Software Bug Count Prediction using Features from Abstract Syntax Trees and Object-Oriented metrics

Report submitted in fulfillment of the requirements for BTech Project

Fourth Year B.Tech.

by

Bavanya (18075034)
Swati S Malik(18075060)

Under the guidance of

Dr.Amrita Chaturvedi

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY (BHU) VARANASI
Varanasi 221005, India
December 2021

Dedicated to

Our parents, teachers, and the almighty god.....

Declaration

We certify that

1. The work contained in this report is original and has been done by ourselves and the general supervision of our supervisor.

2. The work has not been submitted for any other project.

3. Whenever we have used materials (data, theoretical analysis, results) from other sources, we have given due credit to them by citing them in the text of the thesis and giving their details in the references.

4. Whenever we have quoted written materials from other sources, we have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Place: IIT (BHU) Varanasi
Date: 6/12/2021

Bavanya and Swati Malik
B.Tech
Department of Computer Science and Engineering,
Indian Institute of Technology (BHU) Varanasi,
Varanasi, INDIA 221005.

Certificate

This is to certify that the work contained in this report entitled ***" Software Bug Count Prediction using Features from Abstract Syntax Trees and Object-Oriented metrics"*** *being submitted by* ***Bavanya(18075034) and Swati S Malik(18075060)*** *, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology (BHU) Varanasi, is a bona fide work of my supervision.*

Place: IIT (BHU) Varanasi
Date:6/12/2021

Dr. Amrita Chaturvedi
Department of Computer Science and Engineering,
Indian Institute of Technology (BHU) Varanasi,
Varanasi, INDIA 221005.

Acknowledgments

We would like to express our sincere gratitude to our supervisor Dr.Amrita Chaturvedi for their constant guidance and support during the whole project work.

Place: IIT (BHU) Varanasi **Bavanya and Swati**

Date:6/12/2021

Abstract

As multiple versions of the software are released to fulfil the ever-changing requirements, bug prediction systems evolved to aid developers in prioritizing their testing tasks. Earlier works classified the modules into faulty/not-faulty categories or performed multi-class classification to predict the bug count. While few used Object-Oriented (OO) metrics for their bug prediction systems, others extracted features from code through Abstract Syntax Trees (ASTs). This paper considered it a regression problem to predict the bug count and applied deep learning models to solve it. We compared the results of our LSTM and CNN models trained on OO metrics with five classical machine learning models and a multilayer perceptron model, and our LSTM model outperformed three of them in MAE and all of them in MRE. We also trained our LSTM and CNN models on features extracted from file-level ASTs of projects' source code and compared them with our LSTM and CNN models trained on OO metrics. Our CNN model trained on features extracted from file-level ASTs of projects' source code gave comparable MAE results with our LSTM trained on OO metrics but outperformed it in MRE.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Thoroughly carrying out the testing phase in the Software development life cycle (SDLC) will help reduce maintenance costs, often more than the development costs themselves[1]. Also, with the increasing complexity of software systems nowadays, it has become time-consuming to manually identify the locations and intensity of bugs in case of a system failure. Software bug count prediction models address these issues. A predictive system can be run on the software system before starting the testing phase[7]. Then testing and debugging tasks can be assigned to developers after analyzing the available resources and the model's predictions[12].

Prediction models can be built at different granularities like method-level, class-level, file-level, package-level and change-level[2]. Multiple versions of a software system are designed to meet the ever-evolving hardware, software and user requirements. So, prediction models can be developed for cross-version or within-version predictions. Moreover, the models can also be developed with either a within-project or cross-project prediction objective[23]. Under these multiple options, we focused on within-project and file-level bug count prediction using open-source labelled datasets.

For software bug count prediction, deep learning models can be trained to learn the characteristics of the buggy code files within specific versions of projects. Suffi-

cient data must be available for training. Once they learn the characteristics properly, they can predict the bug count in the unseen code file that will undergo testing[7]. Chidamber and Kemerer[3], Henderson-Sellers[4], Martins[5], QMOOD[6], and Tang et al., 1999[7] designed several features to identify such characteristics. These metrics extracted from massive open-source projects are publicly available as the PROMISE dataset with the bug count labels. Other recent works used abstract syntax trees (ASTs) to extract syntactical and semantic features from the source codes instead of relying on handcrafted features (Object-Oriented metrics)[2]. This approach was better at characterizing the code for bug detections[2]. The Simplified PROMISE Source Code (SPSC) dataset is publicly available[2]. This dataset includes the features extracted through the ASTs from the code files of the same software projects used to prepare the PROMISE dataset.

We proposed LSTM and CNN model architectures and trained them on the PROMISE dataset. We then compared their performance with other models like Linear Regression, Multilayer perceptron, Decision Tree Regression, Genetic Programming, -ve Binomial Regression and zero-inflated Poisson Regression whose results are documented in Tang et al., 1999 [7]. After that, we have trained our proposed LSTM and CNN models on the SPSC dataset. The MAE and MRE results of our LSTM and CNN models trained on two different datasets are analyzed and compared.

Our contributions in the paper are as follows:

1. To the best of our knowledge, no work has been done on using advanced deep learning models like LSTMs and CNNs for regression to predict the count of bugs in software programs till date. This paper compares their performance with existing works that developed MLP and other classical ML models for the same problem.

2. Since the existing works in bug count prediction only used Object-Oriented metrics to train models for bug count prediction, we have also trained our LSTM and CNN models on features extracted from code through Abstract Syntax Trees (ASTs),

which include both syntactic and semantic features. 3. Our work paves the way to understanding how deep learning can be leveraged for bug count prediction in software programs through the regression approach and how proper code representations and embedding techniques can improve the predictions.

Chapter 2

Related Work

Related works for the following aspects are discussed in this section: regression techniques for software bug count prediction, LSTMs trained for binary or multi-class classification for bug prediction, binary classification using CNNs for software bug detection, and deep learning models trained on features extracted from ASTs.

Classical machine learning techniques like Ridge and Lasso regression[9], Adaboost with different base learners like Random Forest, Extra Tree, Gradient Boosting and eXtreme Gradient Boosting[8], Negative Binomial Regression[10], Decision Tree Regression[11] were explored in previous works of software bug count prediction. Also, BCV-predictor[12] was architectured using LSTM for multi-class classification to predict the count of bugs on the PROMISE dataset previously. Another work performed binary classification using a tree-structured LSTM trained on features extracted from file-level ASTs for detection of the presence of bugs[13]. Code tokenization followed by an LSTM classifier for vulnerability prediction was also presented previously[14]. We also examined the survey done on software defect prediction using deep learning in Akimova et al. [15]. It analyzed different models like deep belief networks, LSTM, CNNs, transformer models and other networks like stacked denoising autoencoders model, Siamese parallel fully connected networks, deep forest models and graph neural

network for binary classification of files as faulty or not[15].

To the best of our knowledge, no paper has presented deep learning models that treated the software bug count prediction as a regression problem till date. The closest works were performing multi-class classification to predict the probability of the different bug counts for a code file in a software system[12]. We decided to experiment with the regression approach using deep learning because of the following limitation to the multi-class classification, which our approach overcomes. The multi-class classification models will not predict the accurate no. of bugs if the inference is run for a code with more bugs than the maximum bug count value in the multi-class.

After studying the related works, we have built an LSTM regression model for software bug count prediction. We considered the LSTM model because it can store both the long and short-range context information from the source code due to its memory cell[15]. Also, few works have shown how the features extracted from ASTs are effective for code completion and bug detection[2]. Previous works also discussed the details of how the code semantics are buried in the ASTs[16]. Software defect prediction was also made via an attention-based RNN on the features extracted from ASTs earlier[17].

Hence, we have also used the features extracted from ASTs to predict the count of bugs. We compared that approach with hand-crafted features, i.e., object-oriented metrics, to determine the usefulness of AST extracted features for software bug count prediction.

We have also studied the previous works about how the CNNs successfully outperformed traditional feature-based approaches for software defect prediction[2, 18, 19] and hence we chose CNN to train on the features extracted from ASTs. Another motivation for choosing the CNN model is their excellent performance on grid-like data, which is the case after our input passes an embedding layer[2]. Our embedding layer generates a three-dimensional tensor from the features extracted from ASTs to

prepare the data for our CNN and LSTM models. Our embedding layer is based on the skip gram model and it increases the dimensions of the input data.

Chapter 3

Datasets and Performance Measures

3.1 Datasets

A previously published survey presents all the open-sourced labelled and unlabelled software defect prediction datasets [15]. We chose the following two sources from the publicly available datasets with the bug count labels:

3.1.1 PROMISE Repository

We selected eight projects from this repository and trained our models. Each instance of a project corresponds to a code file of the project. The bug count label and 20 specific OO metrics were provided for each software project class. Description and Explanation of all the 20 metrics are given in Table 3.1 and Table 3.2.

Detailed description of the datasets is given in Table 3.3 below:

Among the eight projects taken, the percentage of faulty modules range from 13.71% to 48.25%. Ant, Camel, Xalan and Xerces are open-sourced by Apache and written in C++ and Java.

Table 3.1 Dataset Features

Abbreviation	Dataset Features
CBO	Coupling between objects
RFC	Response for a class
LCOM	Lack of cohesion in methods
NOC	Number of children
DIT	Depth of inheritance
WMC	Weighted methods per class
Ca	Afferent couplings
Ce	Efferent Couplings
NPM	Number of public methods
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
AMC	Average Method Complexity
LOC	Line of Code
CBM	Coupling Between Methods
IC	Inheritance Coupling

Table 3.2 Explanation of the features

Abbreviation	Explanation
CBO	counts the number of other classes to which a class is coupled with.
RFC	counts the number of external and internal classes.
LCOM	measures dissimilarity of methods in a class.
NOC	counts the number of descendants of a class.
DIT	measures number of ancestor classes.
WMC	counts the number of methods in a class weighted by complexity.
Ca	counts how many other classes use a given class.
Ce	counts the number of classes. a class is dependent upon.
NPM	counts the number of public methods in a class.
DAM	the number of private methods divided by the total number of methods.
MOA	counts the number of abstract data types in a class.
MFA	the number of inherited methods divided by total number of methods accessible by its member functions.
CAM	based upon the parameters in a method.
AMC	counts average size of method in a class.
LOC	counts the number of lines of source code.
CBM	counts the newly added func- tions with which inherited based methods are coupled.
IC	it is based upon inheritance-based coupling.

Table 3.3 Description of the datasets chosen from PROMISE.

Project Name	Version	No. of modules	No. of faulty modules	% of faults
ant	1.7	745	166	22.28
camel	1.2	608	216	35.53
camel	1.4	872	145	16.63
camel	1.6	965	188	19.48
xalan	2.4	723	110	15.33
xalan	2.5	803	387	48.25
xalan	2.6	885	411	46.5
xerces	1.3	502	68	13.71

3.1.2 SPSC datasets

These datasets were prepared by parsing the source code of the open-source projects into ASTs and generating token vectors using the specific node types of the parsed ASTs. The node types selected from the ASTs presented in work by Pan et al., 2019[2] are given in Table 3.4 below.

Table 3.4 Special node types of the parsed ASTs.

Formal Parameter	ForStatement
Basic Type	AssertStatement
Interface Declaration	BreakStatement
CatchClauseParameter	Continue Statement
ClassDeclaration	ReturnStatement
MethodInvocation	Throw Statement
SuperMethodInvocation	SynchronizedStatement
MemberReference	TryStatement
ConstructorDeclaration	SwitchStatement
Reference Type	BlockStatement
Method Declaration	TryResource
Variable Declarator	CatchClause
IfStatement	SwitchStatementCase
WhileStatement	ForControl
Do Statement	

As per Pan et al., 2019[2], ASTs were generated using the javalang tool. After generating the ASTs, a list of features are extracted from them. For example, a sample java program which generates and prints the sum of elements in args list is given below.

```
public class CommandParameter{
public static void main(String args[])
int sum=0;
for(String arg:args[])
{
for(int i=0;i<args.length;i++)
{
sum=sum+Integer.parseInt(args[i]);}
System.out.println(sum):
}
```

```
}
```

Table 3.5 Parsed AST tokens of the sample java program

CommandParameter	int
1-2 main	i
args	i
String	length
int	i
sum	block
for	sum
String	sum
arg	parselnt
args	args
block	i
for	printin
forControl	sum

In Pan et al., 2019[2], the tokens in Table 3.5 are generated for the sample java program given in Fig 1 by considering only the special AST node types given in Table 3.4. To map these tokens from strings to integers, every token string is given an index which ranges from one to total number of unique tokens. The integer input vectors are mapped through these indexes from the extracted keyword tokens. Since all data instances in a project do not have the same token vector size, zeros were appended at the end of the shorter tokens to obtain a uniform size[2]. Eight projects are selected, and their detailed description is given in Table 3.6 below:

Table 3.6 Description of the datasets chosen from the SPSC dataset

project	version	tokens length	no of modules	no of faulty modules	% of faults
ant	1.7	3378	741	166	22.4
camel	1.2	1765	595	216	36.3
camel	1.4	2094	847	145	17.12
camel	1.6	2094	934	188	20.13
xalan	2.4	7367	676	110	16.27
xalan	2.5	7389	754	379	50.27
xalan	2.6	7389	875	411	46.97
xerces	1.3	7543	446	67	15.02

3.2 Performance Measures

We have chosen the following metrics to evaluate the performance of our models:

3.2.1 Mean Absolute Error (MAE)

It is used to show the absolute mean of the errors in the predictions compared with the labels. However, this metric will not capture whether the prediction value is greater or lesser than the label.

It is calculated using the following equation:

$$MAE(y, \widehat{y}) = \frac{1}{m} \sum_{i=0}^{m-1} \left| (yi - \widehat{yi}) \right|$$

Where yi is the predicted value of the ith sample and yi is its label. m is the number of modules in the prediction set.

3.2.2 Mean Relative Error (MRE)

MRE gives the average ratio of the absolute error to the actual label. This metric is useful when dealing with labels that are large numerical values.

It is calculated using the following equation:

$$MRE(y, \widehat{y}) = \frac{1}{x} \sum_{i=1}^{x} \frac{\left| \widehat{yi} - yi \right|}{yi + 1}$$

Where yi is the predicted value of the ith sample and yi is its label. x is the number of modules in the prediction set.

Since our labels are whole numbers, we have considered (yi+1) in the denominator to avoid the 0 division problem.

Chapter 4

Experimental Methodology

We performed the train test split at a 4:1 ratio after collecting the data instances. We used the Keras library over the Tensorflow backend. Tensorboard and Matplotlib were used to get the graphical visualizations for the experiments. To maintain the generality of our models, we have run each experiment 10 times and averaged our evaluation results, i.e. the same model was trained and tested on the same dataset 10 times where the experimental hyperparameters remained unchanged. We used Mean

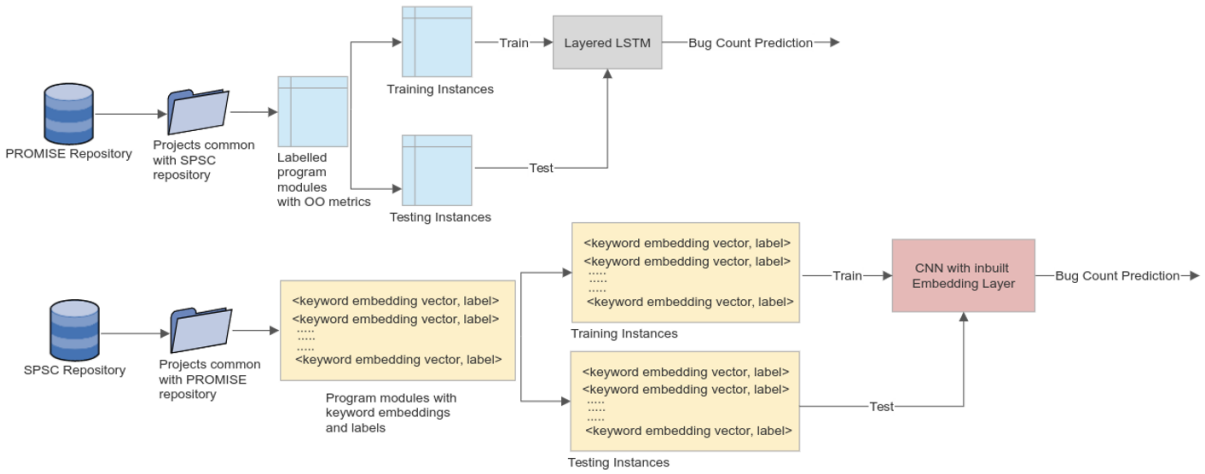

Figure 4.1 Workflow of our experiments on projects from PROMISE repository and SPSC datasets.

squared error as our loss function and used Adam as our gradient-descent optimization algorithm. The models in all the experiments were trained for 100 epochs, and 100 was taken as the batch size. The appropriate batch size was determined after initial exploration. Our labels were whole numbers, and predictions of our models were either zero or positive floating numbers.

Experiments were conducted to answer the following two research questions:
1. RQ-1: Does a deep learning model perform better than traditional machine learning techniques in predicting the count of bugs in software modules?
2. RQ-2: Does a deep learning model trained on features extracted from ASTs perform better than a deep learning model trained on Object-Oriented metrics?

We trained LSTM and CNN models on Object-Oriented metrics to answer the first question and compared their performance with our baseline models in Tang et al., 1999[7]. Our baseline models (Linear Regression, Multilayer perceptron, Decision Tree Regression, Genetic Programming, -ve Binomial Regression and zero-inflated Poisson Regression) are classical machine learning. For the second question, we compared the performance of our LSTM and CNN, which were trained on projects from the PROMISE repository, with our LSTM and CNN models trained on the same projects from the SPSC dataset.

We have used the dropout regularization method for our LSTM method. We used an embedding layer built with the Embedding class of Keras layers API to prepare the features extracted from ASTs for training by our LSTM and CNN models. The embedding layer converts a two-dimensional tensor of (batch size, input length) shape to a three-dimensional tensor of (batch size, input length, output dim) shape. We noticed that taking the output dim as 30 as taken in the Fan et al., 2019[17] paper led to significant overfitting of our model. Hence after tuning, we took six as its value.

Below we give a detailed description of our models' architectures.

4.1 Architecture of the layered LSTM

Our model consists of four LSTM hidden layers and a fully connected layer. The first, second, third, and fourth hidden layers have 200, 100, 50, and 20 hidden units. Drop out rate of 0.2 is considered for the third layer. ReLu activation is added after the fully connected layer to predict the bug count. Fig 4.2 illustrates the overall architecture.

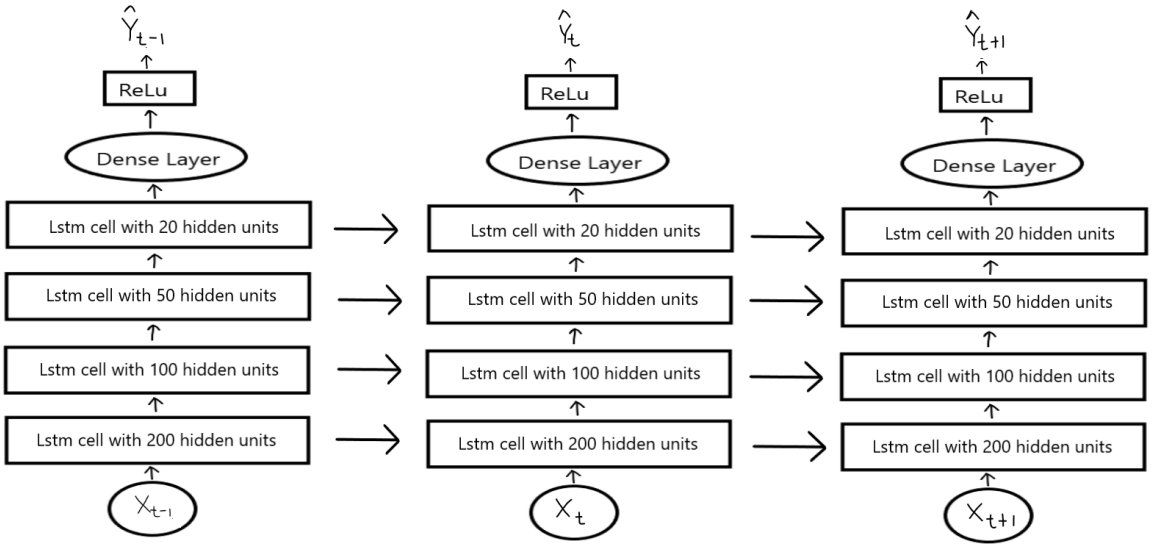

Figure 4.2 Illustration of the proposed LSTM model

The ReLu activation function is as follows: f(x) = 0 for x /¿ 0 and f(x) = x for x /¿= 0. We took ReLu because the bug count value is always greater than or equal to zero.

4.2 Architecture of the CNN model

Fig 4.3 gives the overall architecture. The first layer of our CNN is a convolutional layer. After that, we added a flattening layer and a dense layer with ReLu activation to

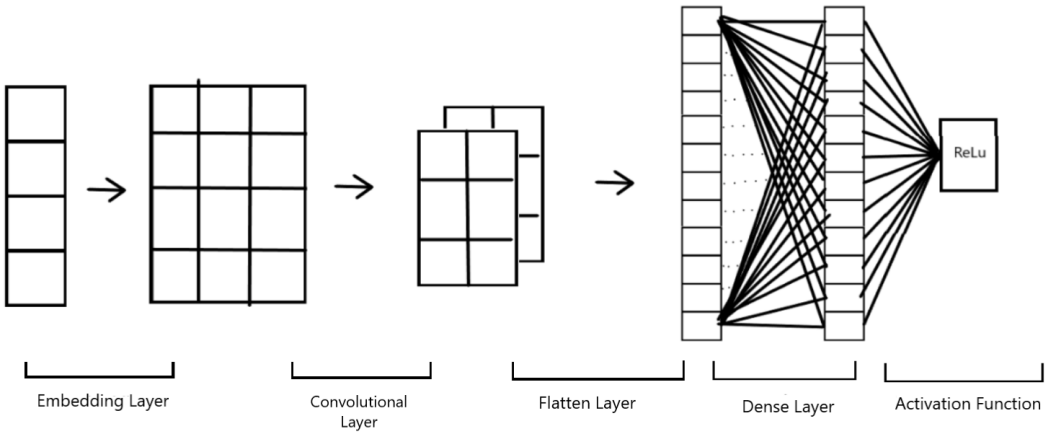

Figure 4.3 Illustration of the proposed CNN model.

get the bug count predictions. Even the convolutional layer used the ReLu activation function. For the convolutional layer, we took two as the kernel size.

Chapter 5

Results and Comparative Analysis

5.1 Justification of RQ-1

The MAE and MRE results of our LSTM and CNN trained on Object-Oriented metrics compared to the baseline models are given in Table 5.1. They are also graphically represented in Fig 5.2. The results of the baseline models are taken from the Tang et al., 1999[7] paper.

Table 5.1 MAE results for RQ-1

	LR	MLP	DTR	GP	-ve BR	0-inflated PR	Our LSTM model	Our CNN model
ant1.7	**0.38**	0.53	0.39	0.61	0.97	1.37	0.506	3.322
camel1.2	1.07	0.98	1.01	1.02	0.94	**0.81**	1.146	1.505
camel1.4	0.48	0.63	0.43	0.62	0.84	0.66	0.523	**0.378**
camel1.6	0.67	0.92	**0.63**	0.7	0.93	0.84	0.648	1.352
xalan2.4	**0.23**	0.33	0.25	**0.23**	1.18	0.92	0.313	2.582
xalan2.5	0.54	0.75	0.56	**0.49**	0.79	0.78	0.611	2.833
xalan2.6	0.53	0.63	**0.52**	0.68	1.01	1.06	0.654	1.187
xerces1.3	0.56	0.74	**0.34**	0.46	1.24	0.96	0.483	3.009

From Table 5.1 and Fig 5.1, we can see that our LSTM model gave better MAE results than zero-inflated Poisson Regression and -ve Binomial Regression and gave comparable results with Genetic programming. However, it could not outperform Decision Tree Regression, MLP and Linear Regression models. In comparison, our CNN model gave very poor results for all the projects except camel1.4 when trained on Object-Oriented metrics. Surprisingly, our CNN model surpassed all the models

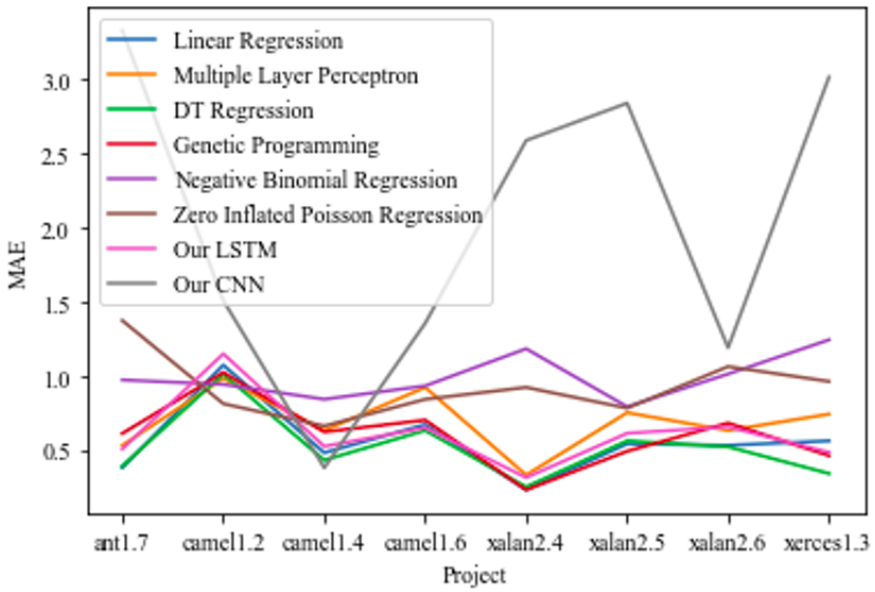

Figure 5.1 MAE results of our LSTM, CNN and the baseline models from Tang et al., 1999

when trained on camel1.4.

In below Table 5.2, we also present the MRE results of our LSTM and CNN models with the baseline models. They are also graphically represented in Fig 5.2.

Table 5.2 MRE Results for RQ-1

	LR	MLP	DTR	GP	-ve BR	zero-inflated PR	Our LSTM model	Our CNN model
ant1.7	0.2	0.27	0.2	0.4	0.88	1.1	**0.191**	1.34
camel1.2	0.61	0.51	0.56	0.6	0.6	0.57	**0.299**	0.375
camel1.4	0.27	0.38	0.24	0.41	0.73	0.5	**0.127**	0.134
camel1.6	0.4	0.64	0.35	0.41	0.78	0.68	**0.193**	0.624
xalan2.4	**0.12**	0.22	0.14	0.16	1.08	0.81	0.181	0.302
xalan2.5	0.39	0.51	0.4	**0.34**	0.55	0.56	0.399	1.186
xalan2.6	0.33	0.36	**0.3**	0.45	0.71	0.6	0.315	0.843
xerces1.3	0.39	0.5	0.2	0.33	1.16	0.77	**0.122**	0.474

From Table 5.2 and Fig 5.2, we observe that our LSTM model's MRE results surpassed those of all the other models. On the other hand, our CNN model gave better MRE results than zero-inflated Poisson Regression and -ve Binomial Regression and was comparable to MLP.

Hence, we conclude that deep learning models for bug count prediction using regression are better than a few machine learning models like zero-inflated Poisson Regression and -ve Binomial Regression and comparable to Genetic programming.

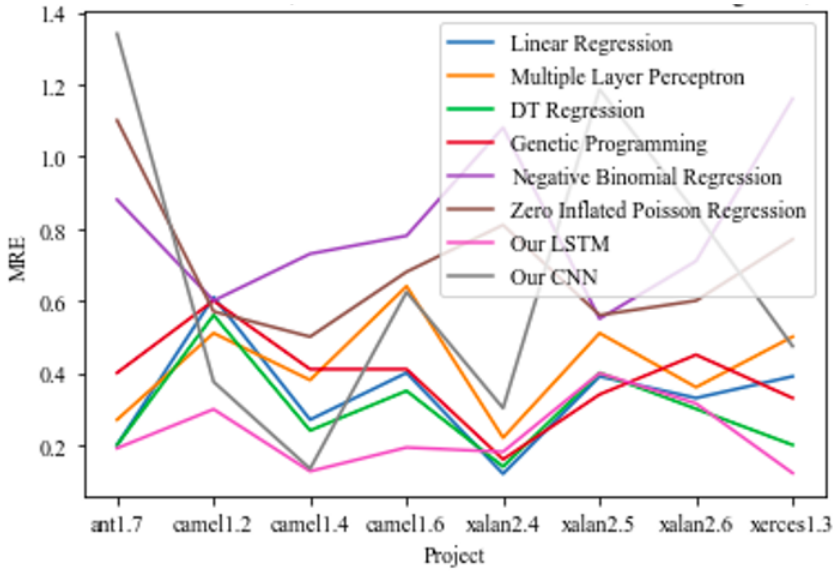

Figure 5.2 MRE results of our LSTM, CNN and the baseline models from Tang et al., 1999

However, other machine learning models like Linear Regression and DT Regression are better performing than deep learning models like LSTM and CNN.

5.2 Justification of RQ-2

We compared the results of our LSTM and CNN models trained on Object-Oriented metrics with our LSTM and CNN models trained on features extracted from ASTs, and the results are given in Table 5.3 below. Graphical representation of the results is shown in Fig 5.3 and Fig 5.4.

Table 5.3 MAE and MRE Results for RQ-2

	MAE				MRE			
	Object-Oriented Metrics (OO)		Features from ASTs		Object-Oriented Metrics (OO)		Features from ASTs	
	Our LSTM	Our CNN	Our LSTM	Our CNN	Our LSTM	Our CNN	Our LSTM	Our CNN
ant1.7	0.506	3.322	0.594	**0.391**	0.191	1.34	0.214	**0.115**
camel1.2	1.146	1.505	0.975	**0.816**	0.299	0.375	0.435	**0.168**
camel1.4	0.523	**0.378**	0.639	0.598	**0.127**	0.134	0.18	0.149
camel1.6	**0.648**	1.352	0.934	0.657	0.193	0.624	0.283	**0.124**
xalan2.4	0.313	2.582	**0.3**	0.384	0.181	0.302	0.183	**0.176**
xalan2.5	**0.611**	2.833	0.71	0.658	0.399	1.186	0.509	**0.264**
xalan2.6	**0.654**	1.187	0.723	0.669	**0.315**	0.843	0.468	0.382
xerces1.3	0.483	3.009	0.432	**0.421**	**0.122**	0.474	0.166	0.134

From Table 5.3 and Fig 5.3 and Fig 5.4, we observe that the performance of our

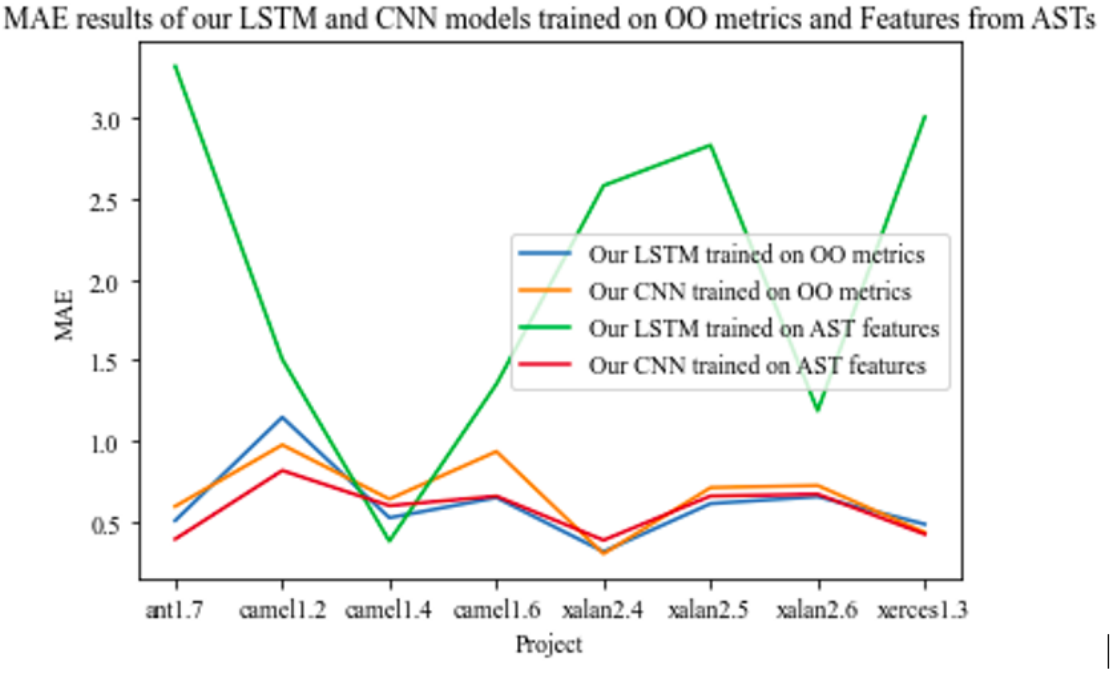

Figure 5.3 MAE results of our LSTM and CNN models trained on Object-Oriented metrics and Features from ASTs.

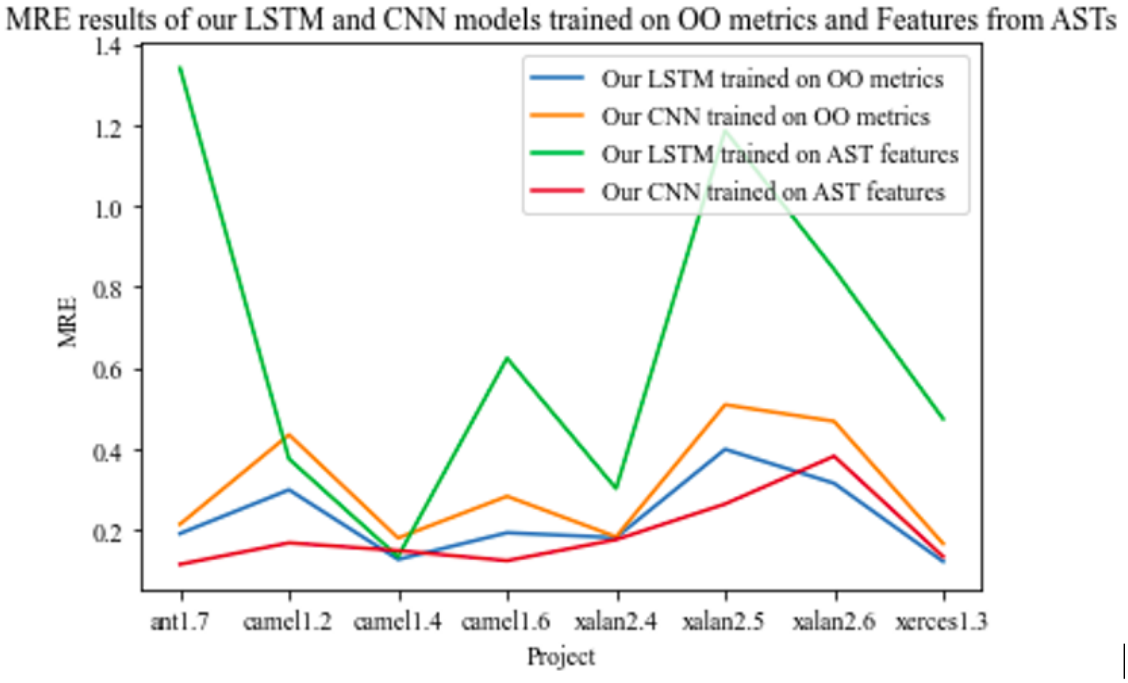

Figure 5.4 MRE results of our LSTM and CNN models trained on Object-Oriented metrics and Features from ASTs.

LSTM model trained on Object-Oriented metrics is comparable to the performance of our CNN model trained on features extracted from ASTs. We also observe that our LSTM model trained on Object-Oriented metrics performed better compared to when it was trained on features extracted from ASTs. However, our CNN model performed better when trained on features extracted from ASTs than when trained on Object-Oriented metrics.

Therefore we conclude that features extracted from ASTs are better suited for CNN models than Object-Oriented metrics, whereas it is recommended to prefer Object-Oriented metrics to train LSTM models for Software Bug Count prediction through regression.

Chapter 6

Threats to Validity

The projects used for the bug count prediction in our experiments were written in C++ and Java. Hence we may not validate the experimental results when our models and approaches are applied to projects written in other languages like C and Python. Since we have used open-source datasets instead of creating our own from the projects' source codes, the quality of our results is based on the quality of the datasets used. Also, from the datasets' descriptions, it is noticed that the number of modules in each dataset is not the same for the PROMISE dataset and the SPSC dataset, which might add a systematic bias in our results. The difference in the number of modules in both datasets varies from four to fifty six. We have only used MAE and MRE as our evaluation metrics for comparing the results of the two approaches. Hence we cannot validate the evaluation results of other metrics.

Chapter 7

Conclusion and Future Work

We designed the LSTM and CNN models for the bug count prediction problem and compared their performance when trained on Object-Oriented metrics with other baseline models. From this comparison, we conclude that though our deep learning models surpassed models like zero-inflated regression and negative binomial regression, models like linear regression and decision tree regression still outperformed them. Also, we conclude that our CNN model trained on features extracted from ASTs gave results comparable to our LSTM trained on Object-Oriented metrics.

In the future, we would like to try different code representation models like codeBERT[21] developed by Microsoft to get better embeddings of the keywords extracted from the ASTs. Another extension to using codeBERT could be to use GraphCodeBERT[22] and compare the two techniques for the bug count prediction problem. Instead of only CNN, we can also build other models like RNN to obtain better results.

Bibliography

[1] W. W. Xiaoxing Yang, *Ridge and Lasso Regression Models for Cross-Version Defect Prediction*, vol. 67 NO.3, 2018.

[2] C.-P. C. Ko-Li Cheng and C.-P. Chu, *"Software fault detection using program patterns"2011 IEEE 2nd International Conference on Software Engineering and Service Science*, 2011.

[3] K. M. T. F. T. Liu and Z. Zhou, *"Isolation Forest," 2008 Eighth IEEE International Conference on Data Mining*, 2008.

[4] B. X. Cong Pan, Minyan Lu and H. Gao, *"An Improved CNN Model for Within-Project Software Defect Prediction"*, 2019.

[5] X. L. Hamza Turabieha, Majdi Mafarja, *"Iterated feature selection algorithms with layered recurrent neural network for software fault prediction"*, 2019.

[6] S. W. N. T. T. V. G. Hoa Khanh Dam, Trang Pham and A. Ghose, *"A deep tree-based model for software defect prediction"*, 2018.

[7] J. Schmidhuber, *"Deep learning in neural networks: An overview"*, 2014.

[8] L. O. H. W. P. K. Nitesh V. Chawla, Kevin W. Bowyer, *"SMOTE: Synthetic Minority Over-sampling Technique"*, 2002.