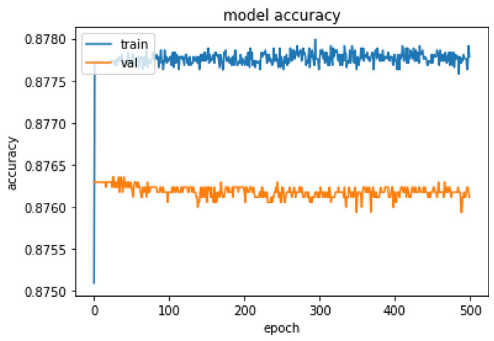
(a) Accuracy of ant metadata.



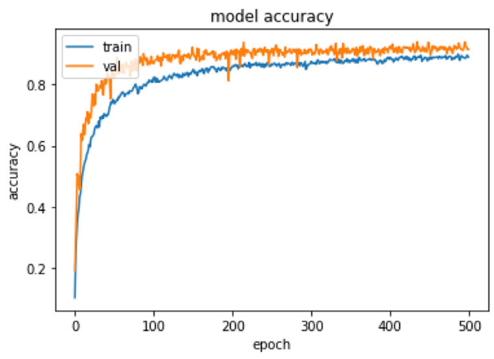
(b) Accuracy of camel metadata.



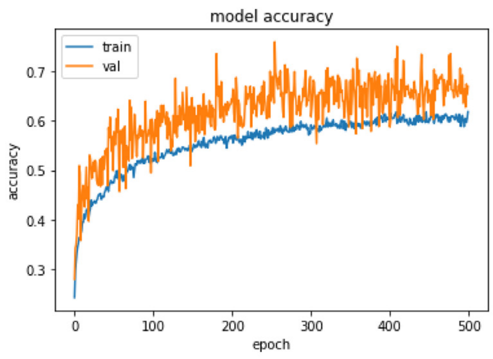
(c) Accuracy of jedit metadata.



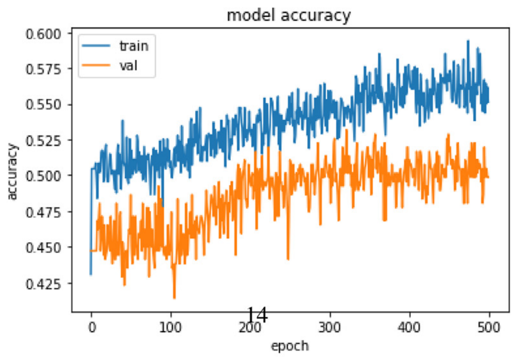
(d) Accuracy of prop metadata.



(e) Accuracy of xerces metadata.



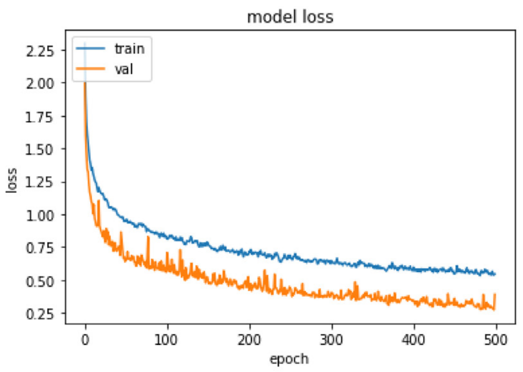
(f) Accuracy of xalan metadata.



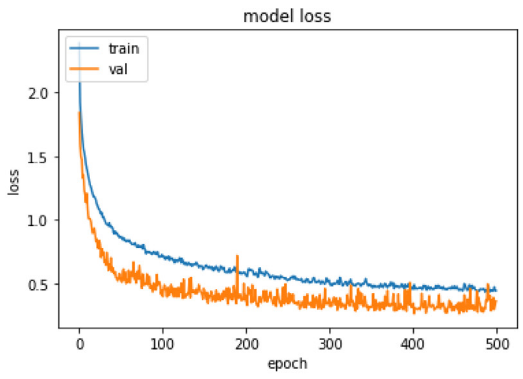
(g) Accuracy of poi metadata.

Fig. 5. Accuracy of all seven metadata. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

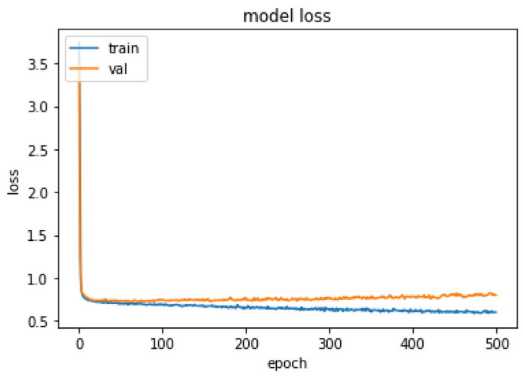
The accuracy over both train set and validation set of the proposed model over 500 epoch is shown in Fig. 5. The blue and orange lines in all sub-figures from Figs. 5a to 5g represents accuracy plot over the train set and validation set till 500 epoch. In Fig. 5a, the accuracy over validation set is approx 85%–87% whereas over 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. 5b, 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, model is stable over camel software. The Fig. 5c shows for jedit metadata, the accuracy is linear till 400 epoch, it is because of lesser data instances, the model is somehow stable over this project. as As prop metadata have sufficient data to train, so the



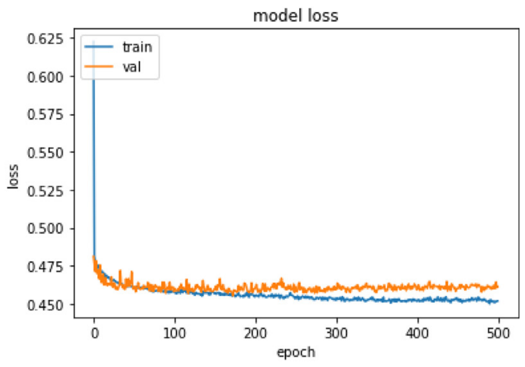
(a) Loss of ant metadata.



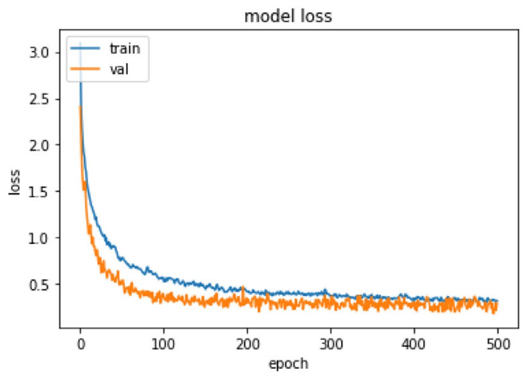
(b) Loss of camel metadata.



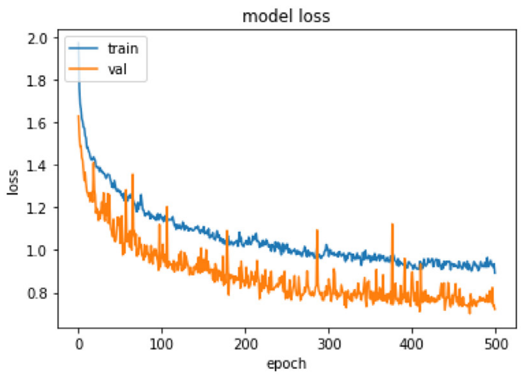
(c) Loss of jedit metadata.



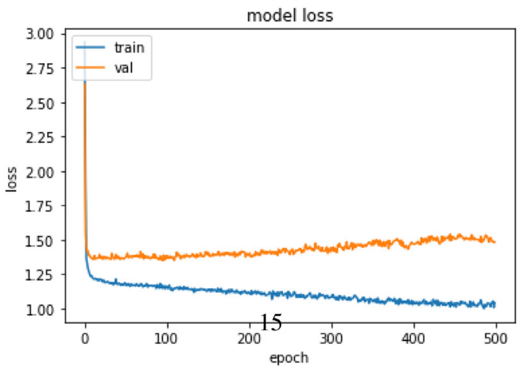
(d) Loss of prop metadata.



(e) Loss of xerces metadata.



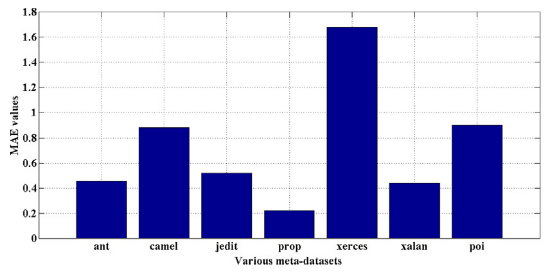
(f) Loss of xalan metadata.



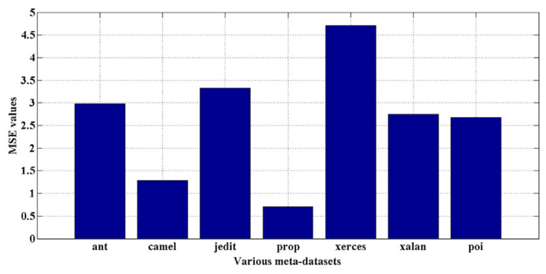
(g) Loss of poi metadata.

Fig. 6. Loss of all seven metadata. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

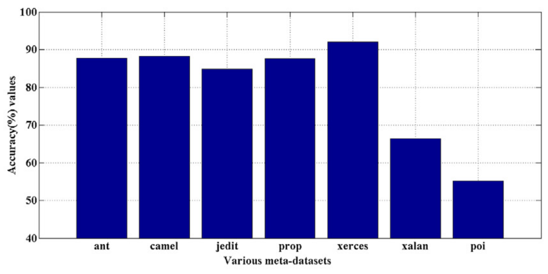
validation accuracy is from 0.8760 to 8465, whereas the train set accuracy is from 0.88775 to 0.8780. Suggested model is highly stable over prop software project. In Fig. 5e, we can see the validation and train set accuracy varying with the epoch of xerces metadata. Still, validation set accuracy is always more than train set accuracy and; the validation accuracy lies between 85% to



(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.

Fig. 7. Bar-graph of MAE, MSE, & accuracy over all meta-datasets.

91%. BCV-Predictor is stable over xerces, it has sufficient number of instances in training set. Fig. 5f 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, 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. 5g, 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-content data instances that lead to unstable model.

In Fig. 6, we have aggregated the plot between epoch and loss. Here blue and orange lines indicate loss over 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 converges or overlapping are good fitted curves and model circumvent overfitting problem. In Figs. 6a, 6b, 6e, & 6f 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 Figs. 6c, 6d, & 6g, 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. 6a), lies from 0.25 to 0.5 after 400 to 500 epoch. Validation loss over camel metadata (Fig. 6b) lies from 0.25 to 0.55 after 400 epoch. The validation loss for jedit not much 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 loss for prop metadata is overlapping from 0 to 220 epoch, as we can see in Fig. 6d, 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 epoch. More variation on validation loss xalan metadata, as shown in Fig. 6f, 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 epoch. The loss curve of ant, camel, prop, xalan, and xerces are good fitted curves, so they avoid overfitting problem. The loss curve of jedit is also diverging, as shown in Fig. 6c, but the difference is negligible, so it also lies under 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. 6g, so it is suffering from overfitting.

4.1.2. Baseline methods

We feed all the seven meta-datasets into eleven states of the art techniques and evaluate their performance. The performance of Linear Regression (LR), Multilayer Perceptron (MLP) [13], Instance-based learning (IBK) [34], Additive Regression (AR) [35], M5rules [36,37], M5P [36], Bagging [38], Gaussian Process (GP) [14], Decision Stump [39], Random Forrest [40], and Regression by Decentralization (RD) [41] are compared with BCV-Predictor shown in Table 5 to Table 7. 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 methods to compare the performance with BCV-Predictor. We used 100 batch sizes for all the methods. Poly kernel, noise as 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 default value of the WEKA tool [42] to the rest of the hyperparameters of all the eleven techniques.

4.1.3. Justification of RQ-2

Table 5 compares the MAE values of various techniques with BCV-Predictor. As shown in Table 5, four out of seven meta-datasets have lesser MAE value 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. Lowest MAE of ant, camel and xerces projects are processed by DS, IBK, and RF models respectively. Table 6 compares the MSE values of different models with our proposed approach. The MSE produced by BCV-Predictor camel, jedit, prop, xerces, and poi meta-datasets is less compared with 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 7. Whereas RF surpasses the accuracy over BCV-Predictor for xalan and poi meta-datasets.

Table 5
MAE comparison of BCV-Predictor with other techniques.

Data	LR	MLP	IBK	AR	M5rules	M5P	Bagging	GP	DS	RF	RD	BCV-Predictor
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 6
MSE comparison of BCV-Predictor with other techniques.

Data	LR	MLP	IBK	AR	M5rules	M5P	Bagging	GP	DS	RF	RD	BCV-Predictor
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 7
Accuracy comparison of BCV-Predictor with other techniques.

Data	LR	MLP	IBK	AR	M5rules	M5P	Bagging	GP	DS	RF	RD	BCV-Predictor
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

4.2. Insightful discussion

We have conducted experiments over seven meta-datasets; few of them have not enough data instances for finer training purposes. As we can see in Table 2, 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 the model becomes healthier, and it leads to satisfactory results [43] with lesser chances of over-fitting. In Fig. 5c, we can see the validation set and train set accuracy are almost unvarying with respect to epoch after 300 epoch. Although small variation started; it is because of very lesser instances in the validation set. Similar effect is also visible in Fig. 6c, the validation loss somehow increases concerning the epoch. If BCV-Predictor has more training data, better accuracy will achieve.

As we observe in Table 4 and Fig. 5g, 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. 6g). Data imbalance [17] also plays a crucial role in such a scenario. Xalan also facing a similar problem, as we can see in Table 4, the accuracy of xalan is 66.41%, and MSE is 2.751. That indicates high loss during prediction and misses the correct bug number in bug count vector. Fig. 6f, 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 metrics as shown in Fig. 8 during experiments. We considered 500 epoch because the loss versus accuracy graph in Fig. 6 is almost stable after 300 epoch, before that loss is moderate for xalan and poi metadata. The Fig. 8a shows that excluding xalan all the other six metadata have almost no variation in MAE values, which indicates stable model. Whereas, some disparity is visible in xalan, which is acceptable. Similarly, only xalan has variation in MSE after 300 epoch, as shown in Fig. 8b, which makes lesser stable. In Fig. 8c, the range of accuracy is shown after 300 epoch, its easily visible that jedit, prop, and xerces, have tiny difference that implies high stability. Whereas xalan and poi have some variation in which specify slight 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.

List of factors that affect the performance of the BCV-Predictor are given below.

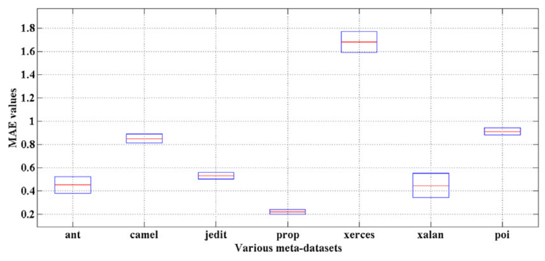
(i) Lack of training instances, which causes miss in accurate prediction of target bug number.
(ii) Skewed distribution of bugs in different module.
(iii) There is always a scope of hyperparameters tuning to enhance the performance of the deep learning model.

4.2.1. Justification of RQ-3

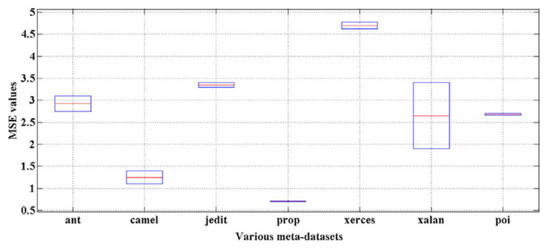
Table 8 listed the training time of BCV-Predictor and other eleven methodologies. Time is given in seconds, μ (micro) second, and millisecond (ms). The training time of BCV-Predictor is more from all the methods except GP and MLP. Prop has the most massive training set. The training time for prop is 17 s + 43 μs, which is very 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. Non-parametric test

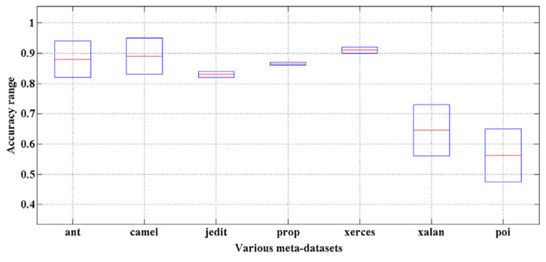
We performed the Wilcoxon Signed-Rank Test [44], a non-parametric test, to testify the significant results of the actual bug versus the predicted bug in the module by BCV-Predictor.



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



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



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

Fig. 8. Boxplot of MAE, MSE, & accuracy over all meta-datasets after 300 epoch.

Table 8
Training time (second) of various models compared with BCV-Predictor.

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

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 has n different version, so we created metadata using a concatenation of n − 1 version. Feed that metadata into BCV-Predictor for training and to predict the bug count vector of nth version. Then compare the predicted bug count vector with existing nth version of S. In Table 9, we have aggregated the actual bugs (y) and predicted bugs ($\hat{y}$) of a different module of a software nth version. The non-parametric tests make no assumptions about the distribution of assess samples. The null hypothesis is that the median difference between the pairwise samples, i.e., actual and predicted number of bug, 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 null hypothesis is significant at $\alpha = 0.005$.

5. Related work

There has been enormous research that has been going on various software metrics and software defect prediction models to improve software quality [45,46], as well as to reduce testing effort. The linear regression-based model to predict the number of bugs in the software module was proposed by Grave et al. [47], they conducted experiment over telecommunication system having software change metrics. They concluded that module age, change made over the module, and age of the change metrics all together produces better bug prediction accuracy. Ostrand et al. [48] suggested a model using negative binomial regression (NBR) for the number of bugs prediction and bug density over a particular module. They have used two industrial software systems and LOC metrics to evaluate the model. They have concluded that NBR is significantly effective to predict the number of bug in prediction models.

Similar work is also reported [49], which enhances the accuracy of existing model by 20%. Janes et al. [50] and Liguo et al. [51] also utilized the NBR in a similar aspect. Liguo et al. [51] applied NBR over object-oriented metrics of large telecommunication systems. Janes et al. [50] compared the performance of NBR based model to logistic regression-based technique. Fagundes et al. [52] presented the study over the number of the bugs prediction model, the study performed over eight public NASA datasets, using a zero-inflated prediction technique. They have used the error rate as a performance metric. Afzal et al. [53] suggested a bug count model using genetic programming; they were carried out over three industrial datasets, using weakly fault count as an independent variable. Recently Ye et al. [54] also utilized genetic programming over bug count using the information of function decomposition of source code.

Over the past few years ensemble learning (EL) based model plays an important role in the number of bug count domains. Bagging, boosting, voting, and stacking was widely used in this era. Many researcher have suggested various models [55,56] to address the bug count problem. Misirl et al. [55] presented an EL method over SBP applying naive bayes, artificial neural networks, and voting techniques. They concluded that the results of EL based methods more effective over other techniques. Zhen et al. [57] performed an experiment over the NASA dataset using boosting algorithm to produce a better bug prediction method. The most recent work using EL on bug count done by Rathor

Table 9
Actual and predicted values of bugs in different samples.

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

et al. [6], they have applied linear and non-linear combination heterogeneous EL method over the PROMISE dataset. Chen et al. [5] illustrated the study over supervised and unsupervised learning methods; they performed experiments using these two categories of ML techniques and compared the results.

Deep learning recently been applied over software bug prediction, Wang et al. [58] proposed software reliability prediction using a recurrent neural network. Recently Majula et al. [59] proposed a deep learning-based model to predict fault in a software system. Pandey et al. [11] combinedly applied autoencoder and heterogeneous EL method for defect prediction over the NASA dataset and concluded better results over EL based models. Dam et al. [60] also proposed LSTM based model for fault prediction; they have also suggested more data instances will intensify the performance.

6. Conclusion and future work

Predicting the bug count vector of the upcoming version of the software system will reduce testing effort and cost of software development. It also helps the project manager to suitably allocate developers and tester 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 predict the bug count vector in the next version of software system. The meta-datasets used to train BCV-Predictor. LSTM deep learning architecture used as a learning algorithm in BCV-Predictor, and the number of versions as a time steps. 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 to avoid overfitting and class imbalance problem, 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 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 value, respectively, compared with other states 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 the future, we are thinking to extend our research in many ways. First, we will apply a similar model in cross-project bug count prediction on the next version of the software. This can be effective in a cross-project software system. Second we are thinking to add attention layer along with LSTM to enhance the performance. Third, we wish to estimate the testing effort for a module through the identification of the size of a bug. Integrating testing methodologies over on design metrics. Fourth, we also want to conduct experiments on other open sources software systems that have huge data instances and several versions. Fifth, hyperparameters tuning can lead to much better results. Sixth, we are also planning to use transfer learning over design metrics to predict number of bugs in successive version of software system. The organizations should avail their datasets for research purposes so that a much better prediction model can be made. We try to apply another sequence model over this problem to achieve more robust results.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Sushant Kumar Pandey: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Anil Kumar Tripathi:** Supervision.

References

[1] R. Malhotra, A systematic review of machine learning techniques for software fault prediction, Appl. Soft Comput. 27 (2015) 504–518.
[2] S.K. Pandey, R.B. Mishra, A.K. Triphathi, Software bug prediction prototype using Bayesian network classifier: A comprehensive model, Procedia Comput. Sci. 132 (2018) 1412–1421.
[3] W. Li, Z. Huang, Q. Li, Three-way decisions based software defect prediction, Knowl.-Based Syst. 91 (2016) 263–274.
[4] N.E. Fenton, M. Neil, A critique of software defect prediction models, IEEE Trans. Softw. Eng. 25 (5) (1999) 675–689.
[5] X. Chen, D. Zhang, Y. Zhao, Z. Cui, C. Ni, Software defect number prediction: Unsupervised vs supervised methods, Inf. Softw. Technol. 106 (2019) 161–181.
[6] S.S. Rathore, S. Kumar, Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems, Knowl.-Based Syst. 119 (2017) 232–256.
[7] D. Bowes, T. Hall, J. Petrić, Software defect prediction: do different classifiers find the same defects? Softw. Qual. J. 26 (2) (2018) 525–552.
[8] J. Sayyad Shirabad, T. Menzies, The PROMISE Repository of Software Engineering Databases, School of Information Technology and Engineering, University of Ottawa, Canada, 2005, URL: http://promise.site.uottawa.ca/SERepository.

[9] C. Wu, W. Fan, Y. He, J. Sun, S. Naoi, Handwritten character recognition by alternately trained relaxation convolutional neural network, in: 2014 14th International Conference on Frontiers in Handwriting Recognition, IEEE, 2014, pp. 291–296.

[10] D. Menotti, G. Chiachia, A. Pinto, W.R. Schwartz, H. Pedrini, A.X. Falcao, A. Rocha, Deep representations for iris, face, and fingerprint spoofing detection, IEEE Trans. Inf. Forensics Secur. 10 (4) (2015) 864–879.

[11] S.K. Pandey, R.B. Mishra, A.K. Tripathi, Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques, Expert Syst. Appl. 144 (2020) 113085.

[12] X. Yang, W. Wen, Ridge and lasso regression models for cross-version defect prediction, IEEE Trans. Reliab. 67 (3) (2018) 885–896.

[13] A. Mahaweerawat, P. Sophatsathit, C. Lursinsap, P. Musilek, Fault prediction in object-oriented software using neural network techniques, in: Advanced Virtual and Intelligent Computing Center (AVIC), Department of Mathematics, Faculty of Science, Chulalongkorn University, Bangkok, Thailand, 2004, pp. 1–8.

[14] A.J. Smola, P.L. Bartlett, Sparse greedy gaussian process regression, in: Advances in Neural Information Processing Systems, 2001, pp. 619–625.

[15] A. Sushant, Meta-datasest and source code, 2020, URL: https://github.com/sushantkumar007007/sushantkumar007007-gmail.com.

[16] K. Gao, T.M. Khoshgoftaar, A comprehensive empirical study of count models for software fault prediction, IEEE Trans. Reliab. 56 (2) (2007) 223–236.

[17] S. Wang, X. Yao, Using class imbalance learning for software defect prediction, IEEE Trans. Reliab. 62 (2) (2013) 434–443.

[18] L. Abdi, S. Hashemi, To combat multi-class imbalanced problems by means of over-sampling techniques, IEEE Trans. Knowl. Data Eng. 28 (1) (2015) 238–251.

[19] E. Rahm, H.H. Do, Data cleaning: Problems and current approaches, IEEE Data Eng. Bull. 23 (4) (2000) 3–13.

[20] L. Al Shalabi, Z. Shaaban, B. Kasasbeh, Data mining: A preprocessing engine, J. Comput. Sci. 2 (9) (2006) 735–739.

[21] N. Japkowicz, S. Stephen, The class imbalance problem: A systematic study, Intell. Data Anal. 6 (5) (2002) 429–449.

[22] S. Wang, X. Yao, Multiclass imbalance problems: Analysis and potential solutions, IEEE Trans. Syst. Man Cybern. B 42 (4) (2012) 1119–1130.

[23] F. Charte, A.J. Rivera, M.J. del Jesus, F. Herrera, Addressing imbalance in multilabel classification: Measures and random resampling algorithms, Neurocomputing 163 (2015) 3–16.

[24] M. Schuster, K.K. Paliwal, Bidirectional recurrent neural networks, IEEE Trans. Signal Process. 45 (11) (1997) 2673–2681.

[25] S. Hochreiter, The vanishing gradient problem during learning recurrent neural nets and problem solutions, Int. J. Uncertain. Fuzziness Knowl.-Based Syst. 6 (02) (1998) 107–116.

[26] R. Pascanu, T. Mikolov, Y. Bengio, Understanding the exploding gradient problem, 2012, p. 2, CoRR, arXiv:abs/1211.5063.

[27] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (1) (2014) 1929–1958.

[29] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015, arXiv preprint arXiv:1502.03167.

[30] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014, arXiv preprint arXiv:1412.6980.

[31] S. Ruder, An overview of gradient descent optimization algorithms, 2016, arXiv preprint arXiv:1609.04747.

[32] I. Sutskever, J. Martens, G. Dahl, G. Hinton, On the importance of initialization and momentum in deep learning, in: International Conference on Machine Learning, 2013, pp. 1139–1147.

[33] P. Covington, J. Adams, E. Sargin, Deep neural networks for youtube recommendations, in: Proceedings of the 10th ACM Conference on Recommender Systems, ACM, 2016, pp. 191–198.

[34] D.W. Aha, D. Kibler, M.K. Albert, Instance-based learning algorithms, Mach. Learn. 6 (1) (1991) 37–66.

[35] E.J. Weyuker, T.J. Ostrand, R.M. Bell, Comparing the effectiveness of several modeling methods for fault prediction, Empir. Softw. Eng. 15 (3) (2010) 277–295.

[36] S.S. Rathore, S. Kumar, A decision tree regression based approach for the number of software faults prediction, ACM SIGSOFT Softw. Eng. Notes 41 (1) (2016) 1–6.

[37] H. Duggal, P. Singh, Comparative Study of the Performance of M5-Rules Algorithm with Different Algorithms, Scientific Research Publishing, 2012.

[38] Z. Sun, Q. Song, X. Zhu, Using coding-based ensemble learning to improve software defect prediction, IEEE Trans. Syst., Man, Cybern., Part C 42 (6) (2012) 1806–1817.

[39] W. Iba, P. Langley, Induction of one-level decision trees, in: Machine Learning Proceedings 1992, Elsevier, 1992, pp. 233–240.

[40] C. Catal, Software fault prediction: A literature review and current trends, Expert Syst. Appl. 38 (4) (2011) 4626–4636.

[41] E. Frank, R.R. Bouckaert, Conditional density estimation with class probability estimators, in: Asian Conference on Machine Learning, Springer, 2009, pp. 65–81.

[42] S.R. Garner, et al., Weka: The waikato environment for knowledge analysis, in: Proceedings of the New Zealand Computer Science Research Students Conference, 1995, pp. 57–64.

[43] X.-W. Chen, X. Lin, Big data deep learning: challenges and perspectives, IEEE Access 2 (2014) 514–525.

[44] R. Woolson, Wilcoxon signed-rank test, Wiley Online Library, 2007, pp. 1–3.

[45] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Trans. Softw. Eng. 34 (4) (2008) 485–496.

[46] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 181–190.

[47] T.L. Graves, A.F. Karr, J.S. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Trans. Softw. Eng. 26 (7) (2000) 653–661.

[48] T.J. Ostrand, E.J. Weyuker, R.M. Bell, Predicting the location and number of faults in large software systems, IEEE Trans. Softw. Eng. 31 (4) (2005) 340–355.

[49] R.M. Bell, T.J. Ostrand, E.J. Weyuker, Looking for bugs in all the right places, in: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ACM, 2006, pp. 61–72.

[50] A. Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, G. Succi, Identification of defect-prone classes in telecommunication software systems using design metrics, Inf. Sci. 176 (24) (2006) 3711–3734.

[51] L. Yu, Using Negative Binomial Regression Analysis to Predict Software Faults: a Study of Apache Ant, IJ Information Technology and Computer Science, 2012.

[52] R.A. Fagundes, R.M. Souza, F.J. Cysneiros, Zero-inflated prediction model in software-fault data, IET Softw. 10 (1) (2016) 1–9.

[53] W. Afzal, R. Torkar, R. Feldt, Prediction of fault count data using genetic programming, in: 2008 IEEE International Multitopic Conference, IEEE, 2008, pp. 349–356.

[54] X. Ye, R. Bunescu, C. Liu, Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation, IEEE Trans. Softw. Eng. 42 (4) (2015) 379–402.

[55] A.T. Mısırlı, A.B. Bener, B. Turhan, An industrial case study of classifier ensembles for locating software defects, Softw. Qual. J. 19 (3) (2011) 515–536.

[56] T. Wang, W. Li, H. Shi, Z. Liu, Software defect prediction based on classifiers ensemble, J. Inf. Comput. Sci. 8 (16) (2011) 4241–4254.

[57] J. Zheng, Cost-sensitive boosting neural networks for software defect prediction, Expert Syst. Appl. 37 (6) (2010) 4537–4543.

[58] J. Wang, C. Zhang, Software reliability prediction using a deep learning model based on the rnn encoder–decoder, Reliab. Eng. Syst. Saf. 170 (2018) 73–82.

[59] C. Manjula, L. Florence, Deep neural network based hybrid approach for software defect prediction using software metrics, Cluster Comput. (2018) 1–17.

[60] H.K. Dam, T. Pham, S.W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, C.-J. Kim, A deep tree-based model for software defect prediction, 2018, arXiv preprint arXiv:1802.00921.