ResNet: Bug Prediction Model

Deepesh Tank
Computer Science and Engineering
Indian Institute of Technology
Varanasi, (IIT BHU)
Varanasi, UP, India
deepeshtank.cse18@itbhu.ac.in

***Abstract*—Software Bug Prediction Task is an important issue in software development and maintenance processes, which concerns with the overall of software successes. This is because predicting the software faults in earlier phase improves the software quality, reliability, efficiency and reduces the software cost. However, developing robust bug prediction model is a challenging task and many techniques have been proposed in the literature. This paper presents a software bug prediction model based on ResNet Neural Network using Multi-class classification approach where each class represents a discrete number of bugs. The evaluation process showed that Deep Learning Model can be used effectively with high accuracy rate. Furthermore, we will compare the model performance on different software modules provided in Open-Source PROMISE repository.**

Keywords—ResNet Neural Network, Software Bug Prediction, PROMISE dataset, ResNet50, Artificial Neural Networks *(ANNs)*

I. Introduction

Software engineering literature reveals extensive interest of researchers to predict faults in software. Availability of limited resources as compared to bugs' quantity needs appropriate allocation of these resources [2,3]. One of the important requirements of quality assurance is not only to perform code testing, but also to identify fault-prone modules as early as possible. Therefore, in recent years researchers have put more efforts to minimize the maintenance cost by developing fault prediction models.

Software repositories i.e., the databases of version controlling and bug tracking systems are being used to develop Machine Learning (ML) based fault prediction models. Bug repository and version controlling data are accumulated during the evolution of any software. Researchers use these evolutionary data to extract software metrics and apply ML techniques to predict fault prone software modules. Different types of metrics such as code, design and requirement are effectively used to predict the faulty modules. Jiang et al. [4] used code and design metrics data of 14 different software projects and applied different modeling techniques on the data to build fault prediction models. They discovered that code and design metrics were useful, but code metrics were more reliable than design metrics. Similarly, several researches have been conducted using software metrics to predict software bugs [5,6]. Whereas, few research works have been performed for the classification and prediction of severe bugs [7].

In this paper We have used PROMISE software Repository to predict bugs in the software [10]. We built our model using untrained ResNet50 Neural architecture and then trained it using labelled data [9]. Our experimental data comprised of 20 metric values of each version of software modules (classes).

The paper is organized as follows: in section II Metrices; in section III Methodology; in section IV Evaluations Parameters; the main results and conclusions are presented in section IV, with conclusions to follow in section VI.The last section contains references.

II. Metrices

Following software metrices are used:

A. Weighted Methods for Class

Weighted methods for Class (WMC) was originally proposed by C&K as the sum of all the complexities of the methods in the class. Rather than use Cyclomatic Complexity they assigned.each method a complexity of one making WMC equal to the number of methods in the class[1].

B. Depth of Inheritance

Depth of Inheritance Tree (DIT) is a count of the classes that a particular class inherits from.

C. Coupling between Objects

Coupling between objects (CBO) is a count of number of classes that are coupled to a particular class i.e., where the methods of one class calls the methods or access the variables of the other.

D. Number of Children

Number of Children (NOC) is defined by C&K the number of immediate subclasses of a class. [1]

E. Response for Class

This is the size of the Response set of a class. The Response set for a class is defined by C&K as; a set of methods that can potentially be executed in response to a message received by an object of that class[1]; That means all the methods in the class and all the methods that are called by methods in that class. As it is a set each called method is only counted once no matter how many times it is called.

F. Lack of Cohesion of Methods

Lack of cohesion of methods (LCOM) is probably the most controversial and argued over of the C&K metrics [1]. In their original incarnation C&K defined LCOM based on the numbers of pairs of methods that shared references to instance variables. Every method pair combination in the class was assessed. If the pair do not share references to any instance variable then the count is increased by 1 and if they do share any instance variables then the count is decreased by 1. LCOM is viewed as a measure of how well the methods of the class co-operate to achieve the aims of the class. A low LCOM value suggests the class is more cohesive and is viewed as better.

G. Efferent Coupling

This metric is used to measure interrelationships between classes. As defined, it is a number of classes in a given package, which depends on the classes in other packages. It enables us to measure the vulnerability of the package to changes in packages on which it depends.

H. *Afferent Coupling*

This metric is an addition to metric Efferent Coupling(Ce) and is used to measure another type of dependencies between packages, i.e. incoming dependencies. It enables us to measure the sensitivity of remaining packages to changes in the analyzed package.

I. *Number of Public Methods*

The Number of public methods (NPM) metric simply counts all the methods in a class that are declared as public. It can be used to measure the size of an API provided by a package.

J. *Lack of Cohesion in Methods*

Lack of cohesion in methods (LCOM3) varies between 0 and 2.

m - number of procedures (methods) in class , a - number of variables (attributes) in class, µ(A) - number of methods that access a variable (attribute)

$$\text{LCOM3} = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1 - m}$$

K. *Average Method of Complexity(AMC)*

L. *Maximum McCabe's Cyclomatic Complexity*

Maximum McCabe's Cyclomatic Complexity (Max_CC) values of methods in the same class.

M. *Avg _CC Mean McCabe's*

Cyclomatic Complexity values of methods in the same class Bug Number of bugs detected in the class.

N. *Depth of Inheritance*

Depth of Inheritance Tree (DIT) is a count of the classes that a particular class inherits from.

III. METHODOLOGY

We take follow the approach of multiclass classification using ResNet Neural Network. Where each class express the number of the bug occurring in the software. Detailed methodology is described using following sub-sections:

A. *Preprocessing*

It can be described in following steps.

1) First we scale the dataset using Z score Normalisation

2) To remove class imbalance we use Random over Sampling. 'not majority' sampling strategy is used in it.

3) We split the dataset in both ratio of 70:30 and 80:20 for getting experimental results.

B. *Model Architecture*

We use Keras Sequential API for creating our Model Architecture[8]. Input is taken in form of image of size (64*64*3) therefore we first add a denser layer to convert input data to image of mentioned dimension. Then we add untrained ResNet50 Model to the architecture[9]. As we know the output of ResNet50 is of dimension 2048. Therefore, we reduce the number of dimensions using multiple dense layers with 'relu' activation last layer of the architecture has a SoftMax layer to predict the number of bugs[9]. The dataset taken for training and testing purpose is PROMISE dataset[10].

C. *Loss Function and Optimizer*

Categorical cross entropy function is used to calculate loss for backward propagation. Adam Optimizer is used to for updating weights of the model.

IV. EVALVUATION PARAMETERS

Following Evaluation parameter used to examine the performance of the deep learning model.

A. *Mean Squared Error (MSE)*

It is the average of sum of squared differences between the measured values and "true" values.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE	= mean squared error
n	= number of data points
Y_i	= observed values
$\hat{Y}_i$	= predicted values

B. *Mean Absolute Error (MAE)*

It is the average of sum of differences between the measured values and "true" values.

$$\text{MAE} = \frac{\sum_{i=1}^{n}|y_i - x_i|}{n}$$

MAE	= mean absolute error
y_i	= prediction
x_i	= true value
n	= total number of data points

C. *Accuracy*

The accuracy of a classifier is given as the percentage of total correct predictions divided by the total number of instances.

Accuracy = (True Positive + True Negative)/(Total)

V. RESULTS

The following Results are obtained when the training is done on 30 epochs and batch size is variable (depends on the software module dataset):

Software Module	MAE	MSE	Accuracy
Log	0.03807	0.02282	0.80769
Camel	0.00819	0.00428	0.91182
Velocity	0.03586	0.01932	0.79170
Poi	0.04013	0.02145	0.81254
Ant	0.02247	0.01226	0.90437
Prop	0.00692	0.00348	0.91146
Synapse	0.03661	0.02066	0.85680
Jedit	0.00254	0.00491	0.88251
Xalan	0.04899	0.02889	0.80647
Xerces	0.02776	0.01698	0.87562
Ivy	0.00681	0.00374	0.90122
Full Promise Dataset	0.02805	0.01704	0.87740

Table 1. Results when dataset is divided in 70:30 Ratio

Software Module	MAE	MSE	Accuracy
Log	0.02802	0.01657	0.86538
Camel	0.00648	0.00351	0.92443
Velocity	0.04058	0.02834	0.75706
Poi	0.03875	0.02482	0.85267
Ant	0.01680	0.01374	0.91803
Prop	0.00700	0.00355	0.87975
Synapse	0.03531	0.02389	0.85328
Jedit	0.00227	0.00411	0.90142
Xalan	0.03962	0.01952	0.86459
Xerces	0.01159	0.00597	0.89233
Ivy	0.00668	0.00392	0.91582
Full Promise Dataset	0.01146	0.00591	0.89900

Table 2. Results when dataset is divided in 80:20 Ratio

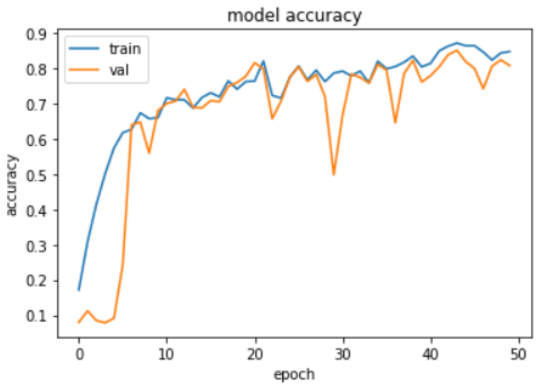

Figure 1. Graph between accuracy and epochs

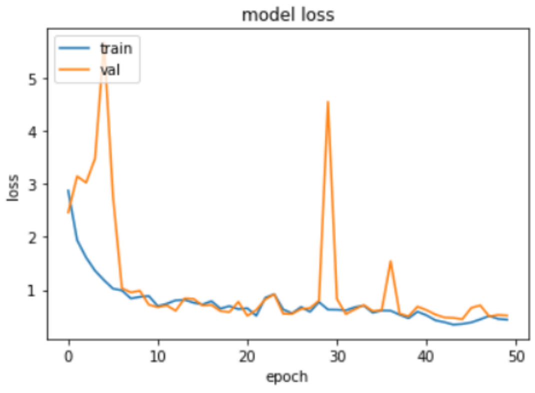

Figure 2. Graph between model loss and epochs

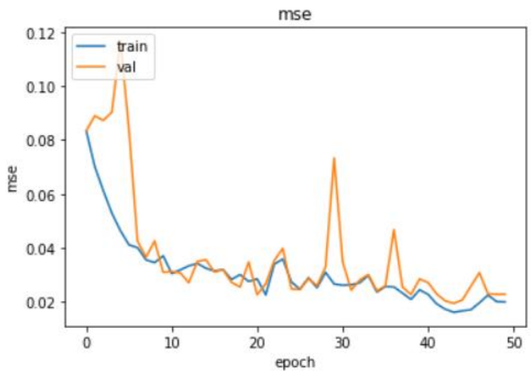

Figure 3. Graph between mse and epochs

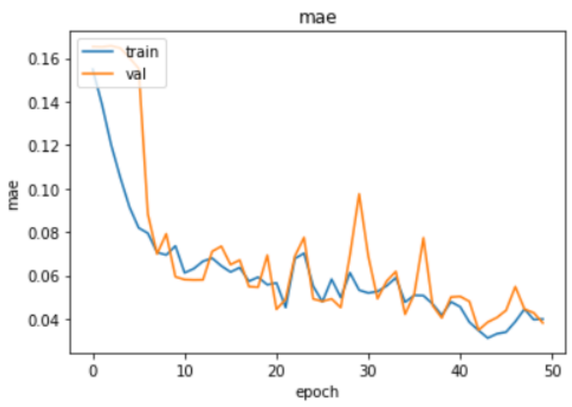

Figure 4. Graph between mae and epochs

V. Conclusion

Following Conclusion can be derived from implementing the paper:

1. Value of hyperparameters (batch size) varies with the choice of training dataset.
2. Temporary spikes in validation loss are normal during training. This means the current local minima doesn't generalize as well as the local minima at the end of the prior epoch.
3. Form table 1 and 2 we can see that model gives best result on 80:20 ratio than on 70:30 (training to testing dataset ratio.)
4. Result may vary on the choice of sampling strategy. 'minority' strategy results in worst result and 'not majority' results in best results.
5. A best accuracy of 0.899 (on 80:20 Full Promise Dataset) shows that our model is overall giving good prediction (finding number of bugs) on testing dataset.

References

[1] Cheikhi, Laila & Al-Qutaish, Rafa & Idri, Ali & Sellami, Asma. (2014). Chidamber and Kemerer Object-Oriented Measures: Analysis of their Design from the Metrology Perspective. International Journal of Software Engineering and Its Applications. 8. 359-374.

[2] D'Ambros M., Lanza M., and Robbe R., "Evaluating Defect Prediction Approaches," Empirical Software Engineering, An International Journal, vol. 17, no. 4-5, pp. 531-577, 2012.

[3] D'Ambros M., Lanza M., and Robbe R., "An Extensive Comparison of Bug Prediction Approaches," in Proceedings of 7 th IEEE Working Conference on Mining Software Repositories, Cape Town, pp. 31-41, 2010.

[4] Jiang Y., Cukic B., Menzies T., and Lin J., "Incremental Development of Fault Prediction Models," International Journal of Software Engineering and Knowledge Engineering (World Scientic Publishing Company), vol. 23, no. 10, pp. 1399-1425, 2013.

[5] Hassan A., "Predicting Faults Using the Complexity of Code Changes," in Proceedings of the 31 st International Conference on Software Engineering, Vancouver, pp. 78-88, 2009.

[6] Hassan A. and Holt R., "The Top Ten List: Dynamic Fault Prediction, " in Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, pp. 263-272, 2005

[7] Menzies T. and Marcus A., "Automated Severity Assessment of Software Defect Reports," in Proceedings of IEEE International

Conference on Software Maintenance, ICSM. Software Maintenance, Beijing, pp. 346-355, 2008

[8] Gulli, A. & Pal, S., 2017. Deep learning with Keras, Packt Publishing Ltd.

[9] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[10] Sayyad Shirabad, J. and Menzies, T.J. (2005) The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada . Available: http://promise.site.uottawa.ca/SERepositor