Chapter 5

DNNAttention: Cross-Project Defect Number Prediction Approach

5.1 Introduction

In the new era of information technology, the software projects are getting more bulky and sophisticated, so the software defect prediction (SDP) [155] technique now becomes a more consequential research domain. The training and testing data belong to the same set of projects called within-project software defect prediction (WPSDP). WPSDP requires extensive historical training data from the same software projects. However, in practice, it is infrequent to get adequate training data for the new project. Still, there are datasets from other projects such as the PROMISE repository; it comprises many diverse software projects released for defect prediction. Cross-project defect prediction (CPDP) employing data from one repository (aka. source project) to train a learning model and predict the defective or non-defective module of different projects (aka. target project), it dispenses a new viewpoint of SDP technique [142, 78, 172]. CPDP is still challenging and leads to poor performance because the model is trained to utilize a set of projects that might not generalize well over the new project [175]. The main challenge is constructing a model that can capture better generalizable properties over target projects and disregards non-generalizable properties that do not hold for the new project. In ML

literature, to bridle the data distribution between various domains [47, 58, 175] transfer learning has been applied. It extracts mutual information from one field and transfers it to another area.

Cross-project defect number prediction (CPDNP) is one step ahead of CPDP. CPDNP is the methodology to estimate the number of defects present in each module of a software system. It is an integration of CPDP and software defect number prediction (SDNP) [202, 203] mechanisms. SDNP is the method to predict the defect count in each module of a test project. As most of the defect datasets are skewed distributed toward negative class, so it suffers from the class imbalance problem [100], which leads to unsatisfactory results. Overfitting is also a prominent challenge in such predictive models. In this chapter, we dealt with the CPDNP problem and proposed a new predictive model to conquer the integrated challenges of CPDP and SDNP. We proposed a novel architecture called DNNAttention for cross-project defect number prediction. The proposed architecture constructed by employing deep neural network [94] along with attention layer [14, 252]. According to our empirical study, DNNAttension can predict the actual defect number vector in the target project using transfer learning. We also compared the performance of DNNAttention over ten baseline methods and perceived the proposed model outperforms on most of the projects. We framed four research queries (RQ) to justify the accomplishment of the proposed model. A list of RQs is given below.

RQ-1: What is the accuracy that DNNAttention reached while predicting defect numbers over the target projects?

RQ-2: What losses the DNNAttension sufferers while predicting defect numbers over a new project?

RQ-3: Compare the performance of the proposed model over exiting baseline methods?

RQ-4: How much the DNNAtention subjugate the CI and overfitting problem? How much the proposed model is stable?

The justification of each RQs is presented in section 5.4. The rest of the chapter is organized as follows. We present problem statements in section 5.1.1. In section 5.2, we discuss the background details; the proposed work is presented in section 5.3. The results and discussions are presented in section 5.4, threats to validity in section 5.5, and finally a summary of the chapter in section 5.6.

5.1.1 Problem statement

The objective of cross-project defect number prediction is to predict the number of defects in each module or class of a new project. The CPDNP is trained by a cross-heap dataset, which we syntheses by uniquely amalgamating 44 projects from the PROMISE repository; the features are the same in all 40 projects. The mathematical problem formulation and objective functions are shown below.

(a) Let S_1, S_2, S_3,..., S_n are n software system, the corresponding datasets are d_1, d_2, d_3,..., d_n. The feature f is the same throughout the projects; let total "k" features in each software S.

(b) The software S_i has a "k" feature vector and one defect number vector represented as def^{S_i}, the defect number is finite for each module, and the maximum defect number is $\max(def^{S_i})$.

(c) The objective of DNNAttention is to predict the actual defect number vector of the new software (S_{new}).

Every software S can have the different number of modules/classes (M), let S_i has m number of modules, so M_1, M_2, ..., M_m are total modules and def^{S_i} is defect number vector of S_i. The software S_i is represented in Eqn. 5.1, the new software project is represented in Eqn. 5.2.

$$S_i = (M_1, M_2, M_3....., M_m, def^{S_i}) \tag{5.1}$$

$$S_{new} = (M_1, M_2, M_3....., M_k, def^{S_{new}}) \tag{5.2}$$

We are predicting the defect number in each module of S_{new}, so we are examining it as a regression problem. Let maximum defect number in cross-heap data ($d_{cross-heap}$) is $\max(def_{cross-heap})$. We assume that the range of defects in S_{new} is from 0 to $\max(def_{cross-heap})$. There are two objective functions, first predicting $def^{S_{new}}$ in the new project S_{new}, as shown in Eqn. 5.3. Second, estimation of defect number vector minimizes the testing effort required in software development of S_{new}, as shown in Eqn. 5.4. Here κ is a nonlinear function. The cross-heap data synthesis process is illustrated in section 5.3.1.

$$Pre(def^{S_{new}}) = \kappa(d_{cross-heap}) \tag{5.3}$$

$$\min(Testing_{effort}(S_{new})) \tag{5.4}$$

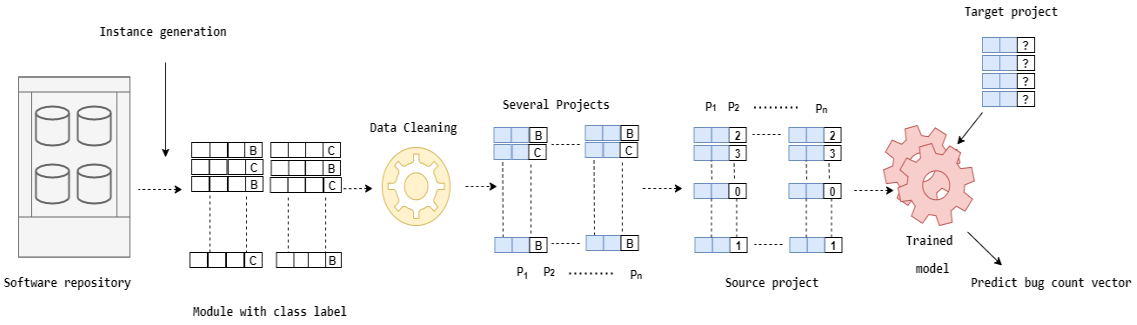


FIGURE 5.1: Underlying framework of cross project defect number prediction.

5.2 Premises

In this section, we present premises details regarding the proposed approach. First, we will illustrate the underlying framework of CPDNP, then after software projects associated with experiments, software metrics corresponding to projects, and finally, evaluation metrics.

5.2.1 Cross project defect number prediction

CPDNP is a process to estimate the defect number in each module/class of a target project. The learning model is trained by mega data of various diverse projects called cross-heap. Fig 5.1 shows the underlying architecture of CPDNP. The different projects are extracted from the software repository. Then the instance generation occurs corresponding to each software project, as present in Fig. 5.1. The process of removing noisy and irrelevant instances conducts under the data cleaning process. Now the dataset associated with each project is combined into one bigger data as a source project. The source project feeds into the learning model to train; the trained model predicts the defect number vector in the target project using transfer learning. The defect number vector contains defect information of every module in a project.

5.2.2 Software projects

We have employed 44 various software systems from the PROMISE repository as discussed in section 2.2. Table 5.1 indicates the projects along with their respective versions and various defect information. The last column of Table 5.1 reports the maximum defect values of those projects. In the last row of the table, information about the cross-heap dataset has been shown. Cross-heap consists of 84646 modules, and the maximum defect value is 62. Table 5.1 reports that most of the dataset is imbalanced; the fifth column of Table 5.1 shows the percentage of defective instances. The cross-heap dataset consists of only 16.3% of defective instances, which connote highly skewed towards the negative class, and can produce biased results towards the negative sample.
Table 2.2 reports the various software metrics that are allied with software projects. The project was designed using code complexity, and feature of object-oriented programs [108]. Table 2.2 also consists of the name and their corresponding details of software features. Each value of these metrics can be extracted from the program module [170].

5.2.3 Evaluation metrics

To compare the performance of DNNAttension over baseline methods, we have used regression performance measures. We used MSE, MAE, and accuracy as a performance measure.

5.3 Proposed work

This section will present the proposed model, followed by the data synthesis process and deep neural network architecture. In the last subsection, we will explain the experimental setup and the baseline methods. The graphical abstract of the proposed work is shown in Fig. 5.2. The instance is generated from the existing projects; the dataset consists of noisy or redundant instances that need to be unfastened. The process to detach repeated or redundant, or noisy instances from the datasets undergoing in the data cleaning process. We have manually removed a few repeated

TABLE 5.1: Software projects description.

Project	**Project's version**	**No. of modules**	**No. of defective modules**	**% of defective modules**	**Max(defective$_P$)**
ant	ant-1.3	125	33	16	3
	ant-1.4	178	40	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	522	35.53	28
	camel-1.4	872	335	16.63	17
	camel-1.6	965	500	19.48	28
ivy	ivy-1.1	111	62	56.76	36
	ivy-1.4	241	18	6.64	3
	ivy-2.0	352	56	11.36	3
jedit	jedit-3.2	272	90	33.09	45
	jedit-4.0	306	226	24.51	23
	jedit-4.1	312	217	25.32	17
	jedit-4.2	367	106	13.08	10
	jedit-4.3	492	11	2.23	2
log4j	log4j-1.0	135	34	24.18	9
	log4j-1.1	109	37	33.94	6
	log4j-1.2	205	188	91.7	10
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
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
synapse	synapse-1.0	157	21	10.19	4
	synapse-1.1	222	99	27.03	7
	synapse-1.2	256	145	33.59	9
velocity	velocity-1.4	196	147	75	7
	velocity-1.5	214	141	65.8	10
	velocity-1.6	230	79	34.34	12
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	98.7	8
xerces	xerces-1.2	440	115	16.14	4
	xerces-1.3	453	193	15.23	30
	xerces-1.4	588	429	72.95	62
	xerces-init	162	77	47.53	11
mega data	cross-heap	84646	13808	16.38	62

or noisy instances. After the collection of various projects, we have synthesized a cross-heap dataset.

5.3.1 Data synthesis

Data synthesis is the process of assembling and measuring knowledge on variables of interest. It includes the amalgamation of various software systems into one mega

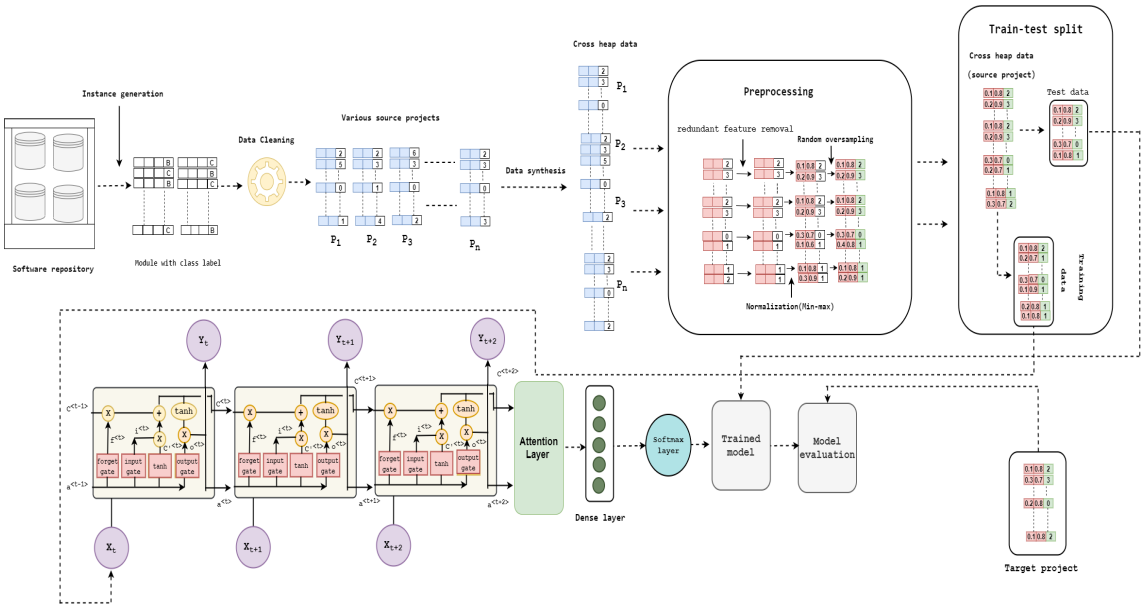


FIGURE 5.2: Graphical abstract.

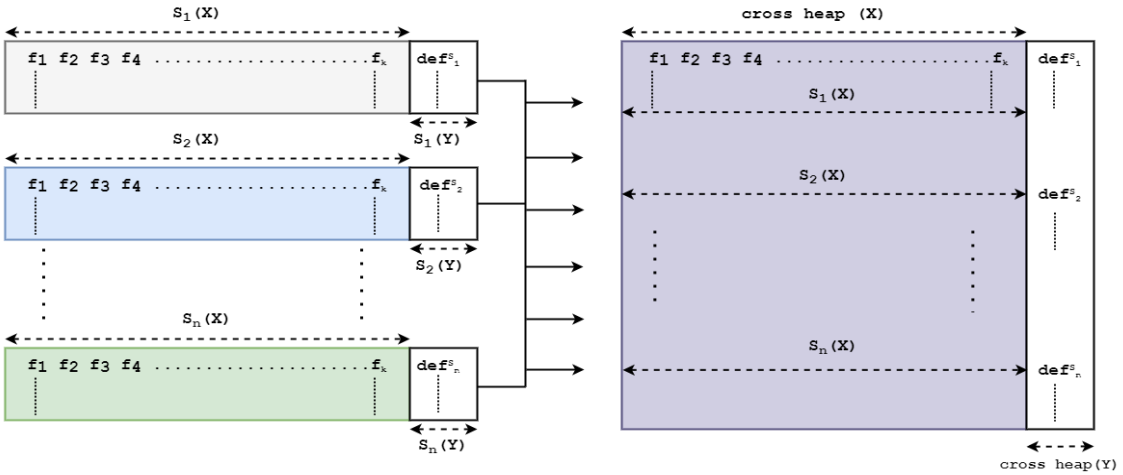


FIGURE 5.3: Underlying framework of Data Synthesis.

dataset called the cross-heap dataset. The learning model is trained using the cross-heap dataset. The trained model can predict the defect number vector using transfer learning over the target project. The software S_1 has two sections; first, $S_1(X)$ is a feature matrix that consists of information about various software metrics, $S_1(X)$ contains k number of software features as shown in Fig. 5.3. Second, $S_1(Y)$ is the column vector, and it incorporates the defect information of every module of S_1, as presents in Fig. 5.3.

$$CrossHeap(X) = S(X_1) * S(X_2) * .... * S(X_n) \tag{5.5}$$

Algorithm 4: Data synthesis

Inputs ← S_1, S_2,..., S_n /* n various software projects */
Output → CrossHeap /* cross-heap dataset */
foreach S_i **do**
  divided into $S_i(X)$ & $S_i(Y)$
end
foreach do
  CrossHeap(X) = $S(X_1)$ * $S(X_2)$ * .... * $S(X_n)$ /* feature matrix amalgamation (Eq. 5.5) */
  CrossHeap(Y) = $S(Y_1)$ * $S(Y_2)$ * .... * $S(Y_n)$ /* defect number vector aggregation (Eq. 5.6) */
  CrossHeap = CrossHeap(S_X) + CrossHeap(S_Y) /* data synthesis (Eq. 5.7) */
end

$$CrossHeap(Y) = S(Y_1) * S(Y_2) * .... * S(Y_n) \tag{5.6}$$

$$CrossHeap = CrossHeap(S_X) + CrossHeap(S_Y) \tag{5.7}$$

The pseudo-code of the data synthesis process is shown in Algo. 4. Let there is n number of software. The total n projects were amalgamated and synthesized into one mega dataset (CrossHeap). All the projects have the same number of software features, i.e., k, as discussed in section 5.1.1. The amalgamation of each software matrix is shown in Eqn. 5.5, whereas the Eqn. 5.6 reflects the combination of defect number vectors. Eqn. 5.7 is the overall amalgamation of a feature matrix along with the defect number vector into one mega data called cross-heap. The sequence of amalgamating $S_i(X)$ should be identical to $S_i(Y)$.

5.3.2 Deep neural network

We have used long short term memory (LSTM) [94], and attention layer [252] as deep neural network architecture; we have used the attention layer after the LSTM cell.

5.3.2.1 Long short term memory

The recurrent neural network (RNN) [217] is a formidable learning architecture for training small sequences, and it is capable of remembering the prior sequences to

predict the next sequence. RNN facing vanish gradient problem [93] and exploding gradient problem [187]. The LSTM [94] subjugate from these challenges. The major element of LSTM architecture is its memory, the memory called cell state or cell. The cell state is accomplished by memories of the present or past sequences. The basic architecture of the LSTM cell is shown in Fig. 2.2. The LSTM cell consists of three gates, first "forget gate," it decides which information needs to be flush, second, "input gate" that carries the input vector, and third the "output gate." All three gates have an exceptional ability to flush down or store or propagate any information. The functional equation of LSTM cell are shown from equation 2.3 to 2.8. x_t , and y_t are input and output sequences, receptively. The w_c, w_i, w_f, w_o are the weight matrix for candidate cell, input gate, forget gate, and output gate, respectively. b_c, b_i, b_f, b_o are bias value for candidate cell, input gate, forget gate, and output gate, respectively.

5.3.2.2 Attention layer

The attention layer [14] initially introduces neural language translation. Later it has been widely applied in many real-world applications such as multilingual machine translation [64], image captioning [278], image classification [258], etc. We have employed the attention layer after LSTM cells in DNNAttenstion. We have utilized Bahdanau [14] attention model in our proposed work. Fig. 2.3 shows the basic architecture of Bahdanau's attention architecture. The attention layer mainly provides extra weights to the input sequences.

5.3.3 Preprocessing

We manually took off repeated or noisy instances from the software projects and utilized all the twenty software metrics in our experiments. Software metric normalization occurs by employing min-max normalization, and its values lie between 0 to 1 as shown in Fig 5.2. Table 5.1 indicates that most of the experimental datasets are imbalanced, so combining these projects into one more extensive data also leads to imbalance. Table 5.1 reports that cross-heap consist of only 16.38% of negative class since it leads to highly skewed towards non-defective class, which causes unsatisfactory results [100]. We have resampled the data distribution over cross-heap by exercising multi-class label random oversampling [3] to circumvent the

class imbalance problem. Algo. 5 illustrates the pseudo-code of multi-label random

Algorithm 5: Multi-label random oversampling

```
Inputs ← CrossHeap, Im          /* cross-heap & percentage of imbalance */
Output → Resampled(CrossHeap)
CloneSample ← | CrossHeap | / Im*100
Label ← DataLabel(CrossHeap)                              /* labed copied */
M_IR ← calculate mean imbalance ratio(CrossHeap, Label)
foreach Label                                  /* packet of positive class */
 do
    IRL ← imbalance ration per label (CrossHeap, Label)
    if IRL_i > M_IR then
        minPacket_i++ ← Bag_i
    end
end
foreach clone_sample >0                                   /* loop for clone */
Clone every sample form minority packet do
    foreach MinP_i in MinP do
        x←random(1, MinP_i) clone_sample(min_Packet)
        if IRL_minPacket_i <= M_IR then
            min_Packet → min_Packet_i
        end
        - - clone_sample                          /* remove from cloning */
    end
end
```

oversampling that have been applied over the cross-heap dataset. The input is the percentage of imbalance ratio of the cross-heap dataset. The calculation of mean imbalance ratio (M_{IR}) and imbalance ratio per label (IRL) are conducted while cloning minority class labels. Algo. 5 clone those minority classes, which have low IRL values and vice-versa. [31] present the full illustration of this algorithm.

5.3.4 Proposed algorithm

We proposed a cross-project defect number predictor model called DNNAttention that predicts the defect number vector of the target project using transfer learning. DNNAttention is a regression model, and it is trained using a cross-heap dataset. The LSTM cell and attention are used as learning techniques. Algorithm 6 is the pseudo-code of DNNAttention. We have utilized n number of projects from the PROMISE repository to generate cross-heap data, the number of features, i.e., k is fixed in all 44 projects. After data preprocessing, cross-heap is divided into train

Algorithm 6: DNNAttension algorithm

CrossHeap ← S_1, S_2,....,S_n /* data synthesis process from n software projects from Algo. 4 */
for every M_i of CrossHeap
NumMetrics = k // number of metrics in each module
CrossHeap(X)← all features matrix of CrossHeap // feature matrix
CrossHeap(Y) defect number vector of CrossHeap // defect number vector
Normalized$_{MinMax}$(X) // Scalling of each feacture of CrossHeap(X)
Sampling$_{random}$(X,Y) // multi-label random oversampling, using Algo. 5
CrossHeap(X)→ Train$_x$, Test$_x$ & CrossHeap(Y)→ Train$_y$, Test$_y$ // cross-heeap divide into train test set
Model$_{DNNAttension}$Train$_x$, Train$_y$, Δ = d, time-step = t) // train set feeded into LSTM along with attention layer with L hidden units in LSTM and fixed droupout
foreach *epoch* e, $Batch_{size}$ = ϵ **do**
  for *each sequencial* $_{LSTM+attention}$ *layer* **do**
    f^t, i^t, $C'^{<t>}$, $C^{<t>}$, $o^{<t>}$, $a^{<t>}$ // LSTM layer from Eq. 2.3 to 2.8
    α_{ts}, c_t, a_t // attention layer from Eqn. 2.10 to 2.12
    κ_q = Prob$_{score}$(q) // score of each defect q during prediction
  end
  Optimize($\hat{Y}$, Y)$_{adam}$ // weight update
  Calculate $\sigma(z_j) = \frac{e^{(z_i)}}{\sum_{k=1}^{K} e^{z_k}}$ for each hidden unit // softmax function
  Compare(κ_q, $\kappa_{q'}$) // compared over validation set
  Calculate L($\hat{Y}$, Y) = - $\frac{1}{M'} \sum_{i=1}^{M'} [y_i \log \hat{y} + (1\text{-}y_i)\log(1-\hat{y}_i)]$ // Loss calculate
  Calculate MSE(y, $\hat{y}$), MAE(y, $\hat{y}$), Accuracy(y, $\hat{y}$).
end
Performance Model$_{DNNAttenstion}$ using MAE($\hat{Y}$, Y), MSE($\hat{Y}$, Y), Accuracy(($\hat{Y}$, Y)). // evaluation of proposed model

set (Train$_x$, Train$_y$) and test set (Test$_x$, Test$_y$) as shown in Fig. 5.2. We divided the cross-heap data into 70% train set and 30 % test set. We have also tried other splitting variations but found optimal results over 70-30 divisions. The training set feed into the LSTM cell with L hidden units, d dropout value, and t time step, then it passes through the attention layer, we set n (number of projects in cross-heap) as a time step.

To minimize loss, we have employed Adam optimization technique [125] for the network weight update process. It is the combination of two heuristic optimization methods. First, Momentum [244], second, RMS prop [213], it impedes search in the direction of oscillations.

$$\sigma(z_j) = \frac{e^{(z_i)}}{\sum_{k=1}^{K} e^{z_k}} \tag{5.8}$$

$$v_t = \beta_1 * v_{t-1} - (1 - \beta_1) * g_t \tag{5.9}$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2 \tag{5.10}$$

$$\Delta_{\omega_t} = -\eta \frac{v_t}{\sqrt{s_t + \hat{\epsilon}}} * g_t \tag{5.11}$$

$$\omega_{t+1} = \omega_t + \Delta_{\omega_t} \tag{5.12}$$

Here η, g_t, and v_t are initial learning rate, gradient at time t, and exponential average of gradient, respectively. The parameters ω_t, and s_t are exponential average of square gradient, β_1, and β_2 are hyperparameters. In DNNAttention, every hidden layer evaluates the probability score, i.e., κ_d of each defect number. The final softmax layer evaluates the performance by calculating the probability score over the validation set (target project). We have utilized sparse categorical cross-entropy as the loss function, Eqn. 5.13 refers to the equation of the loss function. This loss function is preferable when the target label is not a one hot vector [45].

$$L(\hat{Y}, Y) = -\frac{1}{M'} \sum_{i=1}^{M'} [y_i \log \hat{y} + (1 - y_i) \log(1 - \hat{y}_i)] \tag{5.13}$$

In the ith module of the training set, y_i, and $\hat{y}_i$ are the actual class label, & predicted class label, respectively. Furthermore, Y and $\hat{Y}$ are actual class labels & predicted class labels in the validation set, respectively. Values of MSE, MAE, and accuracy are calculated by comparing the actual target label y_i and predicted defect number $\hat{y}_i$ of every module of the defect number vector in the new project.

5.3.5 Transfer learning phase

The DNNAttention is trained using a cross-heap dataset; the trained model is directly applied to the new project to identify the number of defects in every module, as shown in Fig. 5.2. Then the performance of the model is evaluated over the new project. When we apply DNNAttention directly over a target project, we found the model is underperforming. In rare cases, for small target projects such as ant-1.3, ivy-1.1, ivy-1.4, and synapse-1.0, after adding labeled target projects, the one-shot learning [185] of the model occurs, then the performance is calculated. But over most target projects, no such additional training is required because retraining doesn't enhance the performance.

5.3.6 Experimental setup

We have carried out every experiment over the NVIDIA GPU server of 16 GB, 410.104 of NVIDIA-SMI, and CUDA version 10. We have employed anaconda v3 with TensorFlow as a backend and Keras library. We have applied Matplotlib as a visualization package. Scipy, Iblear, and Seaborn are various sampling, and other preprocessing packages are utilized. We have added three LSTM layers, one fully connected layer, and one attention layer into DNNAttention. The first three layers of the LSTM cell contain 100, 80, and 60 hidden nodes. We have also used a 0.2 dropout rate to conquer the overfitting problem at all three LSTM layers. We employed 40 to 60 hidden units in the attention layer. We used a batch size of 125, and every experiment was conducted from 475 to 500 epochs to achieve more extensive evidence. The time step t is set to 44, which is the number of projects in cross-heap data.

We compared the performance of DNNAttention over ten baseline methods, we trained these models by using cross-heap data and tested over the validation set. We also dispensed similar experimental arrangements for baseline techniques. These techniques are Additive Regression (AR) [272], Bagging (Bag) [241, 190], Instance-based learning (IBK) [5], Decision Stump (DS) [96], Linear Regression (LR), M5rules [201, 56], Multilayer Perceptron (MLP) [153, 121], Regression by Decentralization (RD)[65], Random Forrest (RF) [28, 29], and Zero-R (ZeroR) [277].

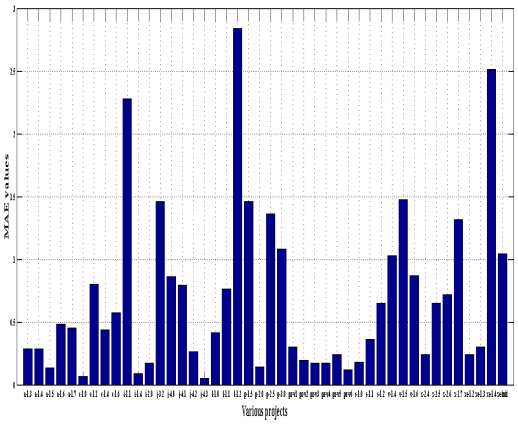
An indistinguishable preprocessing framework has been provided for all 10 methods to avoid random bias. The Batch size is set to 100 for all these learning models. The ridge parameter is set to 1.0E-8 in LR technique. The validation threshold, learning rate, momentum, and training time is set to 20, 0.3, 0.2, and 500, respectively, of the LR learning technique. In the IBK-based approach, the KNN is set to 1. The REPTree is used as a weak classifier in the bagging-based model, and the seed value is set to 1. The decision stump is configured as a weak classifier in the AR-based technique. The J48 learning model is selected as a base classifier, the number of bins is 10, and the univarientEqualFrequncyHistogram is an estimator in the RD approach. The bag size and maximum depth are set to 100 & 10, respectively, in the RF-based technique. The remainder parameters and hyperparameters are configured to the default value of the WEKA tool [74].

TABLE 5.2: Performance of DNNAttention.

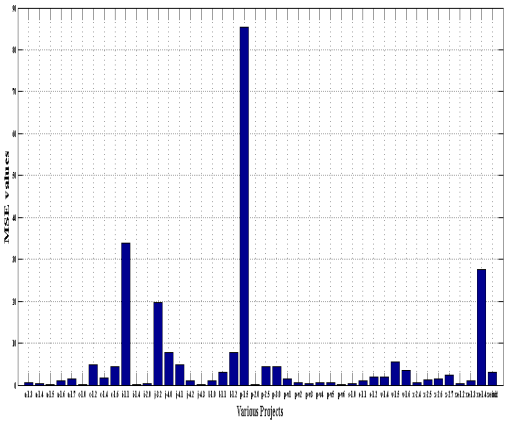
Project	Project's version	MAE	MSE	Val accuracy(%)	Accuracy(%)
ant	ant-1.3	0.286	0.604	75.01	87.50
	ant-1.4	0.283	0.420	65.55	84.96
	ant-1.5	0.136	0.149	87.23	93.05
	ant-1.6	0.480	1.112	71.42	78.12
	ant-1.7	0.447	1.474	75.01	79.83
camel	camel-1.0	0.0653	0.073	96.31	95.53
	camel-1.2	0.797	4.821	59.18	69.33
	camel-1.4	0.433	1.736	87.86	84.38
	camel-1.6	0.572	4.471	77.42	83.31
ivy	ivy-1.1	2.281	33.931	44.44	52.84
	ivy-1.4	0.085	0.123	93.10	94.12
	ivy-2.0	0.173	0.294	85.96	93.30
jedit	jedit-3.2	1.456	19.715	72.73	73.41
	jedit-4.0	0.858	7.871	73.47	81.03
	jedit-4.1	0.791	4.951	84.00	84.42
	jedit-4.2	0.264	1.077	86.44	94.02
	jedit-4.3	0.049	0.055	98.61	96.52
log4j	log4j-1.0	0.415	0.945	68.18	86.05
	log4j-1.1	0.759	3.111	61.11	82.61
	log4j-1.2	2.840	7.870	33.33	56.49
poi	poi-1.5	1.461	85.450	42.11	62.25
	poi-2.0	0.137	0.144	84.31	90.20
	poi-2.5	1.357	4.343	62.901	80.85
	poi-3.0	1.087	4.300	54.47	70.57
prop	prop-v1	0.303	1.354	84.20	85.55
	prop-v2	0.190	0.600	89.47	89.55
	prop-v3	0.170	0.314	89.41	88.59
	prop-v4	0.167	0.487	90.12	90.79
	prop-v5	0.240	0.615	84.37	85.19
	prop-v6	0.121	0.147	89.62	91.22
synapse	synapse-1.0	0.181	0.318	94.66	91.10
	synapse-1.1	0.363	0.908	66.66	81.56
	synapse-1.2	0.648	1.886	75.61	73.62
velocity	velocity-1.4	1.025	2.025	65.62	73.39
	velocity-1.5	1.473	5.612	51.43	62.50
	velocity-1.6	0.867	3.387	62.61	72.60
xalan	xalan-2.4	0.242	0.492	85.34	87.66
	xalan-2.5	0.647	1.216	55.81	66.47
	xalan-2.6	0.716	1.515	62.68	71.91
	xalan-2.7	1.313	2.297	78.77	84.34
xerces	xerces-1.2	0.235	0.415	78.87	88.26
	xerces-1.3	0.302	1.014	83.56	90.66
	xerces-1.4	2.514	27.619	64.89	65.69
	xerces-init	1.038	3.035	57.59	75.73

5.4 Result and discussion

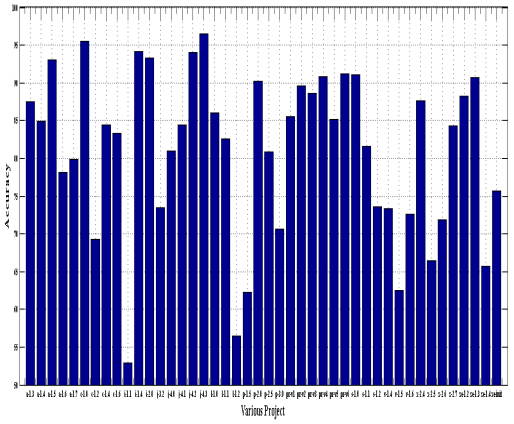
This section will address the four research queries that have been framed in section 5.1. We will also explain a few insightful results about our work and later will discuss the training time and statistical significance of the proposed model.



(a) Bar-graph of MAE over every project.



(b) Bar-graph of MSE over every project.



(c) Bar-graph of Accuracy over every project.

FIGURE 5.4: Bar-graph of MSE, MAE, & Accuracy over every project.

5.4.1 What is the accuracy that DNNAttention reached while predicting defect numbers over the target projects?

To avoid random bias, we have conducted each experiment 20 times and calculated each performance measure's mean value. Table 5.2 reports the MAE, MSE, accuracy, and validation set accuracy of the proposed model. The range of overall and validation set accuracy lies between 52.8 to 96.52 and 42.11 to 96.11, respectively, as shown in the fifth and sixth columns of Table 5.2. The few highest percentage of accuracies are of jedit-4.3 (j-4.3), camel-1.0 (c-1.0), and ivy-1.4 (i-1.4), with values

TABLE 5.3: MSE comparison of DNNAttention over baseline methods.

Project	Project's version	AR	Bag	IBK	DS	LR	M5rules	MLP	RD	RF	ZeroR	DNNAtt.
ant	ant-1.3	0.634	**0.460**	0.52	0.562	0.468	0.471	0.423	0.467	0.837	0.468	0.604
	ant-1.4	0.528	0.484	0.38	0.436	0.505	0.509	0.463	**0.293**	0.991	0.298	0.42
	ant-1.5	0.297	0.285	0.38	0.165	0.295	0.307	0.202	0.159	0.691	0.159	**0.149**
	ant-1.6	0.915	0.614	0.561	1.13	0.865	0.865	0.983	1.42	**0.132**	1.45	1.112
	ant-1.7	0.874	**0.611**	0.26	1.04	0.816	0.818	0.832	1.298	1.44	1.57	1.478
camel	camel-1.0	0.159	0.114	0.167	0.123	0.175	0.171	0.107	1.121	0.487	0.108	**0.073**
	camel-1.2	3.46	2.565	0.761	3.611	3.423	3.381	3.69	3.87	**0.259**	3.89	4.821
	camel-1.4	1.36	0.902	0.444	1.447	1.297	1.271	1.35	1.59	**0.201**	0.343	1.736
	camel-1.6	3.03	2.12	**1.37**	3.28	2.86	2.86	2.55	3.28	4.139	3.25	4.48
ivy	ivy-1.1	18.68	12.95	**10.11**	20.44	18.83	18.51	19.28	23.66	11.88	23.05	33.93
	ivy-1.4	0.498	0.347	0.218	0.242	0.423	0.425	0.266	1.45	0.205	0.345	**0.133**
	ivy-2.0	0.396	0.449	0.388	0.341	0.391	0.419	0.311	0.368	1.20	0.464	**0.275**
jedit	jedit-3.2	12.1	9.48	**4.91**	15.211	12.81	12.81	6.75	17.91	29.81	17.63	19.17
	jedit-4.0	3.01	2.255	**2.17**	4.572	2.98	2.91	1.75	5.72	1.985	17.65	7.99
	jedit-4.1	1.66	**1.22**	0.147	2.62	1.75	1.71	1.27	3.61	1.86	3.62	15.11
	jedit-4.2	0.803	0.625	0.64	0.781	0.725	0.721	0.576	1.105	**0.323**	1.187	1.131
	jedit-4.3	0.955	0.902	0.14	0.360	1.323	1.35	0.575	0.105	0.318	0.107	**0.066**
log4j	log4j-1.0	0.949	**0.626**	0.444	1.27	0.881	0.886	1.145	1.233	1.317	1.237	0.956
	log4j-1.1	1.87	1.45	0.383	2.51	1.88	1.86	1.51	2.45	1.24	2.45	3.31
	log4j-1.2	9.26	8.91	7.977	11.73	8.45	8.36	10.08	9.82	8.791	9.68	**7.37**
poi	poi-1.5	11.76	9.30	12.8	13.49	14.01	12.006	9.31	10.020	11.737	17.09	**8.34**
	poi-2.0	0.499	0.724	0.609	0.227	0.426	0.444	0.376	0.156	0.728	0.151	**0.138**
	poi-2.5	2.109	1.39	**0.581**	2.57	1.73	2.11	2.47	2.96	0.666	2.82	4.44
	poi-3.0	3.03	2.08	**0.109**	3.75	3.36	3.399	3.33	2.38	3.77	4.44	4.28
prop	prop-v1	2.44	1.681	1.99	3.617	3.414	6.736	2.611	1.826	1.601	2.481	**1.29**
	prop-v2	0.731	0.836	1.29	0.741	0.713	0.821	0.871	0.9575	0.816	0.871	**0.589**
	prop-v3	0.321	0.346	0.92	0.429	0.428	0.411	0.681	0.422	0.611	1.321	**0.309**
	prop-v4	0.590	0.499	0.73	0.598	0.586	0.581	0.702	0.543	0.5019	0.850	**0.491**
	prop-v5	0.429	0.684	0.87	0.499	0.722	0.967	0.448	0.812	0.628	0.512	**0.413**
	prop-v6	0.366	0.209	0.465	0.225	0.253	0.283	0.209	0.181	0.69	0.180	**0.139**
synapse	synapse-1.0	0.503	0.697	0.912	0.719	0.643	0.646	1.190	0.541	0.543	0.422	**0.312**
	synapse-1.1	1.822	1.843	1.291	1.901	1.762	1.751	1.793	1.921	1.283	1.761	**0.913**
	synapse-1.2	1.089	1.082	**0.264**	1.252	0.976	0.960	1.09	1.217	0.133	1.22	1.86
velocity	velocity-1.4	1.612	0.986	0.968	1.613	1.698	1.692	2.703	1.481	**0.294**	1.482	2.16
	velocity-1.5	11.63	12.81	10.24	10.19	9.321	9.37	10.83	9.243	6.74	25.23	**5.42**
	velocity-1.6	2.683	1.681	5.621	2.91	2.426	2.47	4.921	3.262	**1.641**	3.22	3.81
xalan	xalan-2.4	0.576	0.662	0.652	0.526	0.519	0.516	0.651	5.761	0.519	0.987	**0.484**
	xalan-2.5	0.773	0.578	**0.132**	0.823	0.779	0.784	0.903	0.988	0.824	0.996	1.31
	xalan-2.6	0.868	0.573	**0.253**	1.06	0.837	0.833	0.918	1.163	0.627	1.19	1.45
	xalan-2.7	1.234	0.680	0.660	1.537	1.183	1.145	1.254	1.251	**0.420**	1.54	2.31
xerces	xerces-1.2	0.705	0.533	0.635	0.4330	0.682	0.677	0.708	0.428	0.503	0.420	**0.407**
	xerces-1.3	2.63	2.288	1.420	2.953	2.40	2.29	1.28	3.63	**0.771**	33.03	1.09
	xerces-1.4	28.63	19.55	**11.83**	31.30	29.13	28.87	30.21	33.03	4.62	332.88	28.32
	xerces-init	2.26	1.78	1.086	2.64	2.28	2.33	2.38	3.03	**0.952**	3.08	3.13

96.52, 95.53, and 94.12, respectively, as shown in Fig. 5.4(c). The overall and validation set accuracy is consistently high over ant (a-1.3 to a-1.7) and prop (pr-v1 to pr-v6) projects, as shown in Table 5.2. Projects ivy-1.1, poi-1.5 (p-1.5), velocity-1.5 (v-1.5), xerces-init (xe-init), have the worst accuracies (both overall and validation), as all these projects are small and have only 111, 237, 214, and 162 number of modules, respectively as shown in Table 5.1. These projects are also imbalanced that leads to produces low accuracy. However, the accuracy is moderate over all other projects.

TABLE 5.4: MAE comparison of DNNAttention over baseline methods.

Project	Project's version	AR	Bag	IBK	DS	LR	M5rules	MLP	RD	RF	ZeroR	DNNAt.
ant	ant-1.3	0.527	0.443	0.422	0.473	0.452	0.459	0.373	0.462	0.371	0.485	**0.297**
	ant-1.4	0.519	0.465	0.482	0.441	0.507	0.502	0.483	0.424	0.302	0.461	**0.282**
	ant-1.5	0.383	0.359	0.242	0.336	0.370	0.382	0.276	0.356	0.160	0.352	**0.131**
	ant-1.6	0.555	0.483	0.285	0.617	0.536	0.542	0.506	0.661	**0.205**	0.662	0.491
	ant-1.7	0.536	0.465	0.713	0.564	0.519	0.525	0.455	0.623	**0.225**	0.622	0.456
camel	camel-1.0	0.279	0.260	0.305	0.293	0.319	0.327	0.213	0.619	0.151	0.320	**0.0743**
	camel-1.2	0.901	0.754	0.923	0.938	0.903	0.918	0.863	0.949	**0.748**	0.940	0.795
	camel-1.4	0.534	0.450	0.623	0.561	0.545	0.542	0.485	0.583	**0.217**	0.582	0.454
	camel-1.6	0.665	0.574	**0.533**	0.681	0.683	0.688	0.623	1.18	0.667	0.705	0.564
ivy	ivy-1.1	1.91	1.61	2.424	1.98	1.871	1.801	1.922	2.05	**1.37**	2.05	2.28
	ivy-1.4	0.371	0.351	0.589	0.342	0.383	0.374	0.255	0.383	0.173	0.334	**0.0859**
	ivy-2.0	0.396	0.406	0.718	0.395	0.410	0.423	0.299	0.395	0.181	0.512	**0.169**
jedit	jedit-3.2	1.352	1.132	1.220	1.445	1.334	1.351	1.140	1.521	**0.422**	1.50	1.495
	jedit-4.0	0.782	0.687	0.713	0.860	0.708	0.714	0.607	0.890	**0.551**	0.780	0.843
	jedit-4.1	0.696	0.608	**0.510**	0.756	0.708	0.717	0.576	0.840	0.526	0.840	0.775
	jedit-4.2	0.552	0.489	0.355	0.880	0.564	0.550	0.395	0.513	0.281	0.518	**0.261**
	jedit-4.3	0.572	0.553	0.114	0.610	0.652	0.653	0.553	0.319	0.284	0.310	**0.0499**
log4j	log4j-1.0	0.514	0.502	0.724	1.11	0.505	0.511	0.8299	0.602	0.499	0.601	**0.415**
	log4j-1.1	0.746	0.77	0.818	1.59	0.756	0.751	2.24	0.885	**0.526**	0.885	0.759
	log4j-1.2	2.134	1.86	**0.905**	2.19	2.12	2.19	0.373	2.178	0.909	2.17	2.37
poi	poi-1.5	1.23	1.08	1.45	1.36	1.27	1.20	1.331	1.382	**0.490**	1.382	1.443
	poi-2.0	0.466	0.642	0.608	0.351	0.646	0.650	0.318	0.356	0.662	0.350	**0.129**
	poi-2.5	1.051	0.854	**0.546**	1.615	1.093	1.101	1.163	1.201	0.620	1.205	1.435
	poi-3.0	0.910	0.75	0.103	1.022	0.923	0.916	0.967	1.091	**0.310**	1.051	1.101
prop	prop-v1	0.472	0.383	0.715	0.481	0.482	0.480	0.393	0.502	0.401	0.504	**0.303**
	prop-v2	0.356	0.254	0.53	0.393	0.336	0.337	0.275	0.412	0.291	0.412	**0.185**
	prop-v3	0.335	0.263	0.137	0.378	0.325	0.312	0.293	0.396	0.268	0.391	**0.170**
	prop-v4	0.333	0.266	0.774	0.377	0.335	0.337	0.283	0.394	0.196	0.390	**0.160**
	prop-v5	0.365	0.305	0.112	0.418	0.343	0.345	0.251	0.433	0.274	0.432	**0.237**
	prop-v6	0.395	0.306	0.672	0.305	0.378	0.373	0.232	0.365	0.213	0.364	**0.121**
synapse	synapse-1.0	0.333	0.307	0.265	0.370	0.369	0.365	0.474	0.376	0.242	0.370	**0.179**
	synapse-1.1	0.539	0.536	0.41	0.569	0.540	0.546	0.545	0.588	**0.212**	0.585	0.363
	synapse-1.2	0.544	0.779	**0.328**	0.643	0.560	0.569	0.790	0.691	0.412	0.655	0.640
velocity	velocity-1.4	0.963	1.433	0.959	0.976	0.911	0.916	1.025	1.25	**0.640**	0.920	1.028
	velocity-1.5	1.398	1.160	1.501	1.445	1.363	1.343	1.440	1.445	**0.486**	1.47	1.443
	velocity-1.6	0.853	0.745	0.983	0.898	0.830	0.836	**0.417**	0.924	0.755	0.920	0.881
xalan	xalan-2.4	0.481	0.503	0.310	0.423	0.485	0.483	0.657	0.423	0.307	0.421	**0.242**
	xalan-2.5	0.631	0.555	0.726	0.665	0.643	0.657	0.655	0.673	**0.441**	0.670	0.651
	xalan-2.6	0.633	**0.533**	0.825	0.693	0.622	0.623	0.606	1.09	0.540	0.729	0.721
	xalan-2.7	0.9232	0.661	2.333	1.041	0.895	0.893	0.910	1.07	**0.550**	1.06	1.37
xerces	xerces-1.2	0.505	0.616	0.382	0.666	0.512	0.513	0.410	0.463	0.505	0.461	**0.235**
	xerces-1.3	0.602	0.645	0.379	0.598	0.606	0.602	0.179	0.675	0.510	2.51	**0.298**
	xerces-1.4	2.44	2.051	4.779	2.53	2.46	2.45	2.50	2.53	**1.07**	2.51	2.49
	xerces-init	1.015	**0.865**	2.550	1.015	0.969	0.967	0.995	1.151	0.953	1.03	1.051

5.4.2 What losses the DNNAttension sufferers while predicting defect numbers over a new project?

The lowermost MAE values are of jedit-4.3 (j-4.3), camel-1.0 (c-1.0), and ivy-1.4 (i-1.4) projects, and their corresponding values are 0.0653, 0.085, and 0.049, respectively, as shown in Table 5.2, these are moderate size projects, and highly imbalanced. The worst MAE yield by the proposed model over log4j-1.2 (l-1.2) and xerces-1.5 (xe-1.5) datasets, with values 2.84 and 2.514, respectively, these two are small size projects. The MAE is consistently low over large projects such as prop-v1 to prop-v6. The middling MAE values turn out for synapse, jedit, xalan, camel, and ant projects, as shown in Fig. 5.4(a). The small-size projects have maximum MAE values such as xerces, and ivy, as reported in Fig. 5.4(a). Due to the lesser number of instance in the target project, the model behave overfitted and cause high loss while predicting actual defect value over the validation set.

The MSE value is nearly moderate for all projects, as shown in Table 5.2, but the

maximum MSE value is 58.45, and it turns out for poi-1.5 projects. The poi-1.5 consists of only 237 instances, as shown in Table 5.1, and the defect number varies from 0 to 20; these can be the reason for such worst MSE. Although the MSE is moderate over the rest of the projects, as shown in Fig. 5.4(b). This figure also reports that MSE is consistently low for all versions of ant, prop, synapse, and xalan software system, and these are large/moderate size projects as presents in Fig. 5.4(b). In remainder projects, MSE values are mediocre, as shown in Table 5.2.

5.4.3 Compare the performance of the proposed model over exiting baseline methods?

Table 5.3 reports the comparison of MSE values of DNNAttention over ten baseline methods. The bold letter in the Table indicates the lowest MSE amounts in all methods. We found 19 out of 44 projects have lowermost MSE values that yields by the DNNAttention. The proposed model produces the minimum MSE for all prop version projects, as shown in Table 5.3. It concludes, the proposed model outperforms baseline methods for larger projects. IBK, RF, and Bagging-based models produce lowermost MSE over 9, 10, and 4 software projects, respectively. The RF-based approach is more efficient over small-size projects, as the minimum MSE value for xerces (x-1.2, x-1.3) and ivy (i-1.4, i-2.0) produces by the RF-based method as shown in Table 5.3. The MSE yield by DNNAttention for ant (a-1.3, a-1.4, a-1.5), camel (c-1.0, c-1.2, c-1.6), and jedit (j-4.2, j-4.3) are minimal compared with other techniques. DNNAttenion exceeds the performance over other baseline methods on few moderate-size projects.

Table 5.4 reports the comparison of MAE between all methods. DNNAttension produces a minimum of MAE over 20 projects. In all versions of the prop project, the minimum MAE value yields by the DNNAttention. Moreover, RF and IBK outperform over small projects. The MAE value yields by the DNNAttention is moderate over middle size projects (camel, ant, jedit); even in few scenarios, DNNAttention outperforms over small size projects such as ivy (i-1.4, i-2.0). Although MLP and Bagging-based approaches are efficient over middle-sized projects, as shown in Table 5.4.

We have compared the accuracy of various models with DNNAttenstion, as shown in Table 5.5. The bold letters in Table 5.5 indicate the maximum accuracy. We found

TABLE 5.5: Accuracy comparison of DNNAttention over baseline methods.

Project	Project's version	AR	Bag	IBK	DS	LR	M5rules	MLP	RD	RF	ZeroR	DNNAt.
ant	ant-1.3	73.82	80.36	79.45	80.99	75.74	74.63	76.98	81.10	86.40	81.85	**87.50**
	ant-1.4	73.13	75.61	78.22	75.36	76.82	79.13	77.97	78.86	82.37	79.20	**84.96**
	ant-1.5	74.19	81.88	80.36	72.74	75.13	76.66	81.63	79.89	87.74	80.21	**93.15**
	ant-1.6	68.16	76.37	79.19	67.73	75.37	72.16	72.98	71.22	**84.87**	70.26	78.12
	ant-1.7	73.16	**80.18**	77.67	64.28	65.96	64.73	72.82	69.16	80.19	75.39	79.83
camel	camel-1.0	74.73	87.46	80.16	71.37	71.82	72.13	82.34	79.36	88.67	76.41	**95.53**
	camel-1.2	65.37	78.21	72.64	70.97	65.46	68.37	68.79	64.34	79.96	75.28	**69.33**
	camel-1.4	70.83	84.64	76.83	68.91	72.64	73.16	75.89	73.36	85.84	77.13	**84.68**
	camel-1.6	72.03	84.33	80.64	72.93	72.82	73.69	78.99	69.61	**85.96**	70.85	84.10
ivy	ivy-1.1	68.64	84.83	80.37	69.91	75.64	74.37	82.62	71.29	**86.74**	68.76	52.84
	ivy-1.4	73.56	85.40	76.09	72.52	69.45	68.13	83.64	72.88	88.37	74.64	**94.12**
	ivy-2.0	69.16	84.64	74.37	71.25	69.71	67.60	80.94	72.37	86.64	70.75	**93.31**
jedit	jedit-3.2	66.91	84.13	80.64	68.37	64.64	62.73	78.67	70.37	**86.64**	67.46	73.41
	jedit-4.0	66.82	**84.25**	79.15	71.64	69.25	68.94	80.23	73.64	83.52	72.13	81.64
	jedit-4.1	72.82	85.37	77.82	74.56	71.60	71.64	80.29	73.25	**88.62**	73.34	84.40
	jedit-4.2	78.23	86.82	80.34	77.46	75.37	74.59	80.25	73.68	87.64	77.52	**94.05**
	jedit-4.3	73.35	86.82	78.38	76.46	74.34	73.96	80.78	69.64	87.85	75.13	**96.52**
log4j	log4j-1.0	73.97	85.31	81.64	76.98	70.75	71.93	82.25	74.62	**87.82**	74.13	86.05
	log4j-1.1	68.70	**83.63**	75.56	70.20	68.93	67.13	77.34	72.64	82.82	73.64	82.88
	log4j-1.2	62.31	74.97	**77.16**	63.23	61.29	62.87	73.23	67.50	76.94	63.20	56.49
poi	poi-1.5	64.60	84.64	79.28	69.97	65.37	66.86	78.21	72.31	**85.52**	66.81	62.64
	poi-2.0	73.63	85.25	79.93	70.09	70.12	71.31	79.65	76.89	86.90	74.91	**90.20**
	poi-2.5	68.66	**83.64**	74.22	73.59	70.19	69.80	76.13	71.63	82.52	70.89	72.30
	poi-3.0	73.13	**84.91**	78.13	73.95	72.37	72.75	79.61	74.61	83.60	71.01	70.57
prop	prop-v1	76.90	85.19	81.34	73.96	76.80	74.26	80.11	73.82	**86.13**	78.90	85.55
	prop-v2	75.89	85.13	80.93	79.34	77.97	78.93	81.93	80.82	86.36	76.82	**89.55**
	prop-v3	78.20	86.97	80.78	77.26	78.96	79.85	81.13	79.96	87.64	77.20	**88.59**
	prop-v4	79.37	88.64	82.91	78.79	79.23	78.13	83.97	81.13	87.96	82.20	**90.79**
	prop-v5	74.90	85.93	81.13	73.67	79.13	79.61	81.23	79.96	**86.20**	77.13	85.19
	prop-v6	78.13	84.43	82.63	76.90	788.13	79.96	81.13	82.96	85.64	80.20	**91.22**
synapse	synapse-1.0	71.10	82.63	76.31	72.75	71.30	70.52	75.71	76.30	83.43	73.66	**91.10**
	synapse-1.1	71.95	82.30	79.10	73.55	72.20	73.60	78.19	77.13	84.52	73.20	**86.12**
	synapse-1.2	77.18	84.60	**86.10**	78.28	73.85	73.64	75.10	76.55	84.63	73.25	73.63
velocity	velocity-1.4	75.29	83.55	81.20	77.28	75.80	76.39	82.80	74.65	**84.39**	76.85	73.39
	velocity-1.5	69.80	86.15	83.09	78.97	75.95	76.80	78.53	80.35	**85.61**	71.20	62.50
	velocity-1.6	79.30	**86.63**	80.36	76.35	74.16	73.90	76.35	80.35	85.82	78.96	72.60
xalan	xalan-2.4	75.60	**88.25**	79.83	73.61	74.86	75.82	81.63	80.38	86.03	78.16	87.66
	xalan-2.5	81.11	**88.69**	89.81	81.65	79.31	77.80	83.10	82.08	88.50	81.55	66.47
	xalan-2.6	79.96	87.13	83.79	79.36	80.79	79.20	80.80	83.69	**88.52**	81.20	71.91
	xalan-2.7	77.37	**89.26**	74.89	79.63	75.79	76.36	80.94	81.30	88.64	79.85	84.34
xerces	xerces-1.2	78.93	88.13	82.82	79.13	78.82	77.36	81.27	82.89	87.29	80.43	**88.26**
	xerces-1.3	73.41	87.82	82.37	76.96	78.15	79.96	81.32	83.96	86.13	79.82	**90.66**
	xerces-1.4	76.79	**86.36**	82.84	77.28	73.93	74.16	81.37	80.82	85.16	77.82	65.69
	xerces-init	78.10	86.25	82.71	76.37	80.25	79.16	82.96	81.82	**87.82**	76.16	81.93

TABLE 5.6: Training time (sec) comparison over cross-heap data of DNNAttention over baseline methods.

Project	AR	Bag	IBK	DS	LR	M5rules	MLP	RD	RF	ZeroR	DNNAtt.
cross-heap	5.50	33.88	481.66	0.58	0.88	40.19	166.03	1697.80	77.10	0.041	12.96

that our proposed model produces maximum accuracy in 19 out of 44 projects, as reported in Table 5.5. The accuracy yields by the DNNAttention is maximum for large and middle-size projects. Projects like ant, camel, prop, and jedit have an accuracy range between 75 to 95, which is incredible. The RF-based model outperforms over 13 projects, and it is also efficient over small projects, followed by Bagging and IBK based approaches; they are also effective over moderate and minor projects.

5.4.4 How much the DNNAtention subjugate the CI and overfitting problem? How much is the proposed model stable?

We have illustrated in the earlier section that multilabel random oversampling and dropout regularization is applied to overcome CI and overfitting problems, respectively. The CI problem avoids the prediction of minority classes and leads to overfitted results. We have plotted the train set and validation set loss versus the number of epochs, as shown in Fig. 5.8 to Fig. 5.10. When these two curves start diverging, the model seems to predict the wrong defect number in the validation set because of the overfitting problem. DNNAttenion subjugates overfitting in larger projects such as prop, as shown in Fig. 5.9(f) to Fig. 5.9(k). Although in a few middle-size projects, the model presents overfitted results, such as projects ant-1.4, ant-1.6, jedit-4.1, jedit-3.2, log4j-1.0, etc. The proposed model also avoids overfitting in several small projects such as all versions of xerces, and synapse, as shown in Fig. 5.6(l) to Fig. 5.9(o) and Fig. 5.9(p) to Fig. 5.9(r), respectively.
The stability of the model can be observed by accuracy versus epoch graph, we have plotted the train set, and validation set accuracy versus epoch graph in Fig 5.5 to Fig. 5.7. The model is unstable in few small projects as there difference in validation set and train set accuracy is very high such as in projects ivy-1.1, ivy-2.0, xerces-1.3, synapse-1.0, synapse-1.1, and synapse-1.2 as shown in Figures 5.7(a), 5.7(c), 5.9(p), 5.9(q), 5.9(r), respectively. DNNAttention is highly stable over large projects. In Fig. 5.6(f) to 5.9(k), we can observe that the two-line curves have a minimum difference, which leads to a highly stable model. In very few moderate-sized projects such as ant-1.4, ant-1.6, and jedit-4.0, the proposed model is moderately stable as two curves have slight differences and are not flatten, as shown in Fig. 5.5(b), Fig. 5.5(c), and 5.5(n), respectively, this is due to the high imbalance ratio in various defect number. In the remaining projects, the model is significantly stable.

5.4.5 Some insightful discussion

DNNAttention is not completely subjugate overfitting problems. Many small projects produce high loss over validation sets such as xerces, poi, ivy, etc. Even the proposed model is unstable over few small projects, this is due to the small number of instances in the dataset, and their defect count values vary from 0 to max-def. The model faces an incalculable situation while predicting actual defect value in the

target projects. We have only added 44 projects to our cross-heap dataset. The deep learning model is the hunger for data points; the more the training data more efficient will be a predictive model. DNNAttention is suffering from inadequate training instances. Table 5.6 refers to the training time taken by all the models. DNNAttention takes reasonable training time, i.e., 12.96 sec compared with other traditional models. RF-based model, which is effective over small projects, still takes 77.10 sec. The attention layer takes quadratic time to run; this is a major downside of the model, and it makes the model time-consuming. Hyperparameter tuning is an evolutionary process; we have tried to optimize the outcome by tunning the parameters and hyperparameters of the DNNAttention. The results can be more optimal when better tunning occurs. No work has been done that can guide to achieve optimal global results by hyperparameters tunning; this is also one of the pitfalls of DNNAttention.

We performed the Wilcoxon Signed-Rank test [275], a non-parametric test, to analyze the performance of DNNAttention statistically. Table 5.5 listed the accuracies of other most optimal models and DNNAttension in the third and fourth columns, respectively. The sample size is 44, the mean is 495, the standard deviation is 85.69, and the z-value is -2.1356. We found that the W-value is 312, the mean difference is 0.98, the sum of positive ranks is 678, and the sum of negative ranks is 312. After calculating, we found the p-value is 0.01618, so the result is significant for $p < 0.05$.

5.5 Threats to validity

In this section, we will illustrate potential threats and their validity during our empirical study and experiments. The internal threats are mainly anxious about the uncontrolled internal constituents; they might influence the experiment's outcome. The principal internal threats are those defects that are introduced while conducting experiments. To minimize these threats, we have used test cases to verify the correctness of our investigations. Vary in hyperparameters can change the results of the DNNAttenstion; more optimal outcomes can be found. The end result of the DNNAttention should be considered within the domain. A software system developed within the organizations (e-commerce projects) may own other defect patterns. We have employed 44 software projects of the PROMISE repository, our selected

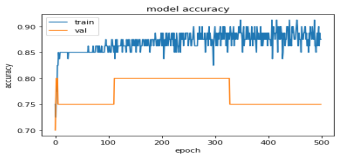
TABLE 5.7: Comparison of accuracy produced by the DNNAttention over most optimal baseline model.

Project	**Project's version**	**Other optimal method**	**DNNAttention**
ant	ant-1.3	86.14 (RF)	**87.50**
	ant-1.4	82.15 (RF)	**84.96**
	ant-1.5	87.64 (RF)	**93.15**
	ant-1.6	**84.56** (RF)	78.12
	ant-1.7	**80.53** (Bag)	79.83
camel	camel-1.0	88.78 (RF)	**95.53**
	camel-1.2	**79.53** (RF)	69.33
	camel-1.4	85.67 (RF)	**84.68**
	camel-1.6	**85.21** (RF)	84.10
ivy	ivy-1.1	**86.99** (RF)	52.84
	ivy-1.4	88.25 (RF)	**92.12**
	ivy-2.0	86.61 (Bag)	**93.31**
jedit	jedit-3.2	**86.99** (RF)	73.41
	jedit-4.0	**84.12** (Bag)	81.64
	jedit-4.1	**88.10** (RF)	84.40
	jedit-4.2	87.88 (RF)	**94.05**
	jedit-4.3	87.33 (RF)	**96.52**
log4j	log4j-1.0	**87.66** (RF)	86.05
	log4j-1.1	**83.63** (Bag)	82.88
	log4j-1.2	**77.14** (IBK)	56.49
poi	poi-1.5	**85.21** (RF)	62.64
	poi-2.0	86.99 (RF)	**90.20**
	poi-2.5	**83.64** (Bag)	72.30
	poi-3.0	**84.69** (Bag)	70.57
prop	prop-v1	**86.16** (RF)	85.55
	prop-v2	86.93 (RF)	**89.55**
	prop-v3	87.82 (RF)	**88.59**
	prop-v4	88.13 (Bag)	**90.79**
	prop-v5	**86.16** (RF)	85.19
	prop-v6	85.71 (RF)	**91.22**
synapse	synapse-1.0	83.08 (RF)	**91.10**
	synapse-1.1	84.22 (RF)	**86.12**
	synapse-1.2	**86.12** (IBK)	73.63
velocity	velocity-1.4	**84.51** (RF)	73.39
	velocity-1.5	**85.65** (RF)	62.50
	velocity-1.6	**86.66** (Bag)	72.60
xalan	xalan-2.4	**88.21** (Bag)	**87.66**
	xalan-2.5	**88.93** (Bag)	66.47
	xalan-2.6	**88.31** (RF)	71.91
	xalan-2.7	**89.19** (Bag)	84.34
xerces	xerces-1.2	88.13 (Bag)	**88.26**
	xerces-1.3	87.16 (Bag)	**90.66**
	xerces-1.4	**86.16** (Bag)	65.69
	xerces-init	**87.94** (RF)	81.93

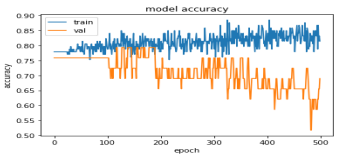
projects are unlike in size and percentage of defects, but the number of software metrics is the same, so we can not generalize our findings over other metrics. Therefore a few other projects, such as commercial projects, are indispensable to further investigation. The results of the experiments can change after varying the experimental environment. To avoid random bias, every experiment is conducted 20 times and taken the mean value of each performance measure. To test the significance of the model, we have performed a non-parametric test.

5.6 Summary

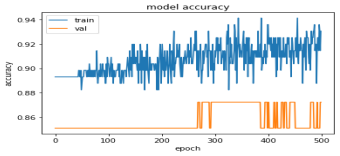
Cross-project defect number prediction is finding the defect number of each module of a new project. It is the combined approach of software defect number prediction and cross-project defect prediction. In this chapter, we try to predict the defect number vector of the target project. We have uniquely synthesized the cross-heap dataset from 44 projects of the PROMISE repository. We have proposed a CPDNP approach by employing LSTM and attention layer architectures called DNNAttention and train it by applying cross-heap. DNNAttenion predicts the defect number vector of the target project by transfer learning. The proposed model subjugates the overfitting, class imbalance problem, and it can more precisely generalize software properties over new projects. We have compared the values of MSE, MAE, and accuracy of DNNAttention over ten baseline methods. We found, out of 44 projects, 19 and 20 have lowermost MSE and MAE, respectively, whereas 19 have maximum accuracy. We discovered that DNNAtention is more suitable for large and moderate-sized projects, whereas it produces a high loss for small-sized projects. We also found that the training time of DNNAttention is modest compared with other methods. The model is stable over large and middle-size projects, although unstable over small projects. The deep learning model is a hunger of training set, due to lack of adequate training data DNNAttension underperformed over small projects.
Estimating a few software metrics of a module and the number of bugs provided for every module gives more information about the development of newer version software systems. This information will help the developer team leader to allocate development resources more optimally. In the next section, we propose a novel architecture to the estimated full upcoming version of a software system.



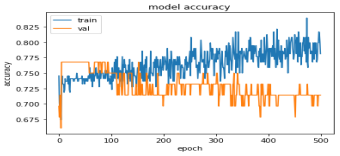
(a) Accuracy of ant-1.3.



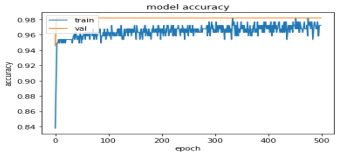
(b) Accuracy of ant-1.4.



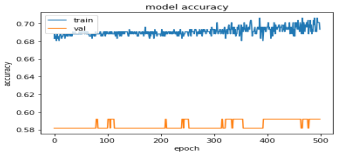
(c) Accuracy of ant-1.5.



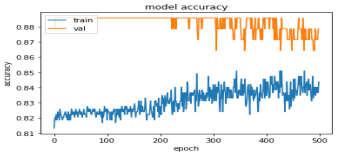
(d) Accuracy of ant-1.6.



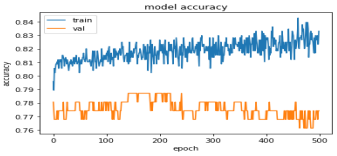
(e) Accuracy of camel-1.0.



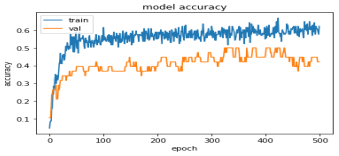
(f) Accuracy of camel-1.2.



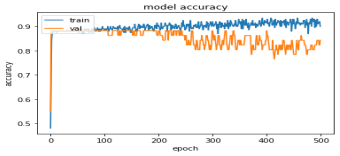
(g) Accuracy of camel-1.4.



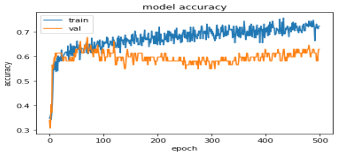
(h) Accuracy of camel-1.6.



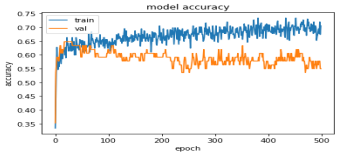
(i) Accuracy of poi-1.5.



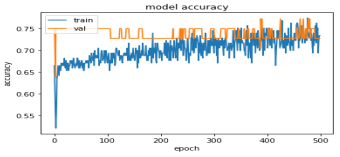
(j) Accuracy of poi-2.0.



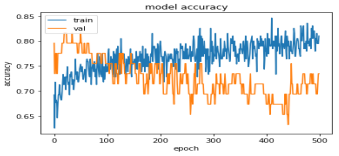
(k) Accuracy of poi-2.5.



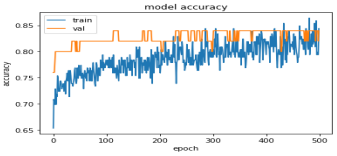
(l) Accuracy of poi-3.0.



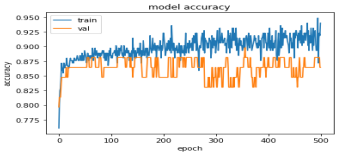
(m) Accuracy of jedit-3.2.



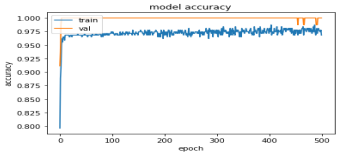
(n) Accuracy of jedit-4.0.



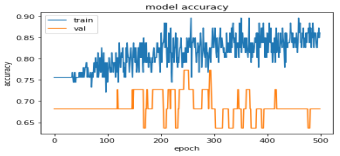
(o) Accuracy of jedit-4.1.



(p) Accuracy of jedit-4.2.

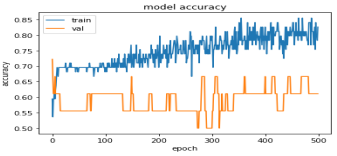


(q) Accuracy of jedit-4.3.

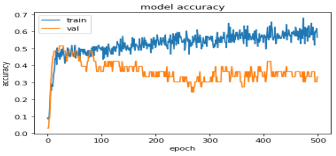


(r) Accuracy of log4j-1.0acc.

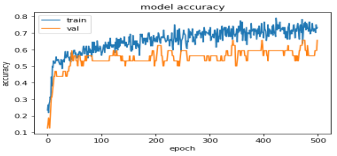
FIGURE 5.5: Accuracy of datasets.



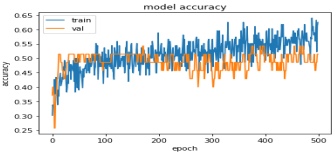
(a) Accuracy of log4j-1.1.



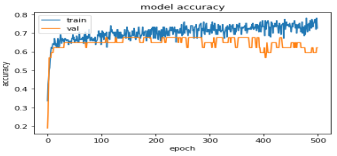
(b) Accuracy of log4j-1.1.



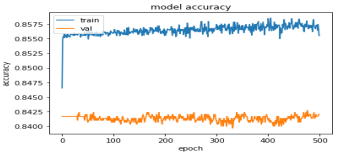
(c) Accuracy of velocity-1.4.



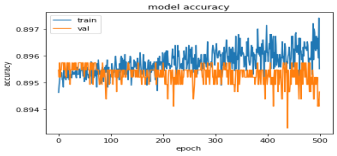
(d) Accuracy of velocity-1.4.



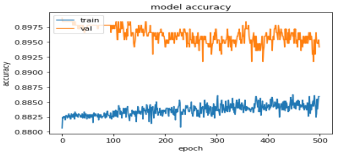
(e) Accuracy of velocity-1.6.



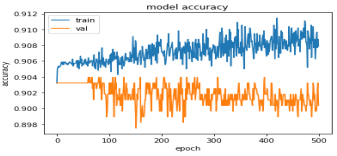
(f) Accuracy of prop-1.



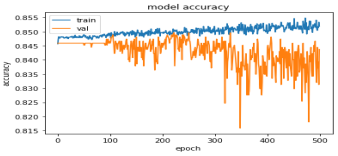
(g) Accuracy of prop-2.



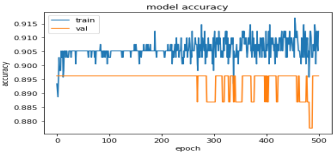
(h) Accuracy of prop-3.



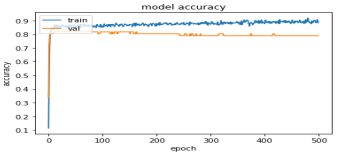
(i) Accuracy of prop-4.



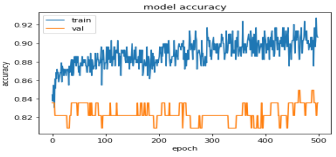
(j) Accuracy of prop-5.



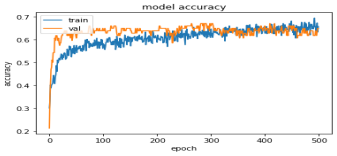
(k) Accuracy of prop-6.



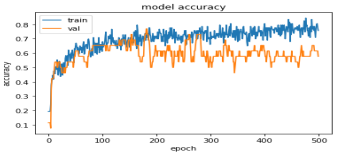
(l) Accuracy of xerces-1.2.



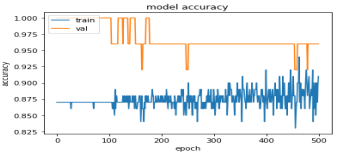
(m) Accuracy of xerces-1.3.



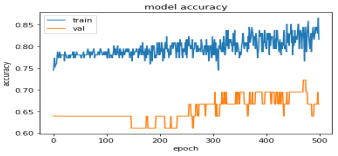
(n) Accuracy of xerces-1.4.



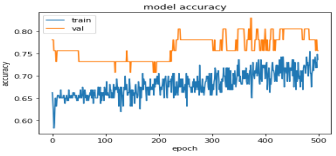
(o) Accuracy of xerces-init.



(p) Accuracy of synapse-1.0.

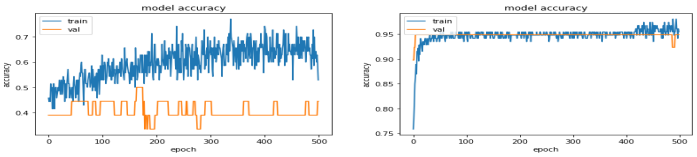


(q) Accuracy of synapse-1.1.



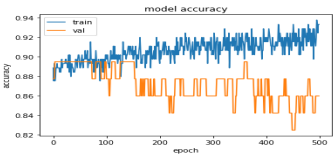
(r) Accuracy of synapse-1.2.

FIGURE 5.6: Accuracy of datasets.

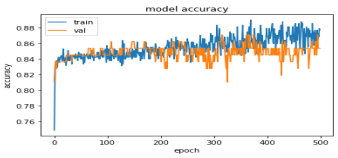


(a) Accuracy of ivy-1.1.

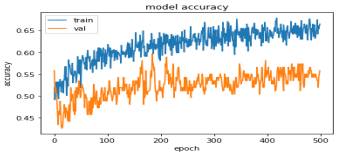
(b) Accuracy of ivy-1.4.



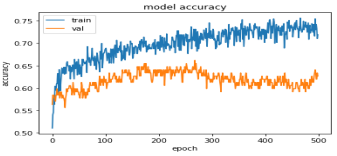
(c) Accuracy of ivy-2.0.



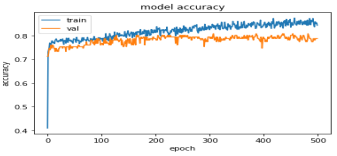
(d) Accuracy of xalan-2.4.



(e) Accuracy of xalan-2.5.

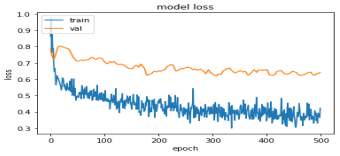


(f) Accuracy of xalan-2.6.

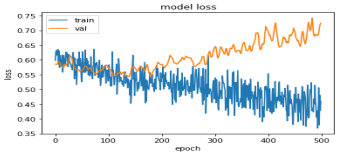


(g) Accuracy of xalan-2.7.

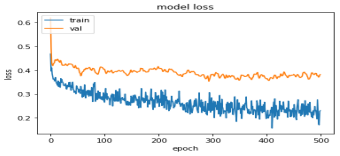
FIGURE 5.7: Accuracy of datasets.



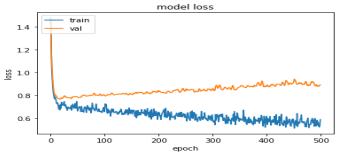
(a) Loss of ant-1.3.



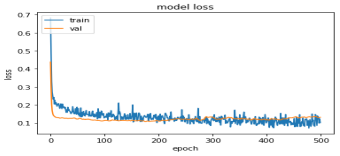
(b) Loss of ant-1.4.



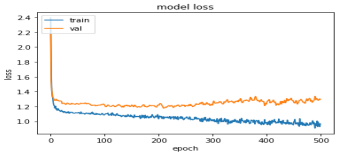
(c) Loss of ant-1.5.



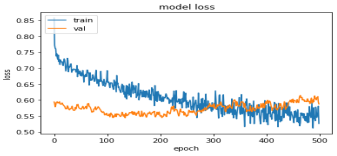
(d) Loss of ant-1.6.



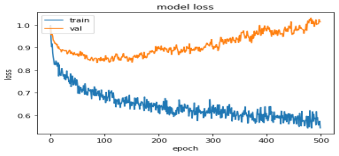
(e) Loss of camel-1.0.



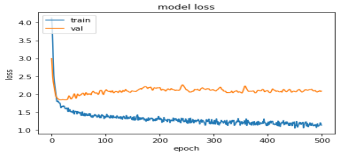
(f) Loss of camel-1.2.



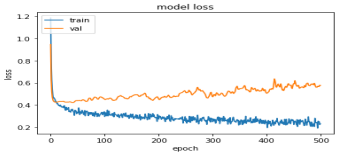
(g) Loss of camel-1.4.



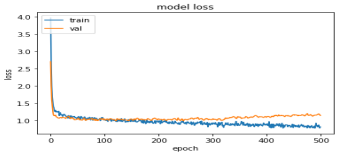
(h) Loss of camel-1.6.



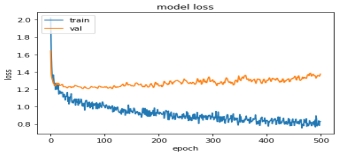
(i) Loss of poi-1.5.



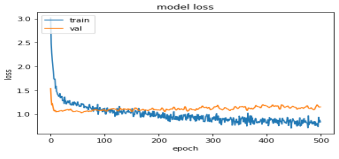
(j) Loss of poi-2.0.



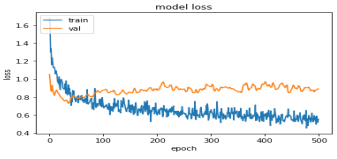
(k) Loss of poi-2.5.



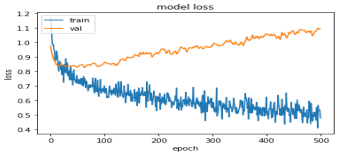
(l) Loss of poi-3.0.



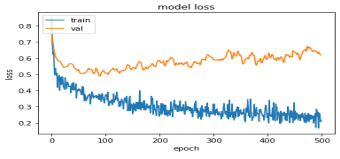
(m) Loss of jedit-3.2.



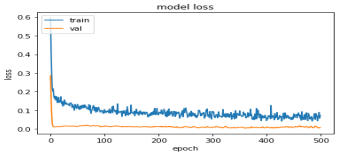
(n) Loss of jedit-4.0.



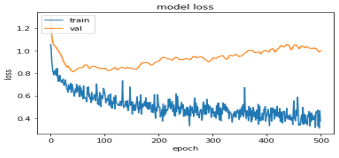
(o) Loss of jedit-4.1.



(p) Loss of jedit-4.2.

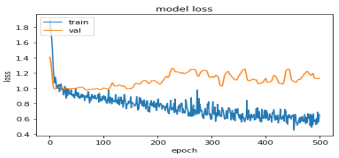


(q) Loss of jedit-4.3.

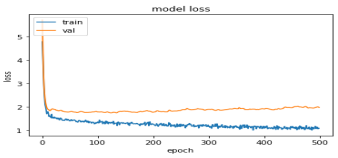


(r) Loss of log4j-1.0loss.

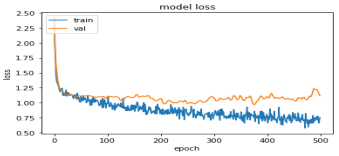
FIGURE 5.8: Loss of datasets.



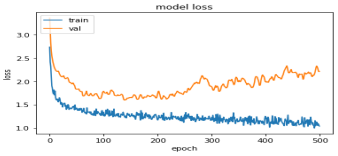
(a) Loss of log4j-1.1.



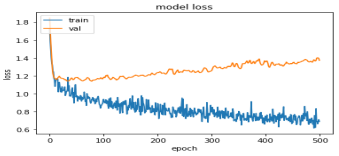
(b) Loss of log4j-1.1.



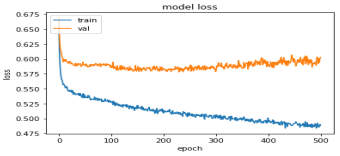
(c) Loss of velocity-1.4.



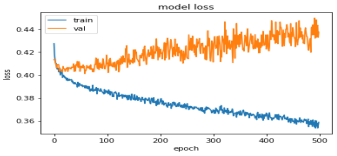
(d) Loss of velocity-1.4.



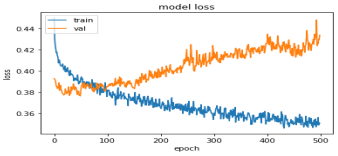
(e) Loss of velocity-1.6.



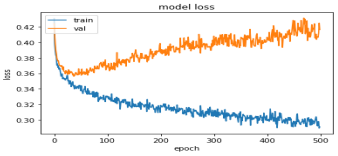
(f) Loss of prop-1.



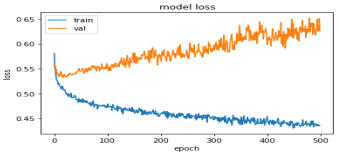
(g) Loss of prop-2.



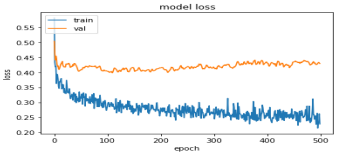
(h) Loss of prop-3.



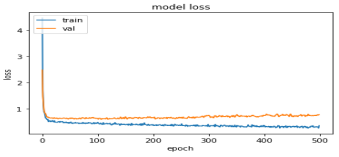
(i) Loss of prop-4.



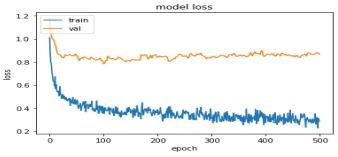
(j) Loss of prop-5.



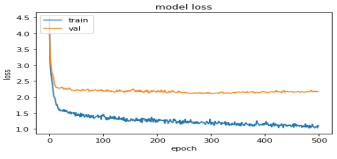
(k) Loss of prop-6.



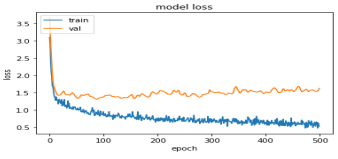
(l) Loss of xerces-1.2.



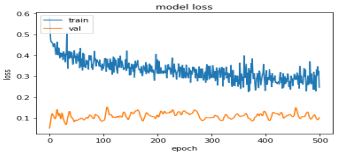
(m) Loss of xerces-1.3.



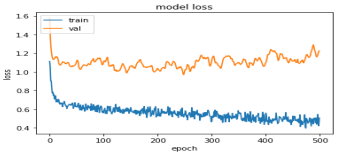
(n) Loss of xerces-1.4.



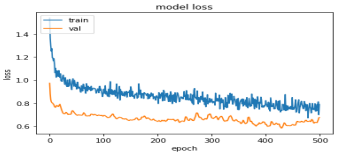
(o) Loss of xerces-init.



(p) Loss of synapse-1.0.

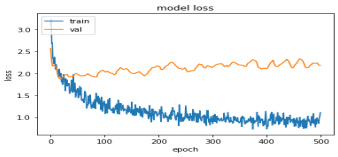


(q) Loss of synapse-1.1.

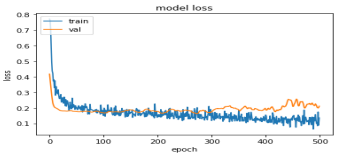


(r) Loss of synapse-1.2.

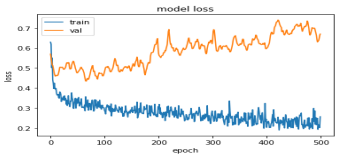
FIGURE 5.9: Loss of datasets.



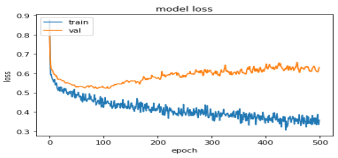
(a) Loss of ivy-1.1.



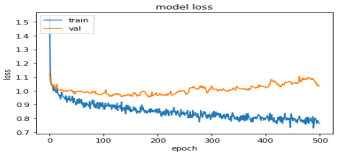
(b) Loss of ivy-1.4.



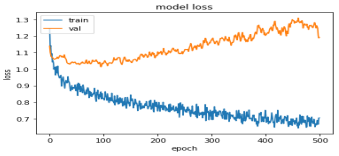
(c) Loss of ivy-2.0.



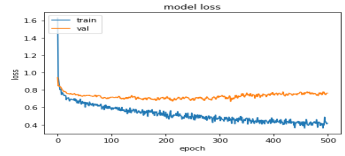
(d) Loss of xalan-2.4.



(e) Loss of xalan-2.5.



(f) Loss of xalan-2.6.



(g) Loss of xalan-2.7.

FIGURE 5.10: Loss of datasets.