Chapter 6

Predicting the next version of the Software System: A Hybrid Regression Analysis

6.1 Introduction

Cross-version defect prediction builds a methodology from the previous version of a software system to predict defects in the current version. Most of the articles over cross-version defect prediction model [281, 224, 279] only focus on bug prediction in the current version of a software system. On the other hand, our work predicts the metrics and the bug count for a new version of the software system, i.e., we are predicting data associated with the next version of a software project. Predicting the next version of any software system allows rapid addition of functionality from user response and resolution of platform compatibility issues of the previous versions. Prediction of metric values and the number of bugs in software modules helps to

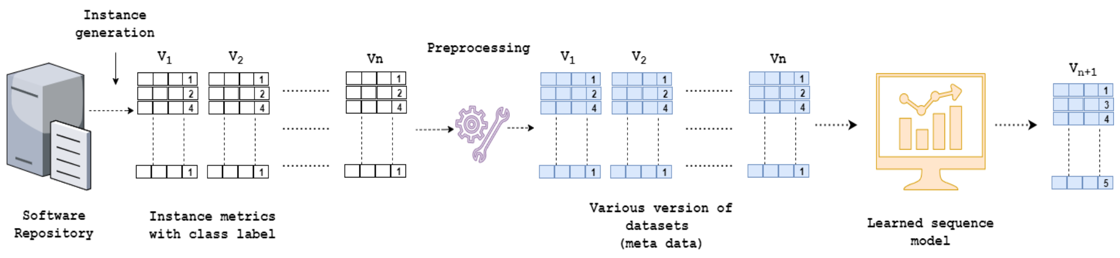


FIGURE 6.1: Structure of next version prediction of a software system.

develop high-quality software products at a low cost, further reducing the testing and development efforts of the software development life cycle. The motivation behind our work is given below:

(i) To date, no previous work on software bug prediction has focused on predicting the entire software (number of bugs along with software metrics in each module) of the successive version of a same software system.

(ii) The success of deep learning architectures in time series prediction and sequence prediction motivates their application for the next version prediction of software systems.

We performed the experiments on eight public software systems from the PROMISE data repository [215], using all the available versions to build our prediction model. The main contributions are summarized as follows:

(I) We propose a new methodology for predicting the entire successive version of a software system (metrics and bug count for each module). We shared all the obtained results and codes on Github for future analysis.

(II) We develop a novel architecture using a sequence to sequence model for predicting the bug count and normalized values of the software metrics in the corresponding software modules.

(III) We uniquely model the various versions of software systems in the time-series domain, utilizing the inherent relationship present across software modules in adjacent versions.

There are many conceptual differences between our work and the existing cross-version bug prediction work [286, 281]. Exiting work focuses on predicting bug or bug count or bug count vector of the next version of a software system. In contrast, we predict the entire software metrics and bug vector of each module of the upcoming version of a software project. The mathematical demonstration of both the work and their difference is shown in section 6.2.1.
No doubt, the prior prediction of each module detail (software metrics/bug count) will help the project leader to develop the next version more efficiently and with lesser cost.

We have framed three research queries (RQ), and the chapter also aims to find answers to these research queries (RQ). The research queries are given below:

RQ-1 *How is the proposed approach effective in predicting the bug count of every module in the next version of a software system?*

RQ-2 *How is the proposed approach effective in predicting the each software metric values of every module in the next version of a software system?*

RQ-3 *How much the proposed model is significantly effective over existing learning methods?*

This chapter is organized as follows. In the next section, we explain the underlying motivation and elaborate on the problem definition. In section 6.3, the premises is defined, while sections 6.4 and 6.5 present the proposed approach and results & discussion respectively. Finally, the threat to validity is discussed in sections 6.6.

6.2 Problem Identification

Let there be n versions in a software system. Our objective is to predict the data for the $(n+1)^{th}$ version. We employ a strategy to train an LSTM model using the first n-1 versions as the train data to predict the n^{th} version. Then, this trained model is used to predict the $(n+1)^{th}$ version given versions 2 to n. Fig. 6.1 shows the basic structure of the prediction approach for the next version of a software system. As shown in Fig. 6.1, the instance with class labels are created from a software repository, the data then is sent for preprocessing to remove noise, data duplication, etc. After preprocessing, the metadata is created using all software versions and then fed to the trained model for prediction of the next version of the same software.

6.2.1 Problem definition

Let there be n versions in some software system, say S, where n is a positive integer. $S = [S_1, S_2,....,S_n]$. Let the dataset corresponding to the i^{th} version be d_i, where each d_i has a m-dimensional feature vector, say f. So $d_i = [f_1, f_2,...., f_m, b_i]$, here b_i is the bug count vector of software S_i. Let metadata be $D_1 = [d_1, d_2,....., d_n]$. The

TABLE 6.1: Dataset description.

Software	**Version**	**No. of modules**	**No. of buggy modules**	**% of Buggy modules**	**Max bug count**
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	184	26.21%	10
	ant-1.7	745	338	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
log4j	log4j-1.0	135	25	18.5%	9
	log4j-1.1	109	37	33.9%	9
	lof4j-1.2	205	188	91.7%	10
lucene	lucene-2.0	195	91	46.6%	22
	lucene-2.2	247	144	58.2%	47
	lucene-2.4	340	200	58.5%	30
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
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
velocity	velocity-1.4	198	49	24.7%	7
	velocity-1.5	214	141	65.8%	10
	velocity-1.6	229	78	34.0%	12
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

existing cross-version defect prediction utilizes the D_1 for training and predict b_{n+1} of d_{n+1}, i.e., the bug count vector of the software S_{i+1}. Our work is significantly different, and our objective is to predict data associated with the next version of D_1, i.e. d_{n+1}, i.e., corresponding data to the next version of the software S_{n+1}. Thus d_{n+1} = [m'$_1$, m'$_2$, m'$_3$,...., m'$_l$, b_{n+1}], where m'$_i$ is the ith module and b_{n+1} is the bug count vector of the next version. The objective function of the proposed work is given in equation 6.1.

$$\kappa(D_1) = d_{n+1} \tag{6.1}$$

Here κ is a non-linear function to predict d_{n+1}.

6.3 Premises

This section illustrates the necessary information of the methodologies (deep learning architecture), software systems (datasets & software metrics), and various performance measures that we have used in this work.

6.3.1 Deep Learning Architecture

We have applied Long Short Term Memory (LSTM) [94] as our deep learning architecture. We first create the meta-dataset D_1 using all the modules present across all the versions of the software system, from which the meta dataset D_2 is created, after removing all the duplicate versions. Sequence to sequence model [245] (Seq2Seq) (also known as encoder-decoder model), map fixed-length input (T_x) to a fixed-length output (T_y), where $T_x \neq T_y$. It consists of three sections- encoder, intermediate (encoder vector) and decoder. The underlying architecture of sequence to sequence model using the LSTM unit is shown in Fig. 2.4. The detailed discussion of LSTM and Seq2Seq is given in earlier section 2.3.2.

6.3.2 Software systems

We have used eight public software systems from the PROMISE data repository, and every software version has twenty metrics and one additional information on bug count. A detailed description of the software datasets used is given in Table 2.4. Table 2.2 listed the 19 metrics used for each software system. Seven are complexity metrics, five coupling, three cohesion, three abstraction, and one encapsulation metrics.

6.3.3 Performance Measures

We have employed MAE, MSE, and accuracy as the performance measure. Detailed discussion about these metrics is given in section 2.4.

Algorithm 7: Proposed algorithm

```
Input
Create a meta_dataset $D_1$ using $d_1$, $d_2$,...., $d_n$
$D_1$ has total p instances        ▷ Duplicates may be present for $i = 1$ to $p$ do
    Add Module $M_i$ to $D_2$ such that $M_i \subset d_1$ OR $d_2$ OR .... OR $d_n$.
end
                                   ▷ Removing duplicate instances
if $M_i$ already in $D_2$ then
    Discard $M_i$ else
    $D_2 \leftarrow M_i$
end
                                   ▷ Extracting out the common modules
for $i = 1$ to $p$ do
    if Module $M_i \subset d_1$, AND $M_i \subset d_2$, AND $M_i \subset d_3$,..., AND $M_i \subset d_n$. then
        $D_3 \leftarrow M_i$
    end
end
$D_4$ = Get_Test_Data(M,$D_2$)              ▷ Goto Algorithm 8
$D_5$ = Data_augmentation($D_3$, $D_4$)     ▷ Goto Algorithm 9
$D_6$ = Next_version($D_5$)                 ▷ Predicted new version of S
```

Algorithm 8: Get_Test_Data(M, D_2)

```
Input M, $D_2$
Let $1 < k < n$
Let $D_2$ contain q modules
for $j = 1$ to $q$ do
    $x_1$ = TRUE
    for $i = 1$ to $k$ do
        if $M_j \not\subset d_i$ then
            $x_1$ = FALSE
        end
    end
    $x_2$ = FALSE
    for $i = k$ to $n$ do
        if $M_j \not\subset d_i$ then
            $x_2$ = TRUE
        end
    end
    if $x_1$ & $!x_2$ then
        $D_4 \leftarrow M_j$
    end
end
return $D_4$
```

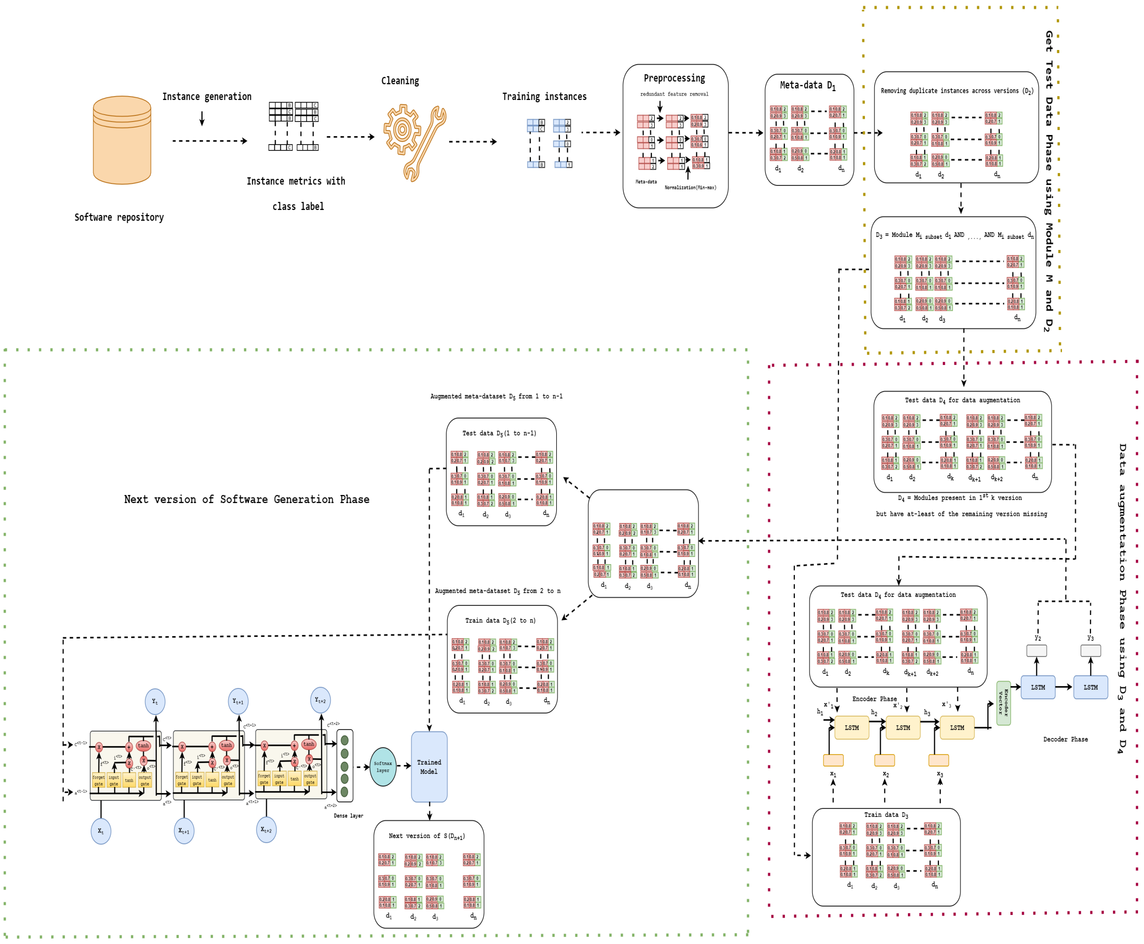

Software repository
Instance generation
Instance metrics with class label
Cleaning
Training instances
Preprocessing
redundant feature removal
Meta-data
Normalization(Min-max)
Meta-data D1
Removing duplicate instances across versions (D2)
Get Test Data Phase using Module M and D2
Test data D4 for data augmentation
D4 = Modules present in 1st k version
but have at-least of the remaining version missing
Data augmentation Phase using D3 and D4
Encoder Phase
Decoder Phase
LSTM
Train data D3
Augmented meta-dataset D3 from 1 to n-1
Test data D3(1 to n-1)
Augmented meta-dataset D3 from 2 to n
Train data D3(2 to n)
Next version of Software Generation Phase
forget gate
input gate
tanh
output gate
Dense layer
Softmax layer
Trained Model
Next version of S (Dn+1)

FIGURE 6.2: The graphical abstract of proposed approach.

6.4 Proposed Approach

The graphical abstract of the proposed architecture, consisting of four phases, is shown in Fig. 6.2. In the first phase, the dataset is extracted from the software repository. The cleaning of the dataset includes the deletion of duplicate instances and the removal of null values. Minmax normalization [6] is used to normalize the features of the dataset to values between 0 and 1.

The second phase involves obtaining the metadata set (D_1) based on the software version datasets (d_1, d_2,..d_n) of software S, using the Get_Test_Data function. From D_1, D_2 is obtained, which contains all the modules encountered in S across all n versions, along with the metrics for the versions in which they are present; and zero values for those in which they aren't present.

The third phase is the data augmentation phase, in which dataset D_3 is obtained

which contains all modules which are present across all n versions of S. A value k is chosen between 1 and n, representing the number of versions required for the data augmentation phase. From D_3, a dataset D_4 is obtained containing those modules which are present across the first k versions of S but are missing in at least one of the remaining n-k versions. The Seq2Seq model, containing k LSTM units as part of the encoder and n-k LSTM units for the decoder, is used to predict the n-k versions of these modules. D_3 and D_4, with the predicted versions, are augmented to obtain the final dataset D_5.
The final phase is the next version prediction phase, where an LSTM network is used to obtain the $(n+1)^{th}$ version of the software S from the n versions present in D_5. A detailed discussion is given in section 6.4.1.

6.4.1 Proposed algorithm

The pseudo-code of the proposed algorithm is shown in algorithm 7. Algorithm 8 represents the Get_Test_Data() function which takes in two parameters, M representing the set of all modules of software S and D_2 containing non-duplicate instances; and outputs the dataset D_4, as described earlier. Algorithm 9 represents the data augmentation phase which uses the Seq2Seq model (discussed in section 6.3.1) to predict the remaining versions of D_4 and D_5 is obtained. Finally, algorithm 10 performs the next version prediction phase on D_5. This uses the versions 1 to n-1 as the train data and versions 2 to n, i.e., d_2, d_3,...,d_n as the test data in an LSTM network.

6.4.2 Experimental setup

We have conducted our experiments on a GPU server with NVIDIA 16GB RAM, NVIDIA-SMI driver version 410.104, and CUDA version 10. We used Anaconda version 3 with Tensorflow as a back-end over Keras library. We have also used Numpy & Pandas for linear algebra operations. Further, for data interpretation, we have used Seaborn and Matplotlib for data visualization.

6.4.3 Optimization and Hyperparameters Tunning

We use the Adams optimizer and dropout regularization techniques. Adams (Adaptive Moment Optimization) [125] combines the heuristics of Momentum, and RMSProp optimizers, as shown in equations 6.2, 6.3, 6.4, and 6.5.

$$v_t = \beta_1 * v_{t-1} - (1 - \beta_1) * g_t \tag{6.2}$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2 \tag{6.3}$$

$$\Delta\omega_t = -\eta \frac{v_t}{\sqrt{s_t - \epsilon}} * g_t \tag{6.4}$$

$$w_{t+1} = w_t + \Delta\omega_t \tag{6.5}$$

where η is the learning rate, g_t is the gradient at time t w.r.t. w^j, v_t is the exponential average of gradient w.r.t. w_j, s_t is the exponential average of square of gradient w.r.t. w_j, and β_1 & β_2 are hyperparameters. Dropout regularization [236] technique avoids overfitting and provides unbiased results.
We have used the LSTM cell as a recurrent unit in the Seq2Seq model. We used k LSTM units in the encoder and (n-k) LSTM units in the decoder with a dropout rate between 0.3 and 0.1. A batch size of 32 is used, and a 20% validation split and 300 epochs with Softmax layer as the output.
A basic LSTM model consisting of 3 LSTM layers containing 100, 80, and 60 hidden units, respectively, is used for the next version prediction phase. A dropout of 0.2 is applied at each layer, and it is run for 500 epochs with a batch size of 64 and Softmax layer as the output. We have provided a similar platform for the baseline methods. We have compared the proposed model next version prediction phase with regular LSTM, Gated Recurrent Unit (GRU) [41], Convolution Neural Network (CNN) [109], Recurrent Neural Network (RNN) [291], Random Forest (RF) [29, 28], AdaBoost [9], J48 [206], and Support Vector Machine (SVM) [229]. To avoid random bias, we have provided similar preprocessing steps and added similar hypermeters and optimization techniques to LSTM, CNN, GRU, and RNN. The number of seeds is 5, the batch size is 500, and the execution slot is 2 in the RF approach. In AdaBoost, we have a 200 batch size, the weight threshold is 100, and the decision stump as the base classifier. We have used 500 batch size; the confidence factor is 0.25, the number of folds is 5 in the J48 technique. We modeled the SVM as a regression technique, and we used 100 batch size, loss at 0.1, and radial basis

function as a kernel. The remainder parameters and hyperparameters are tunned to the default value of the WEKA tool [74].

Algorithm 9: Data_augmentation(D_3, D_4)

```
1  Input D_3, D_4
2  Model = apply_Seq2Seq(D_3)
3  versions = Model.Predict(D_4)                ▷ Predict d_k, d_{k+1},..., d_n
4  for j=1 to n-k do
5      D_4 = D_4 + versions[j]
6  end
7  Let D_4 has q_1 modules & D_3 has q_2 modules
8  for j= 1 to q_1 do
9      D_5 ← D_4[j]
10 end
11 for j= 1 to q_2 do
12     D_5 ← D_3[j]
13 end
14 return D_5
```

Algorithm 10: Next_version(D_5)

```
1  Input D_5
2  Train_data = D_5[1 : n-1]
3  Test_data = D_5[2 : n]
4  Model = apply_LSTM(Train_data)               ▷ Model is basic LSTM network
5  next_version = Model.Predict(Test_data)
6  return next_version
```

6.5 Results and Discussion

Table 6.2 shows the MSE, MAE, and accuracy in percentage for all eight datasets during the data augmentation phase. MSE obtained is 107.40, 115.73, 117.41, 125.06, 127.51, 132.41, 147.37, & 201.20 for jedit, ant, velocity, poi, log4j, camel, xerces, & lucene datasets respectively. The lowest MAE values are of jedit and xerces as 30.59 and 45.47, respectively. The top three accuracies obtained are 70.23, 67.74 & 66.40 for camel, velocity, & ant datasets. Table 6.3 shows the MSE, MAE, and accuracy of the next version prediction phase. The maximum and minimum values of MSE are 45.34 and 75.00 for jedit and poi datasets, respectively. The minimum MAE values are 30.59 and 45.47 obtained for jedit and xerces datasets, respectively.

TABLE 6.2: Performance at data augmentation phase.

Data	**MSE(%)**	**MAE(%)**	**Accuracy(%)**
ant	115.73	68.99	47.58
camel	132.41	69.20	44.78
log4j	127.51	75.31	60.22
jedit	107.40	62.56	51.25
lucene	201.20	76.54	40.72
xerces	147.37	71.57	60.71
velocity	117.41	62.84	32.24
poi	125.06	61.27	57.21

Maximum accuracy is obtained at 70.23% for the camel dataset, while the minimum is at 45.28 for jedit.

Figure. 6.5 shows the plots of accuracy versus epoch and loss versus epoch for all eight metadatasets in the data augmentation phase using the Seq2Seq model. Accuracy varies after 50 epochs for both training and test data in ant, camel, jedit, and velocity metadatasets, as shown in Figures 6.5(a), 6.5(b), 6.5(c) and 6.5(d) respectively. The accuracy over the training set for lucene and log4j datasets varies till 100 and 175 epochs respectively after which they obtain a constant accuracy, as shown in Figures 6.5(e), & 6.5(f) respectively. Poi (Fig. 6.5(g)) and xerces (Fig. 6.5(h)) datasets show a variation in the accuracy throughout the 300 epochs. The loss versus epoch for lucene and log4j over both training and testing phases are almost constant, as shown in Fig. 6.5(e) & Fig. 6.5(f) respectively. Figure 6.6 shows the accuracy versus epoch and loss versus epoch plots for the next version prediction phase. The test set accuracy is high compared with the train set for log4j metadata from 1 to 190 epochs, and then the train set accuracy increases to 500 epochs. In contrast, for the remaining metadatasets, the loss over the test set is higher than that over the train set.

6.5.1 How is the proposed approach effective in predicting the bug count of every module in the next version of a software system?

According to results, five out of eight metadatasets have almost 60% or more accuracy in the next version prediction phase, which is significantly satisfying. Two metadatasets (lucene & xerces) have an accuracy between 50% and 60%. The only jedit has less than 50% accuracy, i.e., 45.47%. It will indeed reduce the enormous amount of software development costs. The predicted version doesn't contain null or negative values of any software metric.

6.5.2 How is the proposed approach effective in predicting the each software metric values of every module in the next version of a software system?

As discussed earlier, we get more than 50% accuracy on seven metadatasets in predicting the next version of software S. We generate the next versions of S, with all 20 metrics and bug count vector. We predicted all the 20 software metrics for the next version of S, the existing version of S is made of all these 20 metrics as discussed earlier. The software metrics are already scaled between 0 and 1. Few metrics values are negative due to the normalization technique. The accuracy of the metric value prediction of each module of the next version software system is the same as a bug count prediction of each module. Table 6.3 shows the accuracy MSE and MSE value of each data while predicting metrics of the next version software system.

6.5.3 How much the proposed model is significantly effective over existing learning methods?

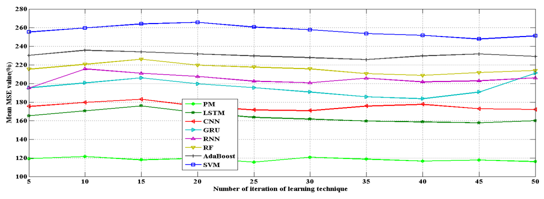
We provided a similar preprocessing step for all eight learning models. We feed metadata into these learning methods to measure the performance. Table 6.4 reports the mean MSE value of existing baseline learning models; the bold letter in Table 6.4 and Table 6.5 indicates the minimum MSE/MAE value produces by any model over given data. As the table indicates, the mean MSE of the proposed model over the next version prediction phase is minimum amongst all these approaches. Similarly, the mean MAE values of the proposed model are minimum as shown in Table 6.5. Whereas the accuracy of the next version prediction phase is maximum as reported in Table 6.6; the bold letter in the table refers to the maximum accuracy produces by any learning model over a given project. After the proposed model, the second-best learning performance is of a traditional LSTM based approach. We also compared the average performance metrics of the proposed model and baseline approaches at every different repeated iteration, as shown in Fig. 6.3. The mean of MSE and MAE, which yield by the proposed model, is lowest at 50 iterations compared with other baseline methods as shown in Fig. 6.3(a), and Fig. 6.3(b), respectively. Similarly, the accuracy produces by the suggested model is maximum at all 50 iterations, as shown in Fig. 6.3(c). It also indicated the model is highly stable and consistent.

We have statistically compared the performance of the proposed approach with baseline models using Cliff's delta [43], which is a non-parametric effect size measure; it quantifies the significant difference between the performances given approaches. The range of δ lies from -1 to 1; when the value δ is 1 or -1, then there is no overlapping between the performance of the two models. However, if δ is 0, it indicates the two approaches are completely overlapping. Table 6.10 illustrate the various δ values and its corresponding explanation [2]. Table 6.7 to Table 6.9 reports the various p values and Cliff's delta values of proposed model compared with baseline techniques. Table 6.11 signifies that the proposed model outperforms over all eight learning methods in every project.

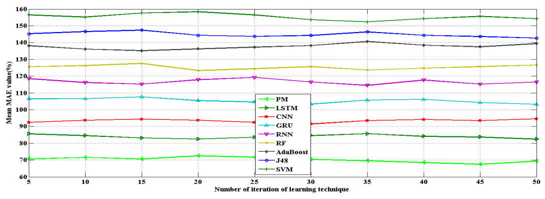
6.5.4 Insightful discussion

Due to fewer data instances of the various software systems, our accuracy obtained is only up to 70.23%. To prevent overfitting due to a lack of data samples, we have used the dropout regularization. jedit has only 1257 modules across all versions, and hence we get low accuracy (45.28%). The metadata set during the experiment also suffers from CIP, which may lead to improper training. This needs to be addressed during both phases. Undersampling and oversampling aren't beneficial to address this problem, as undersampling leads to data loss, whereas oversampling adds synthetic instances that aren't part of the original software dataset. Hyperparameter tuning is an empirical process, which can enhance performance. We have tried to tune parameters as well as possible, but we cannot say that further tuning will not improve the performance.

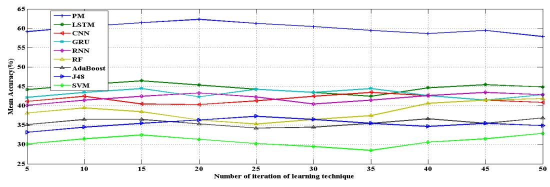
The traditional LSTM approach is the second most optimal technique amongst all existing methods. We only consider the LSTM to investigate the cost-effectiveness of the proposed model and regular LSTM. We inspected an average of all three performance measures at a different percentage of LOC, as shown in Fig. 6.4. While inspective at different percentages of LOC, we found that the proposed model is outperforming at a wide range compared with regular LSTM architecture. The average MSE and MSE is very low at different LOC as shown in Fig. 6.4(a), and



(a) Comparison of average MSE value(%) of all learning model at different iteration.



(b) Comparison of average MAE value(%) of all learning model at different iteration.



(c) Comparison of average Accuracy value(%) of all learning model at different iteration.

FIGURE 6.3: Comparison of average MSE, MAE, and Accuracy value(%) of all learning model at different iteration.

Fig. 6.4(b), respectively. Similarly, the mean accuracy of the proposed model is greater at every inspected LOC compared with the regular LSTM model, as shown in Fig. 6.4(c).

The stability of the sequence to sequence model over the data augmentation phase, Fig. 6.5(a), Fig. 6.5(b), Fig. 6.5(c), Fig. 6.5(d) reports that both the train and test set accuracy is varying, so the model is moderately unstable due to lack of training instances. But, the model is stable over the rest of the metadata. The stability of the proposed model over the next version prediction phase, the zigzag lines in Fig. 6.6(a) and Fig. 6.6(g) indicates the model is moderately unstable. In contrast, the suggested model is stable over the rest of the metadata.

TABLE 6.3: Performance at next version prediction phase.

Data	**MSE(%)**	**MAE(%)**	**Accuracy(%)**
ant	147.77	70.32	66.40
camel	143.09	66.04	70.23
log4j	120.01	71.56	60.20
jedit	45.34	30.59	45.28
lucene	185.50	71.61	54.69
xerces	108.58	45.47	51.72
velocity	109.35	59.90	67.74
poi	75.00	131.21	59.25
Average	117.97±2.48	68.33±2.20	59.43±2.40

Table 6.4: Comparison of MSE (%) of next version prediction phase with baseline learning methods. The results are in the form of mean ± standard deviation.

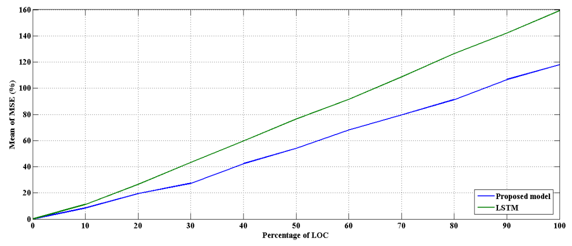
Data	**Proposed model**	**LSTM**	**CNN**	**GRU**	**RNN**	**RF**	**AdaBoost**	**J48**	**SVM**
ant	**147.77**±3.15	198.11±3.18	205.51±3.50	215.10±3.41	220.11±3.42	225.10±4.12	227.10±4.25	241.10±4.62	261.35±4.12
camel	**143.09**±2.85	195.44±2.95	210.61±3.15	223.69±3.41	230.18±3.52	237.41±4.26	240.13±4.02	256.34±4.52	278.63±4.50
log4j	**120.01**±2.52	175.50±2.45	183.40±2.56	207.19±3.15	223.85±3.41	229.50±4.39	234.68±4.56	252.22±3.99	285.91±4.12
jedit	**45.34**±1.25	75.11±1.85	80.50±1.88	93.52±1.95	101.20±2.15	108.90±2.16	113.41±2.45	124.99±3.15	141.71±3.33
lucene	**185.50**±2.55	234.20±2.65	249.50±2.48	287.44±3.50	306.88±3.85	315.60±5.56	319.82±6.66	327.55±7.52	366.48±8.56
xerces	**108.58**±2.85	146.58±1.85	154.85±2.85	179.61±3.65	192.13±3.15	206.41±2.44	211.16±3.15	218.60±3.15	239.10±3.48
velocity	**109.35**±2.55	139.55±3.45	163.82±4.12	179.82±4.41	200.05±5.55	210.54±6.18	217.60±6.69	224.61±7.52	241.34±8.12
poi	**75.00**±2.12	110.20±3.15	122.13±2.52	137.65±2.75	149.68±3.85	158.61±3.69	164.31±4.15	171.22±4.65	199.20±4.10
Average	**117.97**± 2.48	159.33±2.69	171.29±2.88	190.50±3.27	203.01±3.61	211.50±4.10	216.02±4.49	227.07±4.89	251.75±5.04

Table 6.5: Comparison of MAE (%) of next version prediction phase with baseline learning methods.

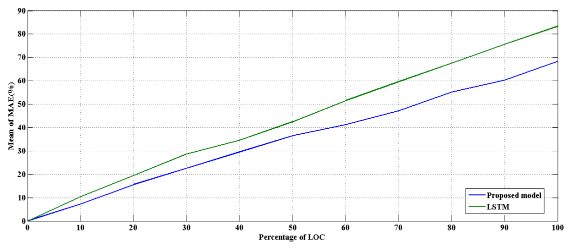
Data	**Proposed model**	**LSTM**	**CNN**	**GRU**	**RNN**	**RF**	**AdaBoost**	**J48**	**SVM**
ant	**70.32**±3.55	87.61±3.45	83.30±2.85	90.22±3.22	104.22±1.99	110.87±4.10	127.52±4.16	134.55±5.05	150.47±5.52
camel	**66.04**±3.15	81.20±2.85	90.55±2.74	105.90±2.95	120.47±3.15	131.20±3.45	147.52±4.15	155.5±4.518	162.92±4.62
log4j	**71.56**±2.11	86.32±2.55	98.52±2.75	110.22±2.69	123.60±2.88	135.82±2.95	146.45±3.15	159.89±3.55	165.50±4.05
jedit	**30.59**±1.85	42.85±1.75	55.69±1.99	69.62±2.15	76.55±2.69	89.52±2.75	95.15±3.05	103.22±3.15	115.64±3.41
lucene	**71.61**±2.15	86.55±2.62	94.65±3.10	108.37±3.12	119.65±3.11	124.33±3.41	138.41±3.16	146.71±3.41	152.82±3.50
xerces	**45.47**±1.25	59.88±1.29	68.23±1.54	76.49±2.10	89.90±1.85	95.11±2.05	107.51±2.10	114.22±2.56	119.63±2.75
velocity	**59.90**±1.45	72.23±1.35	86.55±1.55	97.51±1.60	108.65±1.48	119.88±1.68	130.52±2.18	137.55±2.69	143.55±2.50
poi	**131.21**±2.12	150.22±2.52	169.22±2.74	178.22±2.69	190.85±3.10	206.50±4.19	215.39±4.85	222.50±5.12	231.45±5.26
Average	**68.33**±2.20	83.35±2.29	93.33±2.40	104.56±2.56	116.73±2.53	126.65±3.07	138.55±3.33	146.82±3.75	155.24±3.95

Table 6.6: Comparison of accuracy (%) of next version prediction phase with baseline learning methods.

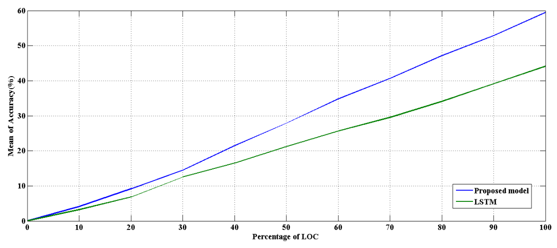
Data	**Proposed model**	**LSTM**	**CNN**	**GRU**	**RNN**	**RF**	**AdaBoost**	**J48**	**SVM**
ant	**66.40**±2.52	43.55±1.22	40.26±1.80	42.35±1.56	41.69±1.75	38.85±1.89	36.77±2.11	34.51±1.46	29.69±1.40
camel	**70.23**±3.11	53.69±1.95	49.67±1.65	50.29±1.44	49.98±1.29	48.55±1.39	46.30±1.55	45.58±1.31	44.23±1.77
log4j	**60.20**±2.55	48.60±2.14	46.85±2.31	47.88±2.09	47.69±1.88	45.96±1.89	39.55±1.76	37.85±1.68	35.99±1.53
jedit	**45.28**±1.79	29.89±1.65	27.61±1.55	28.15±1.75	28.52±1.65	25.99±1.22	22.66±1.47	18.95±1.09	15.66±0.99
lucene	**54.69**±2.65	43.55±2.41	41.69±1.98	42.15±1.69	39.50±1.77	35.41±1.56	32.99±1.35	30.28±1.22	29.65±1.55
xerces	**51.72**±2.11	39.88±2.35	37.56±2.17	35.20±1.85	33.64±1.76	32.10±1.95	31.50±2.10	28.90±1.45	26.88±1.95
velocity	**67.74**±3.12	53.67±2.45	51.56±1.85	52.46±1.95	47.55±1.55	43.11±2.12	40.92±1.75	37.58±2.19	35.52±2.50
poi	**59.25**±1.41	43.51±1.85	40.26±2.65	41.85±2.15	39.95±2.15	38.49±1.89	37.52±2.15	33.68±1.99	31.68±1.85
Average	**59.43**±2.40	44.53±2.01	41.93±1.99	42.54±1.81	41.06±1.72	38.55±1.73	36.06±1.78	33.41±1.53	31.15±1.69



(a) Cost effectiveness by comparing Mean MSE(%).



(b) Cost effectiveness by comparing Mean MAE(%).



(c) Cost effectiveness by comparing Mean Accuracy(%).

FIGURE 6.4: Cost effectiveness of Proposed model compared with regular LSTM in terms of MSE, MAE, and Accuracy.

Table 6.7: P-Value and Cliff's Delta (δ) of proposed model (PM) compared with the existing approaches in terms of MSE.

Dataset	**PM Vs LSTM**		**PM Vs CNN**		**PM Vs GRU**		**PM Vs RNN**		**PM Vs RF**		**PM Vs AdaBoost**		**PM Vs J48**		**PM Vs SVM**	
	p-vale	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ
ant	$8.6*10^{-6}$	0.88	$3.7*10^{-7}$	1	$2.6*10^{-7}$	1	$9.4*10^{-7}$	1	$3.2*10^{-8}$	1	$4.1*10^{-8}$	1	$4.4*10^{-8}$	1	$5.5*10^{-8}$	1
camel	$5.1*10^{-6}$	0.92	$6.4*10^{-6}$	1	$3.1*10^{-6}$	0.84	$1.2*10^{-7}$	1	$2.3*10^{-7}$	1	$5.6*10^{-7}$	1	$8.9*10^{-7}$	1	$1.3*10^{-8}$	1
log4j	$4.3*10^{-6}$	0.89	$5.9*10^{-6}$	1	$3.4*10^{-6}$	0.81	$3.1*10^{-7}$	1	$3.4*10^{-7}$	1	$6.4*10^{-7}$	1	$9.1*10^{-7}$	1	$2.3*10^{-8}$	1
jedit	$2.5*10^{-6}$	1	$3.1*10^{-6}$	1	$1.1*10^{-6}$	0.93	$8.8*10^{-6}$	1	$9.1*10^{-6}$	1	$2.3*10^{-7}$	1	$7.1*10^{-7}$	1	$8.1*10^{-7}$	1
lucene	$3.3*10^{-6}$	1	$2.8*10^{-6}$	1	$3.1*10^{-6}$	1	$7.6*10^{-6}$	1	$9.7*10^{-6}$	1	$3.1*10^{-7}$	1	$6.2*10^{-7}$	1	$7.3*10^{-7}$	1
xerces	$2.7*10^{-6}$	0.90	$2.8*10^{-6}$	1	$3.5*10^{-6}$	1	$6.7*10^{-6}$	1	$8.8*10^{-6}$	1	$2.1*10^{-7}$	1	$4.6*10^{-7}$	1	$6.9*10^{-7}$	1
velocity	$4.1*10^{-6}$	0.93	$5.9*10^{-6}$	1	$6.9*10^{-6}$	1	$8.1*10^{-6}$	1	$1.3*10^{-7}$	1	$3.9*10^{-7}$	1	$5.8*10^{-7}$	1	$1.1*10^{-8}$	1
poi	$3.1*10^{-6}$	0.88	$6.8*10^{-6}$	1	$7.1*10^{-6}$	1	$8.8*10^{-6}$	1	$1.9*10^{-7}$	1	$4.1*10^{-7}$	1	$6.1*10^{-7}$	1	$8.1*10^{-8}$	1

Table 6.8: P-Value and Cliff's Delta (δ) of proposed model (PM) compared with the existing approaches in terms of MAE.

Dataset	**PM Vs LSTM**		**PM Vs CNN**		**PM Vs GRU**		**PM Vs RNN**		**PM Vs RF**		**PM Vs AdaBoost**		**PM Vs J48**		**PM Vs SVM**	
	p-vale	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ
ant	$7.2*10^{-6}$	0.91	$3.4*10^{-7}$	1	$2.9*10^{-7}$	1	$8.8*10^{-7}$	1	$4.5*10^{-8}$	1	$4.9*10^{-8}$	1	$5.8*10^{-8}$	1	$6.7*10^{-8}$	1
camel	$4.2*10^{-6}$	0.96	$5.9*10^{-6}$	1	$4.4*10^{-6}$	0.92	$2.3*10^{-7}$	1	$3.4*10^{-7}$	1	$6.7*10^{-7}$	1	$9.7*10^{-7}$	1	$2.1*10^{-8}$	1
log4j	$3.2*10^{-6}$	0.92	$5.5*10^{-6}$	1	$3.8*10^{-6}$	0.88	$4.2*10^{-7}$	1	$5.1*10^{-7}$	1	$6.9*10^{-7}$	1	$9.7*10^{-7}$	1	$3.1*10^{-8}$	1
jedit	$3.1*10^{-6}$	1	$3.7*10^{-6}$	1	$2.4*10^{-6}$	1	$8.4*10^{-6}$	1	$9.4*10^{-6}$	1	$3.1*10^{-7}$	1	$6.6*10^{-7}$	1	$8.2*10^{-7}$	1
lucene	$3.7*10^{-6}$	1	$2.9*10^{-6}$	1	$4.2*10^{-6}$	1	$8.7*10^{-6}$	1	$9.9*10^{-6}$	1	$4.1*10^{-7}$	1	$5.1*10^{-7}$	1	$6.5*10^{-7}$	1
xerces	$3.1*10^{-6}$	0.94	$3.1*10^{-6}$	1	$4.1*10^{-6}$	1	$7.5*10^{-6}$	1	$8.9*10^{-6}$	1	$1.5*10^{-7}$	1	$3.9*10^{-7}$	1	$7.4*10^{-7}$	1
velocity	$3.6*10^{-6}$	0.96	$4.8*10^{-6}$	1	$7.1*10^{-6}$	1	$8.6*10^{-6}$	1	$1.9*10^{-7}$	1	$4.5*10^{-7}$	1	$6.1*10^{-7}$	1	$2.5*10^{-8}$	1
poi	$2.8*10^{-6}$	0.85	$5.4*10^{-6}$	1	$6.6*10^{-6}$	1	$8.2*10^{-6}$	1	$2.1*10^{-7}$	1	$3.9*10^{-7}$	1	$6.9*10^{-7}$	1	$7.7*10^{-8}$	1

Table 6.9: P-Value and Cliff's Delta (δ) of proposed model (PM) compared with the existing approaches in terms of Accuracy.

Dataset	**PM Vs LSTM**		**PM Vs CNN**		**PM Vs GRU**		**PM Vs RNN**		**PM Vs RF**		**PM Vs AdaBoost**		**PM Vs J48**		**PM Vs SVM**	
	p-vale	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ	*p-value*	δ
ant	$5.8*10^{-6}$	0.97	$3.8*10^{-7}$	1	$4.8*10^{-6}$	1	$5.4*10^{-7}$	1	$6.9*10^{-7}$	1	$9.1*10^{-8}$	1	$1.8*10^{-8}$	1	$2.7*10^{-8}$	1
camel	$6.9*10^{-6}$	1	$8.1*10^{-6}$	1	$7.1*10^{-6}$	1	$9.1*10^{-6}$	1	$2.4*10^{-7}$	1	$3.3*10^{-7}$	1	$5.9*10^{-7}$	1	$1.7*10^{-7}$	1
log4j	$2.9*10^{-6}$	0.92	$6.5*10^{-6}$	1	$4.1*10^{-6}$	0.98	$6.9*10^{-6}$	1	$2.1*10^{-7}$	1	$3.9*10^{-7}$	1	$7.9*10^{-7}$	1	$3.8*10^{-8}$	1
jedit	$2.9*10^{-6}$	1	$4.2*10^{-6}$	1	$3.5*10^{-6}$	1	$5.4*10^{-6}$	1	$8.6*10^{-7}$	1	$2.8*10^{-7}$	1	$5.8*10^{-7}$	1	$7.9*10^{-7}$	1
lucene	$4.9*10^{-6}$	1	$3.5*10^{-6}$	1	$4.3*10^{-6}$	1	$7.5*10^{-6}$	1	$8.6*10^{-6}$	1	$3.2*10^{-7}$	1	$4.8*10^{-7}$	1	$5.2*10^{-7}$	1
xerces	$2.9*10^{-6}$	0.90	$3.1*10^{-6}$	1	$4.7*10^{-6}$	1	$6.7*10^{-6}$	1	$7.6*10^{-6}$	1	$9.7*10^{-6}$	1	$3.1*10^{-7}$	1	$7.4*10^{-7}$	1
velocity	$2.8*10^{-6}$	0.97	$4.0*10^{-6}$	1	$3.4*10^{-6}$	1	$6.4*10^{-6}$	1	$8.7*10^{-6}$	1	$3.4*10^{-7}$	1	$5.3*10^{-7}$	1	$2.5*10^{-8}$	1
poi	$3.1*10^{-6}$	0.91	$6.1*10^{-6}$	1	$4.6*10^{-6}$	1	$7.9*10^{-6}$	1	$9.8*10^{-6}$	1	$2.7*10^{-7}$	1	$5.7*10^{-7}$	1	$6.9*10^{-8}$	1

Table 6.10: Effectiveness level [43] of cliff's delta(δ).

Cliff's delta (δ)	**Level of effectiveness [43]**
$\mid \delta \mid < 0.147$	Negligible
$0.147 \leq \mid \delta \mid < 0.33$	Small
$0.33 \leq \mid \delta \mid < 0.474$	Medium
$\mid \delta \mid \geq 0.474$	Large

Table 6.11: Number of projects in which PM is statistically significantly improves over existing approaches (+); performs more or less or approximately equally well (=); and performs loses (–) compared to the baseline techniques in form of MSE/MAE/Accuracy.

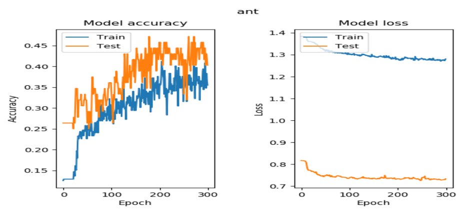
PM versus baseline methods.	+	=	−
PM Vs. LSTM	8	0	0
PM Vs. CNN	8	0	0
PM Vs. GRU	8	0	0
PM Vs. RNN	8	0	0
PM Vs. RF	8	0	0
PM Vs. AdaBoost	8	0	0
PM Vs. J48	8	0	0
PM Vs. SVM	8	0	0

6.6 Threats to validity

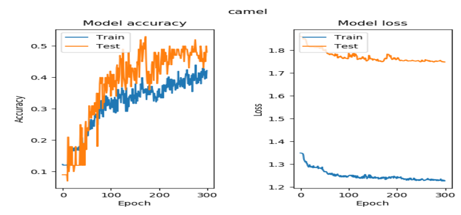
We have used eight projects of various versions from PROMISE repository datasets in our experiments. The projects are of multiple domains. All the software projects that are developed within the organization (commercial) may acquire different bug patterns. The results can be varied if the proposed model is applied without analyzing the domain. We have employed LSTM and seq2seq techniques in our proposed model. We have tried to achieve optimal results by tunning parameters/hyperparameters. Changes in those parameters can lead to better results. The results of the model can be different over other configurations and platforms.

6.7 Summary

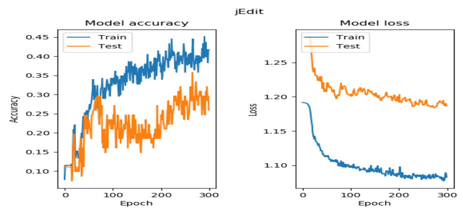
Predicting bug count and the values of other software metrics in the next version of a software system enable software development at a lower cost and proper allocation of resources for development and testing. Cross bug defect prediction only predicts the bugs in the new version, but our approach predicts metrics values and bugs in each module of the upcoming software version. Our proposed method uses deep learning to predict the next version, and we demonstrated this approach using 8



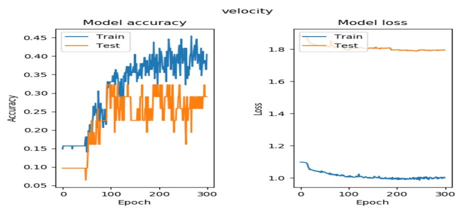
(a) Accuracy and Loss versus number of Epoch of ant metadata.



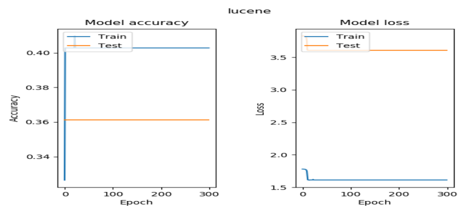
(b) Accuracy and Loss versus number of Epoch of camel metadata.



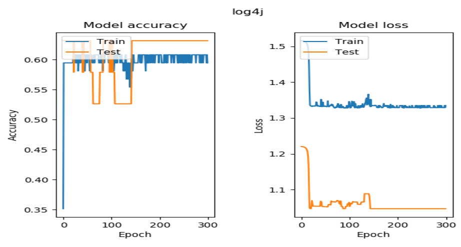
(c) Accuracy and Loss versus number of Epoch of jedit metadata.



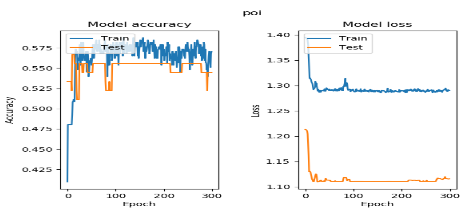
(d) Accuracy and Loss versus number of Epoch of velocity metadata.



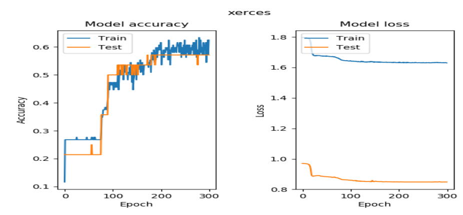
(e) Accuracy and Loss versus number of Epoch of lucene metadata.



(f) Accuracy and Loss versus number of Epoch of log4j metadata.

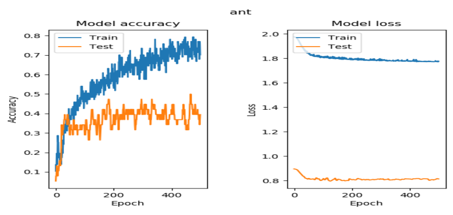


(g) Accuracy and Loss versus number of Epoch of poi metadata.

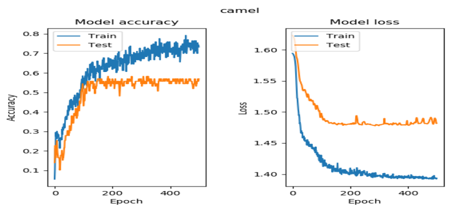


(h) Accuracy and Loss versus number of Epoch of xerces metadata.

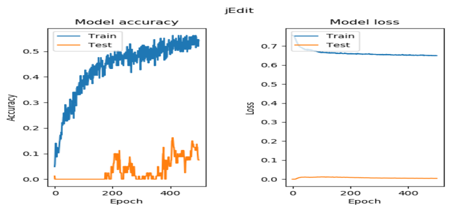
FIGURE 6.5: Accuracy and loss versus epoch of all eight metadataset in data augmentation phase.



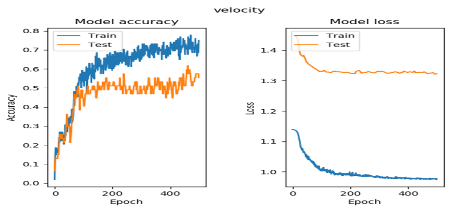
(a) Accuracy and Loss versus number of Epoch of ant metadata.



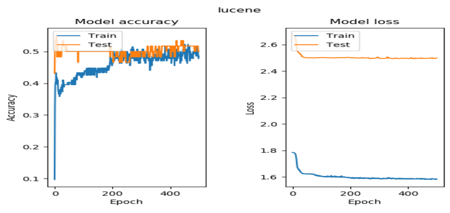
(b) Accuracy and Loss versus number of Epoch of camel metadata.



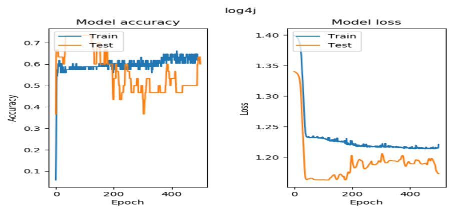
(c) Accuracy and Loss versus number of Epoch of jedit metadata.



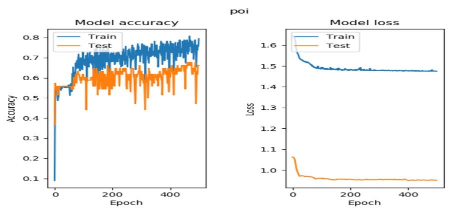
(d) Accuracy and Loss versus number of Epoch of velocity metadata.



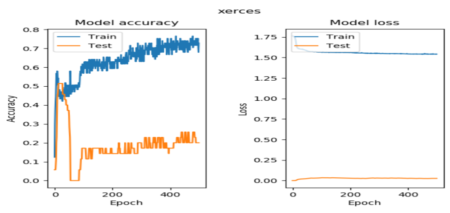
(e) Accuracy and Loss versus number of Epoch of lucene metadata.



(f) Accuracy and Loss versus number of Epoch of log4j metadata.



(g) Accuracy and Loss versus number of Epoch of poi metadata.



(h) Accuracy and Loss versus number of Epoch of xerces metadata.

FIGURE 6.6: Accuracy and loss versus epoch of all eight meta dataset in next version prediction phase.

software systems from the PROMISE repository. Our approach obtains an accuracy of 60% or more on 5 out of the 8 datasets, with the highest accuracy being 70.23% on the camel dataset and the lowest being 45.28% on the jedit dataset. The results obtained are significant in terms of accuracy. The proposed model outperforms baseline learning methods on all eight public datasets.