DEEP LEARNING FOR STATEMENT-LEVEL SOFTWARE DEFECT PREDICTION

Dissertation submitted in partial fulfillment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

INFORMATION TECHNOLOGY

By

BONTHU SAHITA	18VV1A1209
BOKKA SOWMYA SREE	18VV1A1207
BANGARI PRAVEEN PETER	18VV1A1204
SATHIVILLI SANDHYA RANI	19VV5A1273

Under the esteemed Guidance of

Dr.B. TIRIMULA RAO, M.Tech

Assistant Professor

Department of Information Technology

DEPARTMENT OF INFORMATION TECHNOLOGY

JNTUGV -COLLEGE OF ENGINEERING

VIZIANAGARAM-535003, A.P, INDIA

MAY-2022

DEEP LEARNING FOR STATEMENT-LEVEL SOFTWARE DEFECT PREDICTION

Dissertation submitted in partial fulfillment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

INFORMATION TECHNOLOGY

By

BONTHU SAHITA	18VV1A1209
BOKKA SOWMYA SREE	18VV1A1207
BANGARI PRAVEEN PETER	18VV1A1204
SATHIVILLI SANDHYA RANI	19VV5A1273

Under the esteemed Guidance of

Dr.B. TIRIMULA RAO, M.Tech

Assistant Professor

Department of Information Technology

DEPARTMENT OF INFORMATION TECHNOLOGY

JNTUGV -COLLEGE OF ENGINEERING

VIZIANAGARAM-535003, A.P, INDIA

MAY-2022

Department Of Information Technology

JNTUGV -COLLEGE OF ENGINEERING

VIZIANAGARAM-535003, A.P, INDIA

CERTIFICATE

This is to certify that dissertation entitled — "**DEEP LEARNING FOR STATEMENT-LEVEL SOFTWARE DEFECT PREDICTION**" submitted by **B. SAHITA (18VV1A1209), B.SOWMYA SREE (18VV1A1207), B. PRAVEEN PETER (18VV1A1204) S.SANDHYA RANI (19VV5A1273),** in partial fulfillment of for the award degree of BACHELOR OF TECHNOLOGY in the department of INFORMATION TECHNOLOGY from Jawaharlal Nehru Technological University - Gurajada Vizianagaram is a Record of Bonafide work carried out by them under my guidance and supervision during the year 2021-2022. The results embodied in this dissertation have not been submitted by any other University or Institution for the award of any degree.

Signature of the Guide

Dr.B.TIRIMULA RAO

Assistant Professor

Dept. of Information Technology

JNTUGV-CEV, VIZIANAGARAM

Signature of the Head of the Department

Dr.G.JAYA SUMA

Professor & HOD

Dept. of Information Technology

JNTUGV-CEV,VIZIANAGARAM

DECLARATION

We **B. SAHITA (18VV1A1209), B. SOWMYA SREE (18VV1A1207), B. PRAVEEN PETER (18VV1A1204), S. SANDHYA RANI (19VV5A1273),** declare that the project report entitled — "**DEEP LEARNING FOR STATEMENT-LEVEL SOFTWARE DEFECT PREDICTION**" is no more than 1,00,000 words in length including quotes and exclusive of tables figures, bibliography, and reference. This dissertation contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma. Except where otherwise indicated this project is our work.

ROLLNO	NAME	SIGNATURE
18VV1A1209	BONTHU SAHITA	_____
18VV1A1207	BOKKA SOWMYA SREE	_____
18VV1A1204	BANGARI PRAVEEN PETER	_____
19VV5A1273	SATHIVILLI SANDHYA RANI	_____

DATE:

APPROVAL SHEET

This Report entitled "**DEEP LEARNING FOR STATEMENT-LEVEL SOFTWARE DEFECT PREDICTION**" is approved for the BACHELOR OF TECHNOLOGY in the department of INFORMATION TECHNOLOGY.

PROJECT GUIDE

_____

HEAD OF THE DEPARTMENT

_____

Date:

Place:

ACKNOWLEDGEMENT

This project report could not have been written without the support of our professor **Dr.B.TIRIMULA RAO M.Tech Assistant Professor, Information Technology department** who not only served as our superior but also encouraged and challenged us throughout our academic program. Our foremost thanks go to him. Without him, this project would not have been possible. We appreciate his vast knowledge in many areas and his insights, suggestions, and guidance that helped to shape our research skills.

It is needed with a great sense of pleasure and immense sense of gratitude that we acknowledge the help of these individuals. We owe many thanks to many people who helped and supported us during the writing of this report.

We are thankful to our project coordinator **Dr.B.TIRIMULA RAO**, **Assistant Professor Department of Information Technology,** for his continuous support.

We express our sincere thanks to our respected **Dr.G.JAYA SUMA, Professor and H.O.D of Information Technology of JNTUGV - College of Engineering Vizianagaram,** for her Valuable suggestion and constant motivation that greatly helped us in the successful completion of the project. We also take the privilege to express our heartfelt gratitude to **Prof. R. RAJESWARA RAO, Principal, JNTUGV - College of Engineering Vizianagaram.**

We are thankful to all associate faculty members for extending their kind cooperation and assistance. Finally, we are extremely thankful to our parents and friends for their constant help and moral support.

BONTHU SAHITA	**18VV1A1209**
BOKKA SOWMYA SREE	**18VV1A1207**
BANGARI PRAVEEN PETER	**18VV1A1204**
SATHIVILLI SANDHYA RANI	**19VV5A1273**

TABLE OF CONTENTS

TABLE OF CONTENTS Page.No

ABSTRACT

Software systems have become larger and more complex than ever. Such characteristics make it very challenging to prevent software defects. Therefore, automatically predicting the number of defects in software modules is necessary and may help developers efficiently to allocate limited resources. Various approaches have been proposed to identify and fix such defects at a minimal cost. However, the performances of these approaches require significant improvement. Therefore, we propose a novel approach that leverages deep learning techniques to predict the number of defects in software systems. First, we preprocess a publicly available dataset, including log transformation and data normalization. Second, we perform data modeling to prepare the data input for the deep learning model. Third, we pass the modeled data to a specially designed deep neural network-based model to predict the number of defects. We also evaluate the proposed approach on four well-known datasets. The evaluation results illustrate that the proposed approach is accurate and can improve upon the machine learning approach. In this experiment, our trained model could successfully classify the unseen data (that is, fault- proneness of new statements) with average performance measures in terms of recall, precision, and accuracy, respectively. These experimental results suggest that SLDeep is effective for statement-level SDP. The impact of this work is twofold. Working at the statement-level further alleviates the developer's burden in pinpointing the fault locations. Second, the cross-project feature of SL Deep helps defect prediction research become more industrially viable.

LIST OF FIGURES

LIST OF TABLES

CHAPTER-1

INTRODUCTION

INTRODUCTION

There is an increase in demand for high-quality software in today's developing technology. Humans are relying on software for every task. Thus, the current software has become more complex and costly to develop. Meanwhile, we are still in trouble with low-quality software as it has imposed large economic costs.

To improve software quality and alleviate the underlying costs, an approach has been to construct effective software defect prediction (SDP) techniques. By using faulty historical data along with static code features gathered from previous software versions, SDP techniques seek to develop prediction models. These models are intended to shed light on problematic portions of the code in the next release of the software. This is achieved by predicting (estimating) fault-prone code fragments. Once fault-prone portions are identified, testing and code review activities can focus on them.

The SDP research is of great practical importance as it makes software quality assurance activities more targeted and productive. As a result, high-quality software can be obtained. SDP techniques that exist can be used in the early stages of software development such as requirements engineering, analysis, and design. The current techniques are developed to work at coarse-grained code areas such as a class, module, file, function, or plug-in binaries. We observe that SDP techniques that work at finer granularities such as statements have received little or no attention.

For techniques that rely on defect prediction at higher granularities, developers have to spend non-trivial time detecting and localizing the faults within a module labeled as fault-prone. Fine-grained defect prediction analysis can result in the acceleration of identifying fault locations. We hypothesize that statement-level SDP has much potential to pinpoint fault-prone locations. Consequently, faults would be detected with less time and effort.

To reify our hypothesis, we propose a new technique, which we call Statement-Level defect prediction using Deep-learning models (SLDeep). SLDeep can be used for both within-project and cross-project defect prediction. Cross-project defect prediction can be a viable case for companies as it significantly reduces the effort in data collection. In addition, the existing within-project data might no longer be indicative of the current practices, since practices change over time.

Our goal is to construct a prediction model that can be used to classify a program statement as fault-prone or fault-free. It works on source code defect prediction at the statement level. Also, it is designed neither for usage at the early stages of the software development lifecycle nor for online defect prediction. For statement-level SDP, we first introduce a suite of 32 new code metrics that measure some features of a statement.

In the project, SLDeep could successfully predict fault-prone statements with average performance measures in terms of recall, precision, and accuracy. Based on these results, SLDeep seems to be effective at statement-level software defect prediction and can be adopted. The significance of SLDeep lies in its usage of LSTM for a practical problem. It can lead to saving software development resources and more reliable software.

1.1 Problem Statement

The main objectives of our project are to improve software quality and reduce costs and effort. This is used to alleviate the developer's burden in a pin- pointing out locations of faults, hence providing high-quality software with less time and effort.

1.2 Motivation

SLDEEP: We use SLDEEP model for finding the fault-prone areas of code. Here, we integrate the introduced metric suite with LSTM to develop our proposed SDP system. To train an LSTM model, a matrix so-called M_P should be established per each program P.

The i^{th} row of M_P is devoted to the i^{th} line (statement) of P. The columns of M_P are divided into two broad categories. The first category is devoted to 32 metrics introduced and is

fixed for every program. The second category of columns is devoted to the individual tokens present in each line. This category is of paramount importance as it serves us as a means to capture the code structure itself. The second category can also break any tie introduced using the code metrics in the first category.

To use SLDeep, one needs to compute metric data, extract tokens and construct a matrix for a given program (project). Then an LSTM model is trained and used to predict whether a new unseen statement is either fault-free or fault-prone.

1.3 Project Contribution

Deep learning is a machine learning technique that teaches computers to do what comes naturally to humans. It can achieve state-of-the-art accuracy, sometimes exceeding human-level performance. Models are trained by using a large set of labeled data and neural network architectures that contain many layers. Models are trained by using a large set of labeled data and neural network architectures that contain many layers.

For our SLDeep, we use an effective machine-learning technique, namely long short-term memory (LSTM), which belongs to the deep-learning sub-category. LSTM belongs to the recurrent neural networks (RNN) family. A feed-forward neural network, which is extended to support feedback connections, is called RNN. An LSTM uses self-loops to establish paths in which the gradient can flow for long durations. In addition, the weights of self-loops are not fixed; they are determined depending upon the context. Self-loops serve as internal recurrence within units called LSTM cells. These cells have the same input/outputs as ordinary RNNs.

Using this method and algorithm, based on deep learning which is also based on machine learning requires lots of mathematical and deep learning frameworks understanding by using dependencies such as TensorFlow, Keras, etc., we can detect every fault in the software. This also includes the accuracy of each method for identifying faults.

1.4 Project Organization

The rest of the report is organized as follows. Chapter 2 gives the Review of the Literature; Chapter 3 illustrates the software requirement specification document. Chapter 4 describes the steps involved in our methodology. Chapter 5 gives the introduction to Deep Learning. Chapter 6 contains SLDEEP Architecture. Chapter 7 contains Source code and Chapter 8 has results and discussions. In Chapter 9 we conclude and describe the future work done in brief. And Chapter 10 includes Appendix and bibliography respectively.

CHAPTER-2

REVIEW ON LITERATURE

2.1 Background and Related work

Software Defect Prediction:

Software fault prediction is an important and beneficial practice for improving software quality and reliability. The ability to predict which components in a large software system are most likely to contain the largest numbers of faults in the next release helps to better manage projects, including early estimation of possible release delays, and affordable guide corrective actions to improve the quality of the software. However, developing robust fault prediction models is a challenging task. Traditional software fault prediction studies mainly focus on manually designing features (e.g., complexity metrics), which are input into machine learning classifiers to identify defective code.

However, these features often fail to capture the semantic and structural information of programs. Such information is needed for building accurate fault prediction models. In this project, we discuss various approaches to fault prediction, also explaining how in recent studies deep learning algorithms for fault prediction help to bridge the gap between programs' semantics and fault prediction features and make accurate predictions.

BUGS: There are two types of errors:

- Simple error: If the error can be fixed by changing one faulty statement. The more substantial the fix must be, the more complex we consider the error. As a measure of complexity for error fixes, we introduce Cyclomatic Change Complexity which is inspired by existing program complexity metrics
- Seeded error: the process of deliberately introducing errors within a program to check whether the test cases can capture the seeded errors. We find that seeded errors are significantly less complex, i.e., require significantly fewer substantial fixes, compared to actual regression errors.

We propose Code4Bench for the controlled study of regression testing, debugging, and repair techniques. Code4Bench provides tables, in which the correct and faulty versions for C/*C*++ programs are specified. In addition, the exact faulty lines of defective programs are specified.

Bias in bug-fix datasets:

Bug detect-Software engineering researchers are interested in where and why bugs occur in code. Bug tracking systems, and code version histories, record when, how, and by whom bugs were fixed; from these sources, datasets that relate file changes to bug fixes can be extracted. We investigate historical data from several software projects and find evidence of systematic bias. We then investigate the potential effects of *unfair, imbalanced* datasets on the performance of prediction techniques.

Statement-Level software code metrics:

There exist software code metrics that have been used in different software quality assurance and software defect prediction. The metrics are categorized into various groups, such as object-oriented, function-level, and statement-level.

From the general point of view, each group of the existing source code metrics can be divided into two categories, namely general-purpose and special-purpose. The former category includes software code metrics that are commonly used in studies. The latter category includes software code metrics that are defined for a certain study; they are often customized for the specific context in which they are applied.

Several change metrics are extracted from software repositories, and models are built using machine learning algorithms, namely Decision Tree, K-Nearest Neighbor, and Random Forest.

Software metrics can be classified into the following types:

- Code and complexity metrics: Complexity metrics indicate how complex a code block is, and it is widely known that the more complex the code is, the more vulnerability and risk exist for that module.
- Object-oriented metrics: These metrics are valid for an object-oriented programming paradigm, and they are related to specific programming concepts, such as inheritance, class, cohesion, and coupling.

- Change or process metrics: These metrics are related to the changes made during the software development process.
- Developer metrics: These metrics are directly related to the software developer and are extracted from the different developers who contribute to the software.
- Network metrics: These metrics are the most recent ones used for fault prediction. Network metrics are extracted from the dependency relation between different entities.

Software metrics threshold calculation techniques to predict fault-proneness:

Problems generated by faults could cause major damage and important losses of money. Having high-quality software can prevent faults, therefore reducing costs incurred for their correction.

The problem is that it is cost-prohibitive, difficult, and often impossible to exhaustively test all execution paths of complex software to ensure that it is fault-free. To support developers and testers in the testing process, quality models and tools can be used for identifying poor quality and particularly fault-prone code. These models generally use source code metrics to identify fault-prone classes or methods.

Models based on the threshold effect of code metrics consider classes as fault-prone when the values of their metrics exceed certain thresholds. The advantage of these specific models is that they can easily be implemented and understood by software engineering experts and developers.

In addition, they can provide valuable and simple insights into why a specific class is classified as fault-prone by indicating which metrics have problematic values and need to be adjusted.

There are two techniques for threshold calculation as fault-proneness predictors.

- ROC (Receiver Operating Characteristic) Curves method.
- VARL (Value of an Acceptable Risk Level)

ROC Curves method produces threshold values for other datasets and achieves good binary fault-proneness prediction performance. For methods like ROC Curves or VARL, we need fault data history to calculate threshold values. We also need to perform this cross-project and cross-version prediction with supervised approaches, to investigate which methodology should be used in a real-life context.

Choosing software metrics for defect prediction:

The selection of software metrics for building software quality prediction models is a search-based problem. Defect prediction models are necessary for aiding project managers in better utilizing valuable project resources for software quality improvement. The efficacy and usefulness of a fault-proneness prediction model are only as good as the quality of the software measurement data.

Source code metrics are essential components in the software measurement process. They are extracted from the source code of the software, and their values allow us to reach conclusions about the quality attributes measured by the metrics.

The effect of dataset size, metrics sets on software fault prediction problem:

Fault-prone modules can be automatically identified before the testing phase by using the software fault prediction models. Generally, these models mostly use software metrics of earlier software releases and previous fault data collected during the testing phase. The benefits of these prediction models for software development practices are shown as follows:

•Refactoring candidates can be identified from the fault prediction models, which require refactoring.

•Software testing process and software quality can be improved by allocating more resources to fault-prone modules.

•The best design approach can be selected from design alternatives after identifying the fault-prone classes by using class-level metrics.

•As fault prediction is an approach to reaching dependable systems, we can have a highly assured system by using fault prediction models during development.

Metrics Selection:

To achieve effective metrics, we leverage the insights used in the literature in defining coarse-grained metrics. Then, we adapt them to obtain fine-grained, relevant metrics. The metric suite we provided is intended to have the following properties:

- They capture different static features of a statement
- The metrics are expressive enough to distinguish as many similar statements as possible.
- They reflect some complex aspects of a statement that relates to fault-proneness.

CHAPTER-3

METHODOLOGY

3.1 DATASET

Our dataset consists of C/C++ programs within the Code4Bench. The program codes also have faulty lines. The benchmark comprises a diverse set of programs and versions, written by thousands of developers. Therefore, it tends to give a model that can be used for cross-project SDP. In the experiments, our trained model could successfully classify the unseen data with average performance measures in terms of recall, precision, and accuracy respectively.

3.2 Operations on Dataset

The preprocessing methods we followed are:

1. Metrics Extraction

The metric suite is intended to extract static information out of a statement that can be used to effectively distinguish it from another statement. We need to preserve every extracted metric per each statement. The metrics are also intended to estimate how much a statement is likely to be faulty.

We use a neural network to extract a set of metrics (features) for code change measurement. Then, we trained the data from extracted metrics to predict the defect-prone code changes.

2. Tokenization

Tokenization is essentially splitting a phrase, sentence, paragraph, or an entire text document into smaller units like individual words or terms. Each of these smaller units is called a token shown as words, numbers, or punctuation marks.

For each statement of each C/C++ program, we measured our introduced metrics. Then, we tokenized each statement to its lexemes. For fault data, we used the corresponding ready-made information, available in Code4Bench. Next, we made a matrix per program, in which

each row is devoted to a statement. For each statement, the corresponding columns are used for metric data, including lexemes, the fault data, and a label indicating fault-prone or fault-free. Finally, we trained an LSTM model using the matrix of all programs.

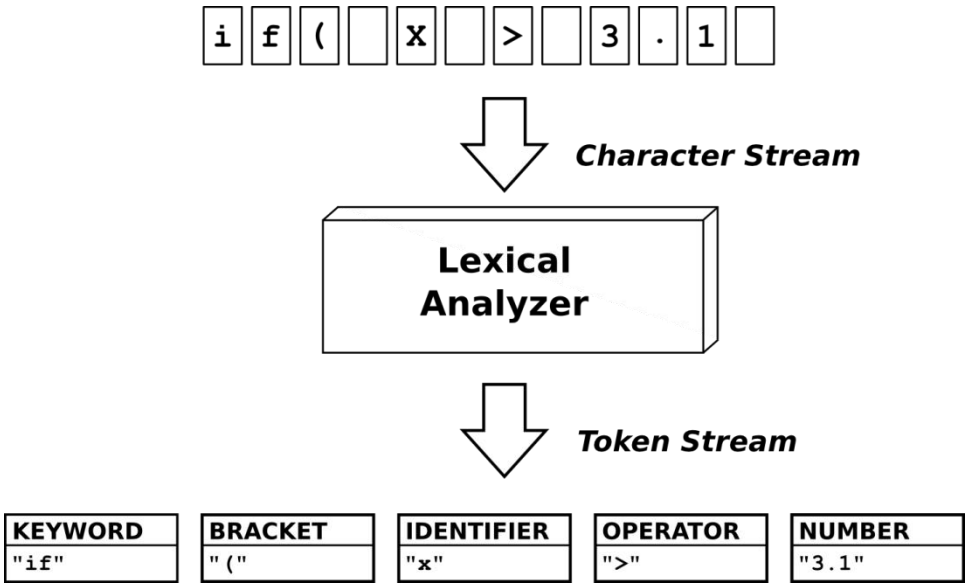


Fig 3.1

3. Batch Normalization

We use Batch Normalization in this project. It enables faster and stable training of deep NN by stabilizing the distributions of layer inputs during the training phase. To improve the training in a model it's important to reduce the internal co-variant shift. This works here to reduce the internal covariate shift by adding network layers that control the means and variances of the layer inputs.

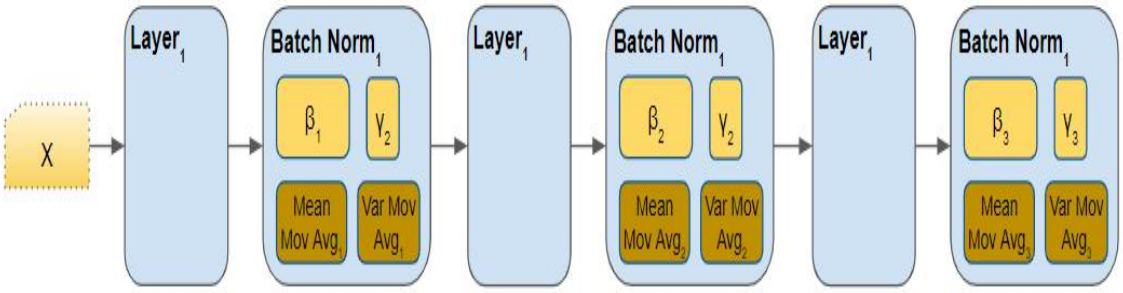


Fig3.2

Data Pre-Processing:

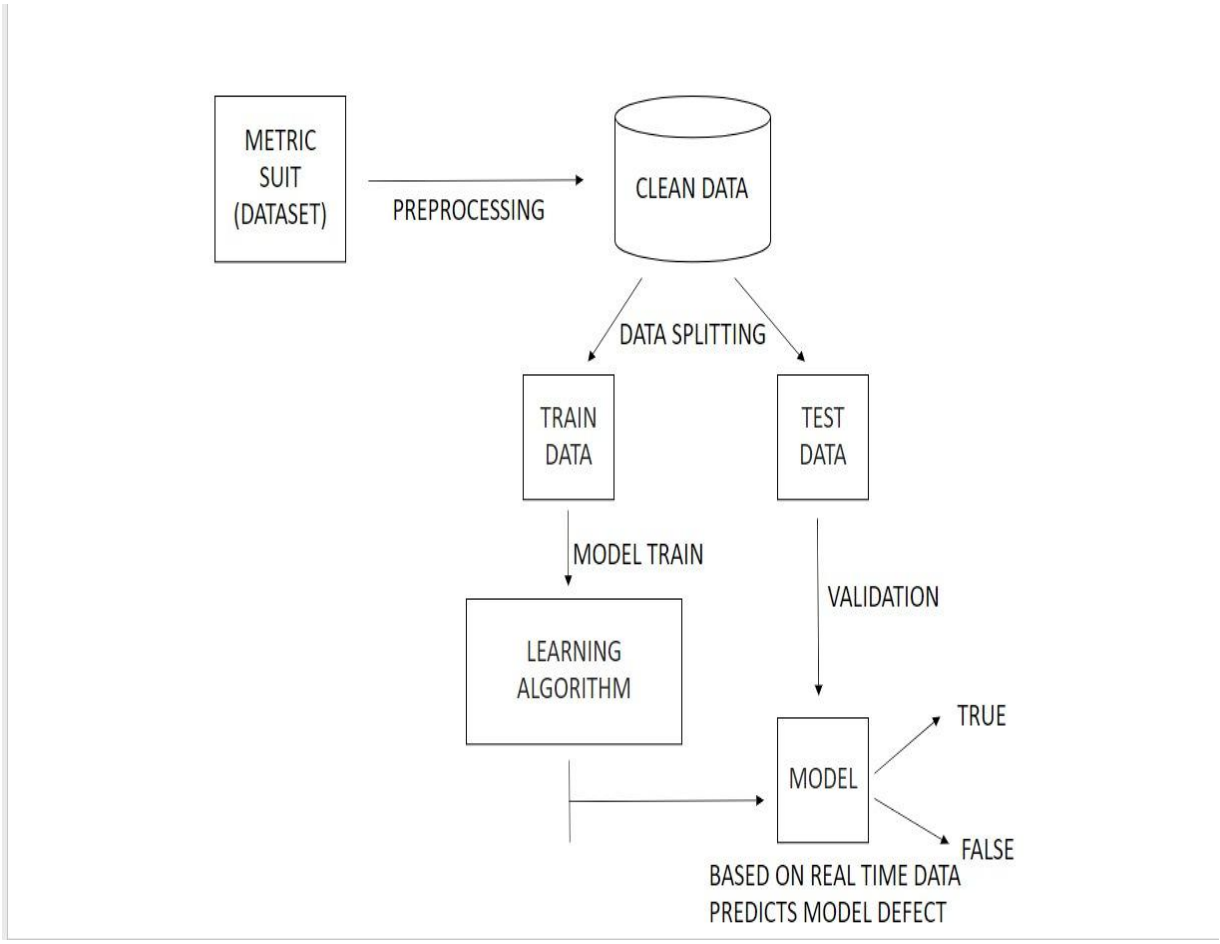


Fig3.3

3.3 Metric Suite

Metrics are independent variables. External-linear metrics capture external characteristics that influence the complexity of a statement. Internal-linear metrics estimate the complexity of a statement based on the existing properties of the statement.

The metrics are categorized into various groups:

1. Object-oriented: Object-oriented metrics are units of measurement that are used to characterize: object-oriented products, e.g., design source code, and test cases.

2. Function-level: Function-Oriented Metrics are also known as Function Point models. This model generally focuses on the functionality of the software application being delivered.

3. Statement level: We divide the source code into several statements. In this project, we use the Statement Level metric suite.

ID	Metrics	Description
1	Function	Is the line located in a function
2	Recursive Function	Is the line located in a recursive function?
3	Blocks Count	The number of nested blocks in which the line is located.
4	Recursive Blocks Count	The number of nested recursive blocks in which the line is located.
5	FOR Block	The number of nested FOR blocks in which the line is located.
6	DO Block	The number of nested DO WHILE blocks in which the line is located.
7	While Block	The number of nested WHILE blocks in which the line is located.
8	IF Block	The number of nested IF blocks in which the line is located.
9	SWITCH Block	The number of nested SWITCH blocks in which the line is located.
10	Conditional Count	The number of single conditions checked to reach a line. This includes the number of components in a compound condition as well as nested conditionals.

Table 3.1

3.4 Learning Model: LSTM

It is an effective machine learning technique. It is used to model the relationships between a sequence. It belongs to the Recurrent neural networks family. It uses self-loops to establish paths. LSTM is an effective deep-learning model for widespread usage.

Most of the current SDP research relies on machine-learning techniques to construct a learning model. The reason is that machine-learning models are tolerant of imprecise and partially incorrect data. For our SLDeep, we use an effective machine-learning technique, namely long short-term memory (LSTM), which belongs to the deep-learning sub-category.

LSTM was invented in 1997 to address some of the mathematical issues introduced when modeling long sequences. Today, it is also used to model the relationships between a sequence and other sequences. LSTM belongs to the recurrent neural networks (RNN) family. A feed-forward neural network, which is extended to support feedback connections, is called RNN.

An LSTM uses self-loops to establish paths in which the gradient can flow for long durations. In addition, the weights of self-loops are not fixed; they are determined depending upon the context. Self-loops serve as internal recurrence within units called LSTM cells. These cells have the same input/outputs as ordinary RNNs. However, they have more parameters and several gate units that control the information flows

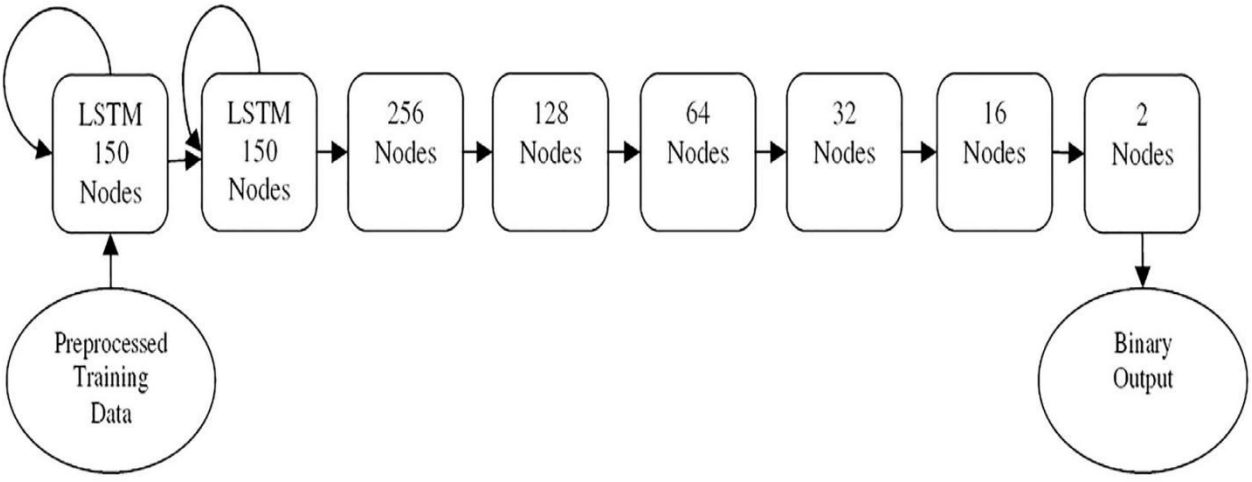


Fig 3.4

We need a model with memorization to consider the code structure as a sequence of statements, not each statement in isolation. The reason lies in the fact that we seek to apply a model that can learn based on what has been observed so far and make more informed decisions. As a result of these requirements, we found that LSTM could be a relevant model.

The structure of the matrix constructed for each program to be given to the LSTM model:

Rows /Columns	Metrics1	Metrics2	….	Metric32	Token1	Token2	….	Token max	Class
Line1	M1,1	M1,2	….	M1,32	(t1,v1)1	(t2,v2)1	….	(t max,vmax)1	Fault-free/fault-prone
….	….	….	….	….	….	….	….	….	….
Linen	Mn,1	Mn,2	….	Mn,32	(t1,v1)n	(t2,v2)n		(t max,vmax)n	Fault-free/fault-prone

Table3.2

3.5 SLDeep

The significance of SLDeep for intelligent and expert systems is that it demonstrates a novel use of deep-learning models to the solution of a practical problem faced by software developers.

To use SLDeep, one needs to compute metric data, extract tokens and construct a matrix for a given program. Then an LSTM model is trained and used to predict whether a new unseen statement is either fault-free or fault-prone. A major component of SLDeep is the learning model, which is constructed using LSTM.

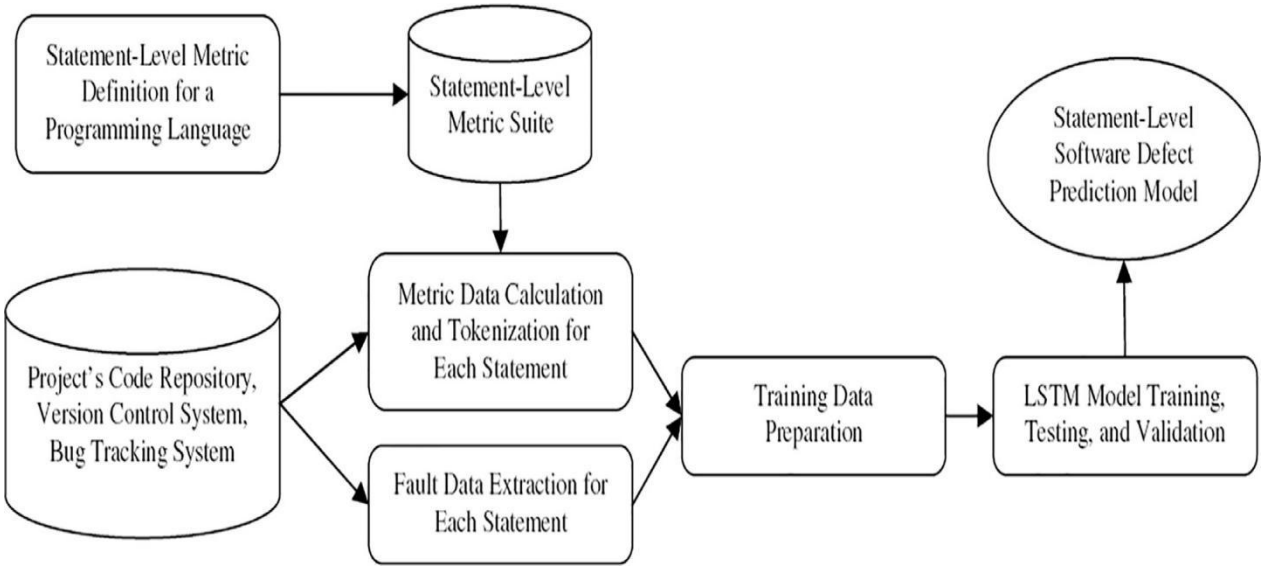


Fig 3.5

3.6 Performance Metrics

To evaluate the effectiveness of SL Deep, we computed four performance measures. The measures are Accuracy, Precision, Recall, and F-Measure, which are computed as following

1. Accuracy

It is the most common performance metric for classification algorithms. It may be defined as the number of correct predictions made as a ratio of all predictions made. We can easily calculate it by confusion matrix with the help of the following formula −

Accuracy= TP + TN / TP + TN + FP + FN

2. Precision

Precision, used in document retrievals, may be defined as the number of correct documents returned by our ML model. We can easily calculate it by confusion matrix with the help of the following formula −

Precision = TP/ TP + FP

3. Recall

Recall may be defined as the number of positives returned by our ML model. We can easily calculate it by confusion matrix with the help of the following formula −

Recall = TP/ TP + FN

4. F-measure

This measure will give us the harmonic mean of precision and recall. Mathematically, F-measure is the weighted average of precision and recall. The best value of the F-measure would be 1 and the worst would be 0. We can calculate the F-measure score with the help of the following formula −

F measure = 2 × precision × recall/ precision + recall

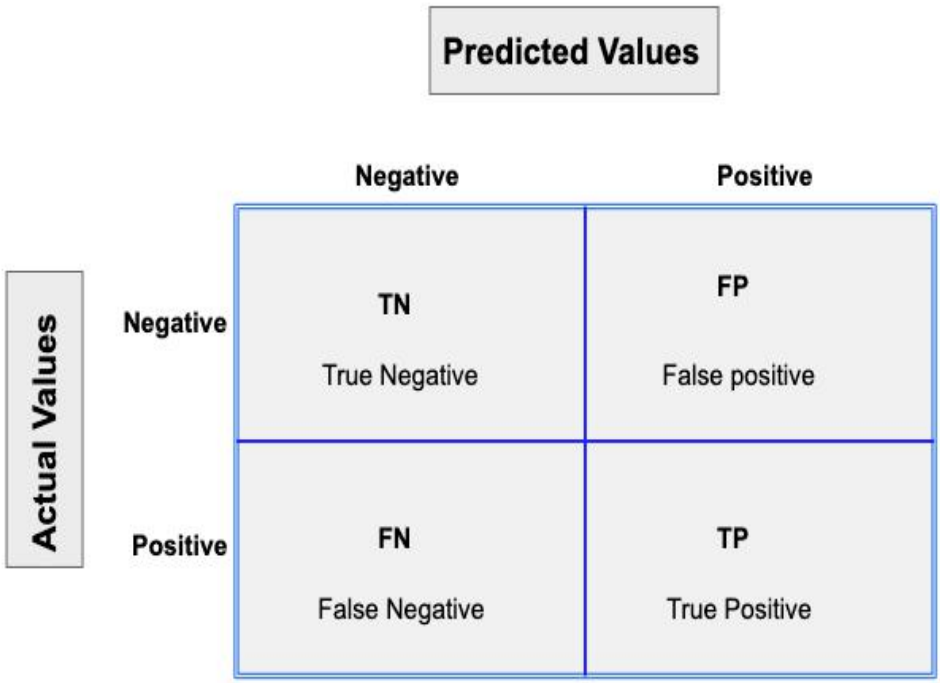


Fig 3.6

CHAPTER-4

EXPERIMENTAL DESIGN

4.1 Neural Networks

Neural networks are typically organized in layers. Layers are made up of several interconnected 'nodes' which contain an 'activation function'. Patterns are presented to the network via the 'input layer, which communicates to one or more 'hidden layers' where the actual processing is done via a system of weighted ' connections '. The hidden layers then link to an 'output layer' Neural Networks are typically organized in layers. Layers are made up of several interconnected 'nodes' which contain an 'activation function'. Patterns are presented to the network via the 'input layer, which communicates to one or more 'hidden layers' where the actual processing is done via a system of weighted ' connections '. The hidden layers then link to an output layer.

Deep Neural Networks

Deep Neural networks are computational algorithms. A Deep Neural Network (DNN) is the piece of a computing system designed to simulate the way the human brain analyzes and processes information. Deep Neural network is typically organized in layers. Layers are made up of many interconnected 'nodes' which contain an 'activation function'. A neural network may contain the following 3 layers:

- Input layer: This layer accepts input features. It provides information from the outside world to the network, no computation is performed at this layer, nodes here just pass on the information to the hidden layer.
- Hidden layer: Nodes of this layer are not exposed to the outer world, they are part of the abstraction provided by any neural network. The hidden layer performs all sort of computation on the features entered through the input layer and transfer the result to the output layer.
- Output Layer: This layer bring up the information learned by the network to the outer world.

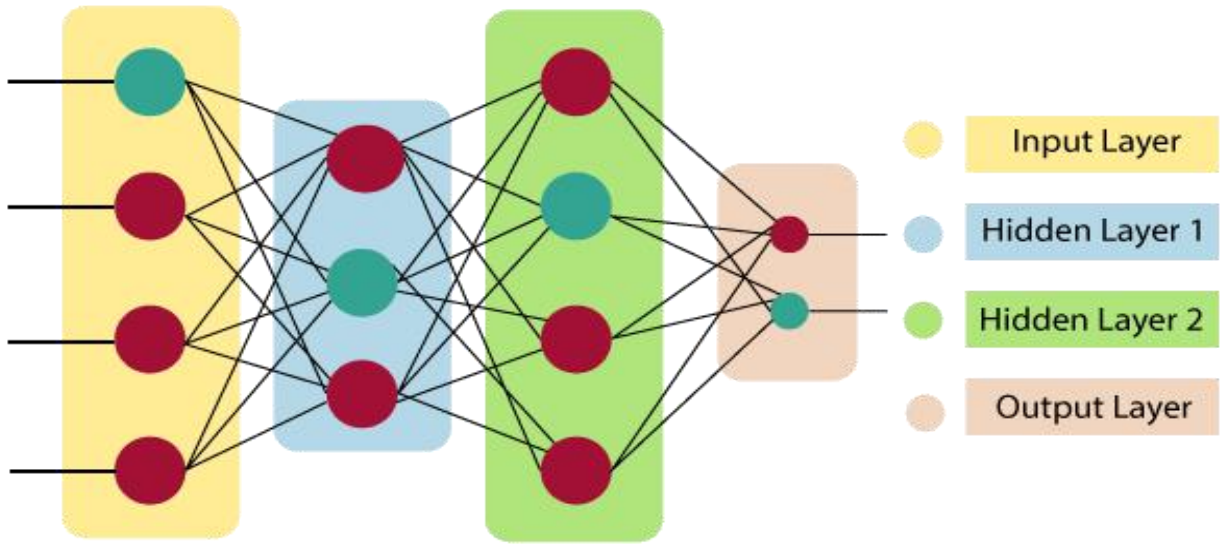


Fig 4.1

Activation Function:

- The activation function is also known as the Transfer function. The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The purpose of the activation function is to introduce non-linearity into the output of a neuron. A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks. The activation function also helps to normalize the output of any input in the range between 1 to -1. The activation function must be efficient and it should reduce the computation time because the neural network is sometimes trained on millions of data points.

We know, a neural network has neurons that work in correspondence with weight, bias, and their respective activation function. In a neural network, we would update the weights and biases of the neurons based on the error at the output. This process is known as backpropagation. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.

The main activation functions are:

- Tanh
- ReLU

4.1.1 ReLU Activation Function:

ReLU stands for Rectified Linear Unit. It is the most widely used activation function. Chiefly implemented in hidden layers of Neural network. It is non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function. It ranges from (0 to infinity).

The ReLU function is calculated as follows:

max(0.0, x)

This means that if the input value (x) is negative, then a value of 0.0 is returned, otherwise, the value is returned.

4.2 RANDOM FOREST

Random Forest is also a “Tree”-based algorithm that uses the qualities features of multiple Decision Trees for making decisions. Additionally, the Random Forest algorithm is also very fast and more robust than other models. Random Forest is a classifier that contains several decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset." Instead of relying on one decision tree, the random forest takes the prediction from each tree, and based on the majority votes of predictions, it predicts the final output. The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting. The fundamental concept behind random forest is a large number of relatively uncorrelated models(trees) operating as a committee will outperform any of the individual constituent models. The reason for this wonderful effect is that the trees protect each other from their errors (as long as they don't constantly all err in the same direction). While some trees may be wrong, many other trees will be right, so as a group the trees can move in the correct direction.

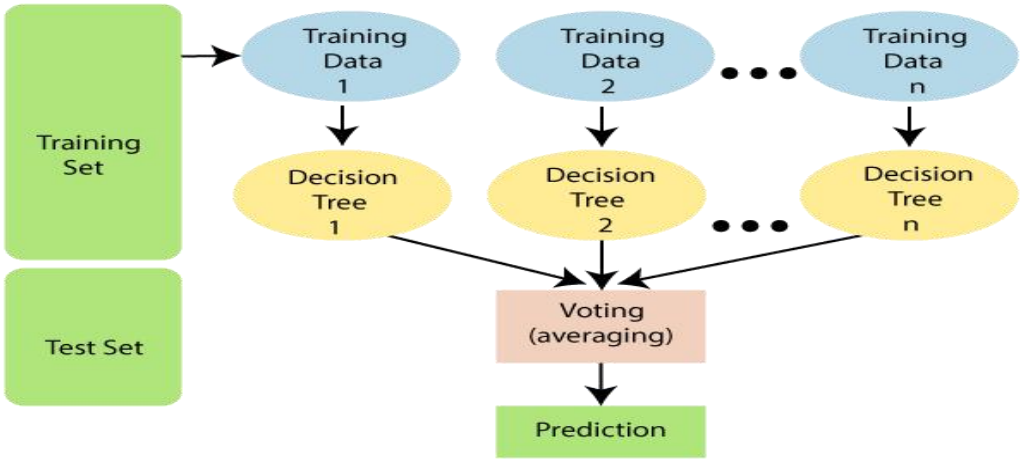


Fig 4.2

4.3 Cross-validation Strategy:

Cross-validation strategy we have used is:

- K-Fold(k=10):

A 10-fold cross-validation strategy is applied to compute the parameters. Each dataset is randomly partitioned into 10 subsets, each of which is equal to the others in terms of its size and one of which is considered as test data every time while the other nine subsets are considered as training data. This action should be reiterated ten times for running the same algorithm on data.

The algorithm of the k-Fold technique:

1.Pick a number of folds – k. Usually, k is 5 or 10 but you can choose any number which is less than the dataset's length.

2.Split the dataset into k equal (if possible) parts (they are called folds)

3.Choose k – 1 folds as the training set. The remaining fold will be the test set

4.Train the model on the training set. On each iteration of cross-validation, you must train a new model independently of the model trained on the previous iteration

5.Validate on the test set

6.Save the result of the validation

7.Repeat steps 3 – 6 k times. Each time use the remaining fold as the test set. In the end, you should have validated the model on every fold that you have.

8.To get the final score average the results that you got on step 6.

4.4 Hyper parameter Tuning

We need to set parameters for LSTM and Random Forest Models.

What is a Hyper Parameter?

A Hyper Parameter is a parameter whose value is used to control the learning process.
By contrast, the values of other parameters (typically node weights) are derived via training.
Hyper Parameters can be classified as model hyper parameters, that cannot be inferred while fitting the machine to the training set because they refer to the model selection task, or algorithm hyper parameters, that in principle have no influence on the performance of the model but affect the speed and quality of the learning process.

What is Hyper Parameter Tuning?
Hyper Parameter Tuning is the process of tuning the parameters present as the tuples while we build machine learning models. These parameters are defined by us which can be manipulated according to programmer wish.
Hyper Parameter tuning aims to find such parameters where the performance of the model is highest or where the model performance is best and the error rate is least.

Steps to be followed for Hyper Parameter Tuning:
Step1: Select the type of model we want to use like LSTM or any other

model.

Step 2: Check for the parameters of the model.

Step 3: Select the methods for searching the hyper parameter.(Here, we are using GridSearchCV).

Step 4: Select the cross-validation approach.

Step 5: Evaluate the model using the given software metrics.

Step 6: For hyperparameter tuning, we have used GridSearchCV method.

Steps to be followed for GridSearchCV:

For Random Forest, we have used:

- max_depth : The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- min_samples_split :The minimum number of samples required to split an internal node.
- min_samples_leaf: The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.
- max_features: The number of features to consider when looking for the best split.

```
param_grid = {
'max_depth': [128],
'min_samples_leaf': [0.2],
'min_samples_split': [0.2],
‘n_estimators’:400
}
```

LSTM					RF		
Optimizer	Learning_rate	Loss_function	Epochs	Batch-size	max_depth	min_sample_ leaf	min_sample_s plit
Adam	1e-3	Categorical Cross-entropy	15	16	128	0.2	0.2

Total Parameters	646634
Trainable parameters	645642
Non-trainable	992

Table 4.1

CHAPTER-5

EXPERIMENTAL SETUP

5.1 Import Libraries

- Tensor Flow - It's an open-source library used for high-level computations. Its mostly used in Machine Learning and Deep Learning Algorithms.
- Keras - It's a high-level, deep learning API used for the implementation of Neural Networks.
- Pandas -Pandas use sentinels for missing data, and use two already-existing Python null values: the special floating-point NaN value, and the Python None object.

- Numpy- It's a machine learning library that supports large matrices and multi-dimensional data. It consists of in-built mathematical functions for easy computations.

- StandardScaler- We import standardScalar from sklearn. preprocessing to remove the mean and scales each variable to unit variance.

5.2 Configure the Global Variables

Global Variables: Variables created outside of a function. Can be used both inside and outside of a function.

Define literals: They represent a fixed value in the source code. They can also be defined as raw values or data given in variables and constants. In this project, we use String, numeric, Integer, Float, and constant literals in the cpp.py file.

Tokens: It's the smallest unit in python program. All statements and instructions in a program are built with tokens.

Lex Scanner: It's a popular scanner (lexical analyzer) generator.

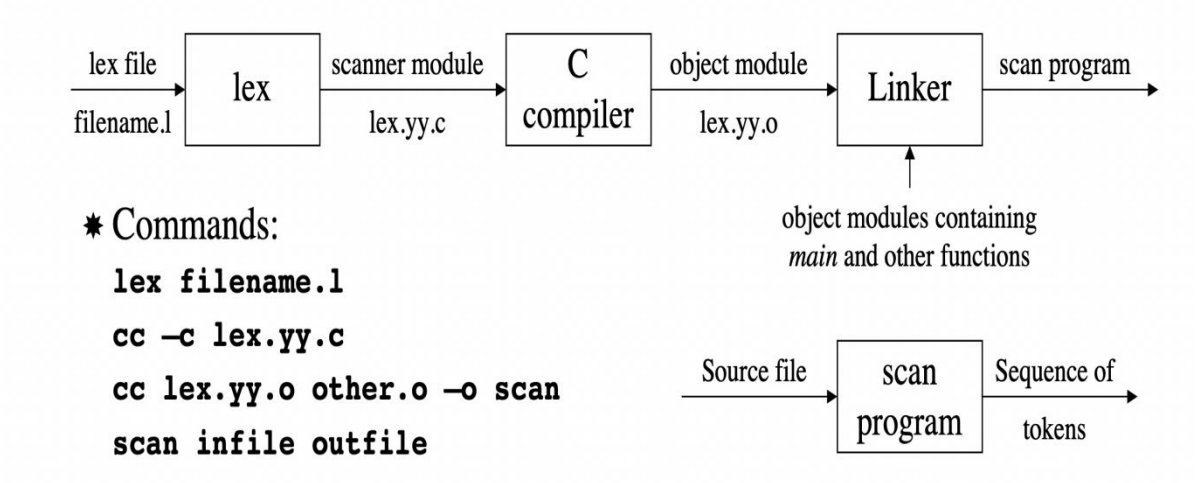


Fig 5.1

5.3 Making A Custom Loss Function

Machines learn using a loss function.

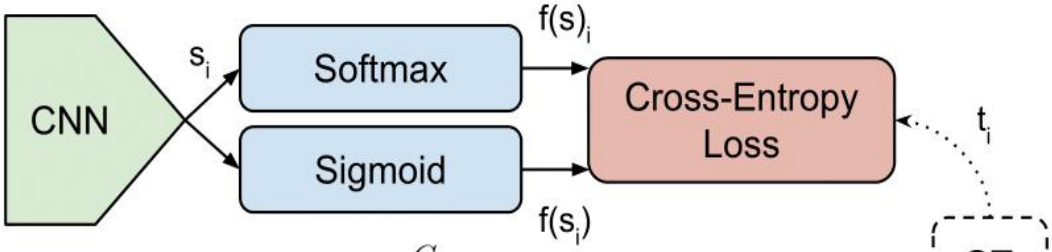
- Loss Function: It's a method of evaluating how well your algorithm models your dataset. With the help of some optimization function, the loss function learns to reduce the error in prediction.
- Neural Network uses optimizing strategies like stochastic gradient descent to minimize the error in the algorithm. The way we compute this error is by using a Loss Function.
- Gradient Descent: A gradient is the slope of a function. The greater the gradient, the steeper the slope. Starting from an initial value, Gradient Descent runs iteratively to find the optimal values of the parameters to find the minimum possible value of the given cost function.
- Stochastic Gradient Descent (SGD): It's a type of Gradient Descent that uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.

$$\text{for } i \text{ in range } (m):$$
$$\theta_j = \theta_j - \alpha \left(\widehat{y}^i - y^i\right) x_j^i$$

5.3.1 Categorical Cross Entropy Loss

Multi-Class Classification Loss Function: It is a type of Loss function where the target variable has more than two classes Multi-Class Classification Loss function is used.

1. Categorical Cross-Entropy Loss: Cross-entropy is the default loss function to use for binary classification problems. These are similar to binary classification cross-entropy, used for multi-class classification problems.



$$CE = -\sum_{i}^{C} t_i log(f(s)_i)$$

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$

Fig 5.2

Output Activation Functions:

SoftMax: SoftMax is a function, not a loss. It is applied to the output scores s.

Sigmoid: It squashes a vector in the range (0, 1). It is applied independently to each element of s si. It's also called the logistic function.

Losses:

1. Cross-Entropy loss- usually an activation function (Sigmoid / SoftMax) is applied to the scores before the CE Loss computation. Logistic Loss and Multinomial Logistic Loss are other names for Cross-Entropy loss.

Categorical Cross-Entropy loss: It is a SoftMax activation plus a Cross-Entropy loss. We will train CNN to output a probability over the C classes for each image.

5.4 Lexical Scanner and Tokenize Data Frames

- Lex is a popular scanner (lexical analyzer) generator.
- Input to Lex is called Lex specification or Lex program.
- Lex generates a scanner module in C from a Lex specification file.
- The scanner module can be compiled and linked with other C/C++ modules.
- A-Lex specification file consists of three sections: definition section, rules section, and auxiliary functions.
- The definition section contains a literal block and regular definitions. The rules section contains regular expressions and C code.
- An auxiliary function is a function that exists to help another function do its job.

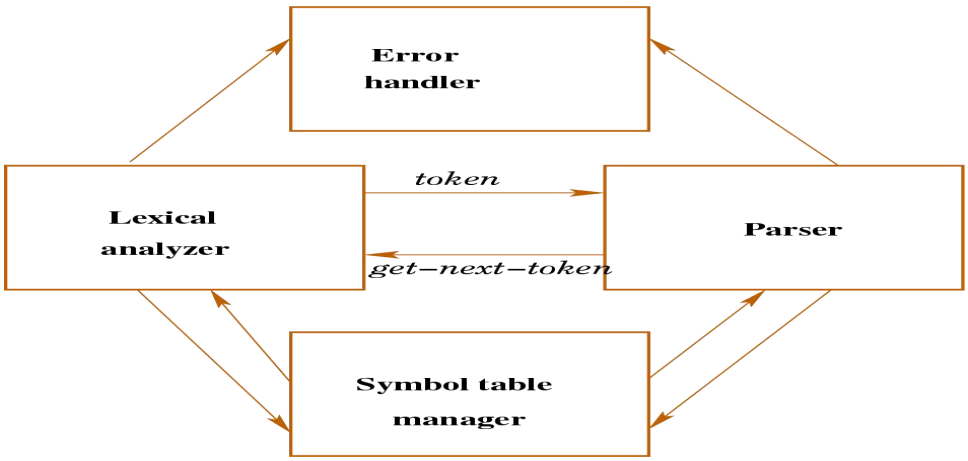


Fig 5.3

TOKENIZATION

Tokenization is essentially splitting a phrase, sentence, paragraph, or an entire text document into smaller units like individual words or terms. Each of these smaller units is called a token shown as words, numbers, or punctuation marks.

The meaning of the text could easily be interpreted by analyzing the words present in the text through the tokenization process.

5.5 Concatenating 32 Processed Columns

- We use a suite of 32 new code metrics that measure features of a statement i.e. no of binary and unary operators used in a statement. An eg of such metrics is the number of binary operators used in a line.
- This captures the complexity of a statement that influences fault-proneness.
- The number of columns can be calculated by using this formula:
- The number of columns = 32 + max (NTi,j); NT is the number of tokens, where *NT*: number of tokens for the j^{th} row of program P_i, $1 \leq i \leq P, 1 \leq j \leq r_i$, *P*: the number of programs; r_i: the number of rows of program P_i.

5.6 Train, Test, and Validation

Training

- Training data is the initial dataset used to train machine learning algorithms.
- Models create and refine their rules using this data.
- A set of data samples was used to fit the parameters of the ML model to train it.

Testing

- The test set is a set of observations used to evaluate the performance of the model using some performance metric. It is important that no observations from the training set are included in the test set.
- Balancing memorization and generalization, or over-fitting and under-fitting is a problem common to many machine learning algorithms. Regularization may be applied to many models to reduce over-fitting.

Validation

- The validation set is a set of data, separate from the training set, that is used to validate our model performance during training.
- The main idea of splitting the dataset into a validation set is to prevent our model from overfitting i.e., the model becomes good at classifying the samples in the training set.

Cross-Validation:

Cross-validation is a technique for evaluating a machine learning model and testing its performance. CV is commonly used in applied ML tasks. Process:

1. Divide the dataset into two parts: one for training, and another for testing.

2.Train the model on the training set.

3. Validate the model on the test set.

4. Repeat 1-3 steps a couple of times. This number depends on the CV method that you are using.

TOTAL =10,000 Files

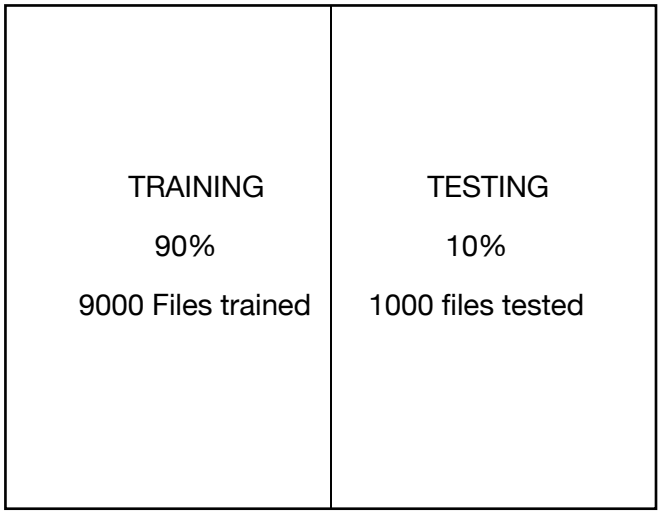


Fig 5.4

Chapter 6

SLDeep Architecture

6.1 SLDeep Architecture

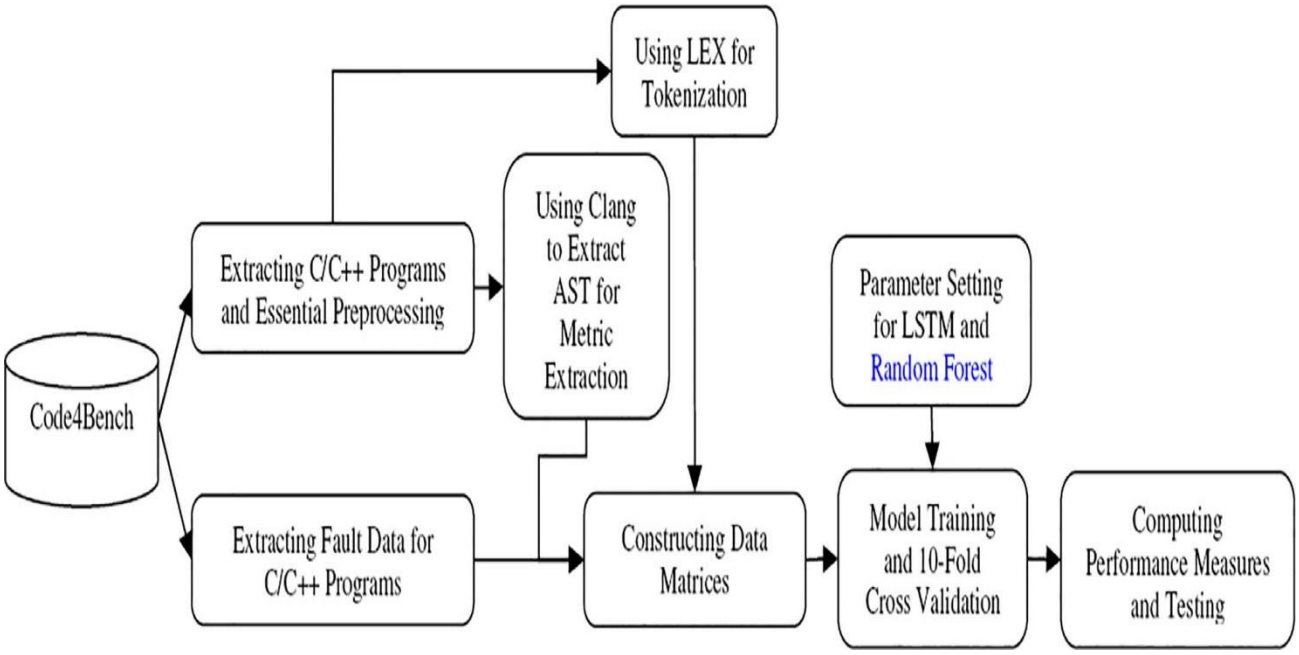


Fig 6.1

6.2 Steps Followed:

1. Choose a dataset that consists of C/C++ programs.
2. Extract C/C++ programs and perform data Preprocessing.
3. Perform metric extraction and generate tokens using LEX.
4. Train the data using two models:
   (i) LSTM
   (ii) RF
5. Calculate 10 - Fold Cross-Validation for both the models.
6. Compute the performance metrics and perform testing.
7. The final Statement level software defect prediction model is generated.

6.3 Model Description:

1. The input data taken by the LSTM model is (8999,18,93).
2. There are two LSTM layers with 150 nodes each.
3. 6 Dense layers are present - 256,128,64,32,16,2
4. 7 Droupout layers are present - 0.2,0.2,0.3,0.25,0.3,0.4,0.4
5. 5 Batch Normalizations are present between 6 Dense layers.
6. Activation Function:

   Rectified linear unit (RELU) for 5 Dense Layers.

   Softmax for last dense layer.

7. Random Forest :Parameter Tuning

   Best Parameters: max-depth =128

   min_samples = leaf=0.2,

   min_samples_split=0.2,

   n_estimators=400

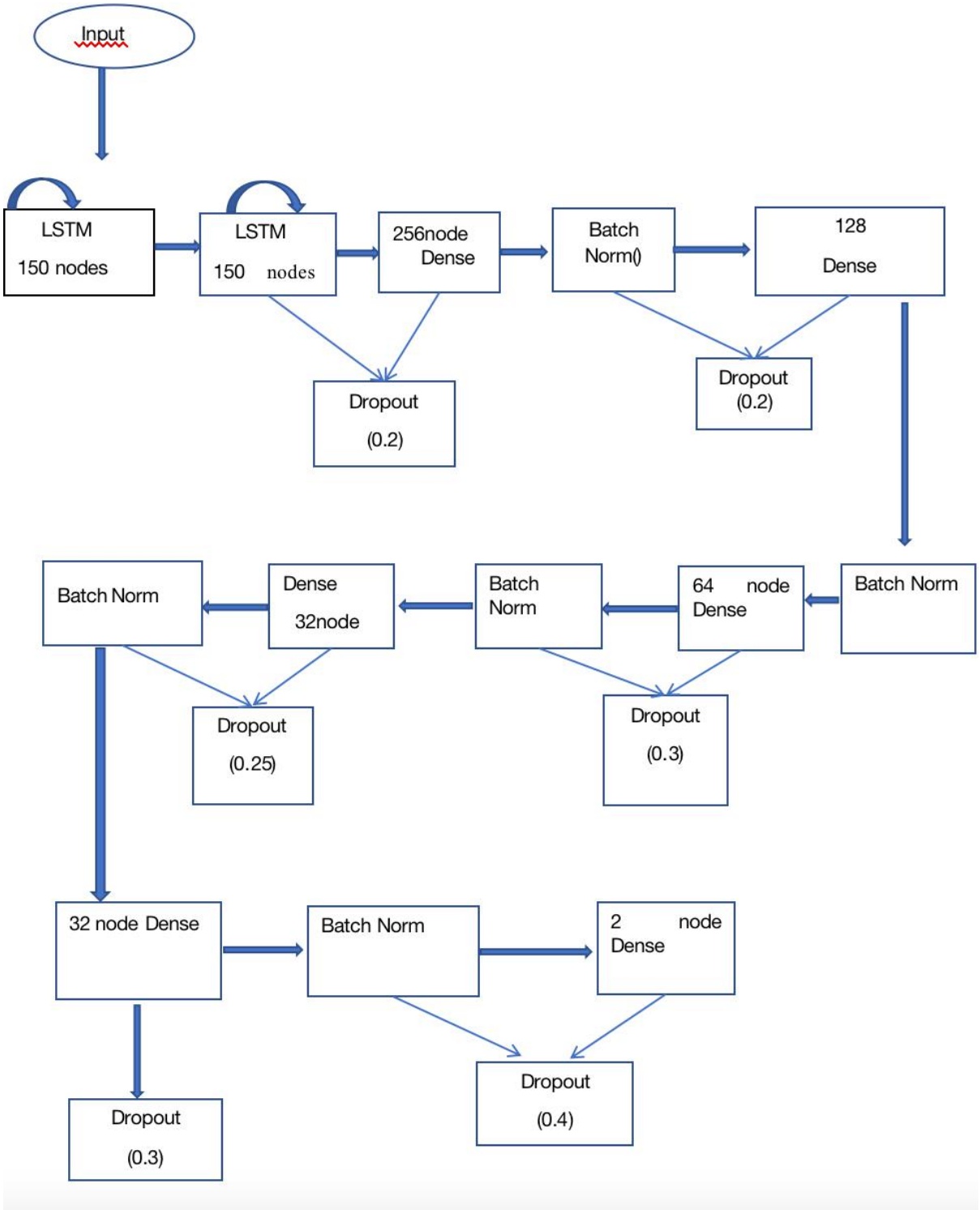

Input
LSTM
150 nodes
LSTM
150 nodes
256node
Dense
Batch
Norm()
128
Dense
Dropout
(0.2)
Dropout
(0.2)
Batch Norm
Dense
32node
Batch
Norm
64 node
Dense
Batch Norm
Dropout
(0.25)
Dropout
(0.3)
32 node Dense
Batch Norm
2 node
Dense
Dropout
(0.3)
Dropout
(0.4)

Fig 6.2

CHAPTER- 7

PSEUDOCODE

7. SOURCE CODE

#import libraries

```
import numpy as np
import pandas as pd
import keras
from keras.layers import *
from keras.models import *
from keras.activations import *
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
import os
import itertools
import string
import tensorflow as tf
from itertools import product
import math
from utils import lex
from utils import yacc
from utils import cpp
import zipfile
```

 # Here We Will Configure The Global Variables

```
zip_name = "./data.zip"
# The columns containing the code info
```

```
cols = ["code", "block"]

# The Archive Containing The Actual Codes

archive = zipfile.ZipFile('data.zip', 'r')

# The first name is the name of the containing folder of the codes

list_files = archive.namelist()[1:]

# Now We will make a scanner for the C++ language

scanner = lex.lex(cpp)

# If any thing was considered fault at the line i, we will consider all the lines [i - range_n, i +
range_n) to be fault

range_n = 4
```

 # Then We Define The literals of the program

```
lits = cpp.literals

# Then We Define The Tokens

toks = list(cpp.tokens)

# We remove the White Space token to add it later

toks.remove("CPP_WS")

# We add the White Space token here because we want it to have the value of zero, we'll use
this latter for padding lines of code

toks.insert(0, "CPP_WS")

# Tok 2 N : a dictionary from tokens to thier integer, mapped, value

tok2n = dict(zip(toks + [i for i in lits], itertools.count()))

# N 2 Tok : a dictionary from integers to thier token, mapped, value

n2tok = dict(zip(itertools.count(), toks + [i for i in lits]))

# The maximum value we allow in as a constant value in a code
```

```
max_v = 2147483647 - 1

# The amount of importance we give to 1s 0s and false postives and false negatives

WEIGHTS_FOR_LOSS = np.array([[2,0.5],[0.1,0.1]])

# Here We Will Make A Custom Loss Function
def get_loss_function(weights, rnn=True):

'''

gives us the loss function

'''

def w_categorical_crossentropy_mine(y_true, y_pred):

nb_cl = len(weights)

if(not rnn):

final_mask = K.zeros_like(y_pred[:, 0])

y_pred_max = K.max(y_pred, axis=1)

y_pred_max = K.reshape(y_pred_max, (K.shape(y_pred)[0], 1))

y_pred_max_mat = K.equal(y_pred, y_pred_max)

for c_p, c_t in product(range(nb_cl), range(nb_cl)):

final_mask += ( weights[c_t, c_p] * K.cast(y_pred_max_mat, tf.float32)[:, c_p] * K.cast(y_true, tf.float32)[:, c_t] )

return K.categorical_crossentropy(y_true, y_pred, True) * final_mask

else:

final_mask = K.zeros_like(y_pred[:, :,0])

y_pred_max = K.max(y_pred, axis=2)

y_pred_max = K.reshape(y_pred_max, (K.shape(y_pred)[0], K.shape(y_pred)[1], 1))

y_pred_max_mat = K.equal(y_pred, y_pred_max)
```

for c_p, c_t in product(range(nb_cl), range(nb_cl)):

final_mask += (weights[c_t, c_p] * K.cast(y_pred_max_mat, tf.float32)[:, :,c_p] * K.cast(y_true, tf.float32)[:, :,c_t])

return K.categorical_crossentropy(y_true, y_pred, True) * final_mask

return w_categorical_crossentropy_mine

 # Lexical Scanner Function

def get_replacement(scanner, string_in):

'''

gets a string and returns the None, 2 which is the tokenized version

'''

try :

scanner.input(string_in)

except Exception as e :

print("Exception in using the lex", e)

print(string_in)

token = scanner.token()

id2n and n2id are the same as n2tok tok2n but they are extended to contain the information of the symbol table of each code separately

id2n = dict(zip([i for i in lits], [tok2n[i] for i in lits]))

n2id = dict(zip([tok2n[i] for i in lits], [i for i in lits]))

n_id = len(lits) + 1

res = []

```
while token is not None :
t = token.type
# If we have recieved a token and it is not something we need to use ord for
if t in cpp.tokens :
# Reciving a white space
if token.type == cpp.tokens[cpp.tokens.index("CPP_WS")]:
#this is because this will make it easier for us to pad our data
v = 0
# If we receive a string (We don't use the value of strings)
elif token.type == cpp.tokens[cpp.tokens.index("CPP_STRING")]:
v = -1
# If we recive #
elif token.type == cpp.tokens[cpp.tokens.index("CPP_POUND")]:
v = -2
# If we recive ##
elif token.type == cpp.tokens[cpp.tokens.index("CPP_DPOUND")]:
v = -3
# If we recive char
elif token.type == cpp.tokens[cpp.tokens.index("CPP_CHAR")]:
v = -4
elif token.type in cpp.tokens[3:]:
print("something went wrong")
# Here We Will Tokenize Our Data Frames
def tokenize_data(data):
```

'''

reads data and tokenizes each of the sentences and adds them together.

The output will contain the actual data, max number of lines per code, and mean number of lines per code

the actual data will have the following shape :

Number of codes, Number of lines per each code, 2 (Data and State)

The state will contain (Code being right, Code Being Wrong)

Data Will Contain (Number Of Words, 2 (Token, Value))

'''

res = []

x = []

mean = 0

max_num = 0

for i in data:

If We had any code submissions that was empty, we skip them

if i.shape[0] == 0 :

continue

temp = []

mean += i.shape[0]

max_num = max(max_num, i.shape[0])

for j in i.values :

try :

tok = get_replacement(scanner, j[0]).astype(np.float32)

except Exception as e :

```
for j in i :continue

x.append(tok)

y = j[-2:]

temp.append([tok, y])

res.append(temp)

mean /= len(res)

return res, mean, max_num
```

 # Concatenating The 32 Processed Columns And Lexical Feature

```
def get_final_data(tokenized_final, data):

'''

adds the information from the parser to the things that were gained from the information of scanners

tokenized_final will be the output of "change_cols" and data will be the output of "get_data"

'''

# The first line reads data and drops the following columns : columns containing text of the parser or lex and the

# The last two columns which are the state of the code which we are trying to predict

dataR = np.concatenate([i.drop(cols, axis = 1).values[:, :-2] for i in data], axis = 0)

dataR = dataR.astype(np.float32)

cnt = 0

res = []

for i in tokenized_final :

temp = []
```

```
add = dataR[cnt, :]
temp.append(np.concatenate([add, j], axis = 0))
cnt += 1
res.append(np.array(temp))
res = np.array(res)
return res
```

 # First We Read Our Data From The Zip File

```
data = get_data(list_data, archive)
# The We tokenize our data
r, mean, max_num = tokenize_data(data)
# Then We Create Our Empty Vector
empty = np.array([tok2n["CPP_WS"], 0]).reshape(1, 2).astype(np.float32)
# The Defualt Option for Padding
if pad1 is None :
pad1 = int(mean) + cons_per_line
#Making The Neural Network Model

def get_model(shape):
'''gets the first rnn model
shape : shape of the input : shape of the codes [Number of lines (which can be None),
Number of Fetures per line] '''
in1 = Input(shape)
X =Bidirectional(LSTM(150, return_sequences=True, dropout=0.25,
recurrent_dropout=0.1))(in1)
```

```
X = LSTM(150, return_sequences=True, dropout=0.25, recurrent_dropout=0.1,)(X)

X = Dropout(0.2)(X)

X = Dense(256, activation=relu)(X)

X = Dropout(0.2)(X)

X = BatchNormalization()(X)

X = Dropout(0.3)(X)

X = Dense(128, activation=relu)(X)

X = BatchNormalization()(X)

X = Dropout(0.25)(X)

X = Dense(64, activation=relu)(X)

X = BatchNormalization()(X)

X = Dropout(0.3)(X)

X = Dense(32, activation=relu)(X)

X = BatchNormalization()(X)

X = Dropout(0.4)(X)

X = Dense(16, activation=relu)(X)

X = BatchNormalization()(X)

X = Dropout(0.4)(X)

X = Dense(2, activation=softmax)(X)

model = Model(in1, X)

return model
```

#Using All The Defined Functions

```
# K is for our K-fold

k = 10
```

```
# The name of the codes we want to use, you can slice this list to a smaller list for a fast test
l = list_files[:]
# The number of codes we use on each fold
size = math.ceil(len(l) / k)
# Verbose for our NN model
verbose = 0
# The results for trains and tests respectivly
trs = []
ts = []
for i in range(k):
print("k", i)
# start and end will be the indicies for what we'll use for test
start = i * size
end = min(len(l), (i + 1) * size)
data_train = l[:start] + l[end:]
data_test = l[start : end]
if len(data_test) <= 0 or len(data_train) <= 0 :
print("hey")
continue
# gathering data for train
r_train, scaler, pad1, pad2 = gather_data(data_train, archive, add_all = True)
# gathering data for test, please note that the same mean and standard deviation that was computed for the train will be used to
```

```
# normalize test data and also the information of pad1 and pad2 is computed from train so that no information will be

# leaked from train and also none of the aforementioned are dependent on the test data

r_test, _, _, _ = gather_data(data_test, archive, scaler = scaler, add_all = True, pad1 = pad1, pad2 = pad2)

print("data read")

# configuring model

model = get_model([None, r_train.shape[-1] - 2])

loss = get_loss_function(WEIGHTS_FOR_LOSS)

model.compile(keras.optimizers.adam(lr = 1e-3), keras.losses.categorical_crossentropy, metrics = ["accuracy"])

# making the X, y for train and test set

X_train = r_train[:, :, :-2]

y_train = r_train[:, :, -2:]

X_test = r_test[:, :, :-2]

y_test = r_test[:, :, -2:]

# training the model

print("training on the data started ")

model.fit(X_train, y_train, validation_data = [X_test, y_test], epochs = 20, batch_size = 8, verbose = verbose)

print("training on the data finished ")

# saving and printing the accuracy of training data

print("train : rec1, prec1, accuracy, f1, acc , tp , fn , fp , tn ")

trs.append(get_mus(y_train, X_train,model))

print(trs[-1])
```

```
# preparing to write the training accuracy to a file
strRes = "train : "
for counter in range(8):
strRes = strRes + '%.5f' % trs[-1][counter] + " , "
strRes += " \n "
# saving and printing the accuracy of testing data
print("test : rec1, prec1, accuracy, f1, acc, tp , fn , fp , tn ")
ts.append(get_mus(y_test, X_test,model))
print(ts[-1])
# preparing to write the testing accuracy to a file
strRes += " test : "
for counter in range(8):
strRes = strRes + '%.5f' % ts[-1][counter] + " , "
# writing the accuracies on to a file
f = open("test.txt", "a")
f.write(strRes + "\n")
f.close()
#Results
trs = np.array(trs)
ts = np.array(ts)
print("avg train : rec1, prec1, accuracy, f1, acc")
print(np.mean(trs[:, : 4], axis=0))
print("avg test : rec1, prec1, accuracy, f1, acc")
print(np.mean(ts[:, : 4], axis=0))
```

CHAPTER- 8

RESULTS AND DISCUSSIONS

8.1 K-fold LSTM Average Results

	Names	train_res	test_res
0	rec1	0.5728405077628820	0.5301813110768230
1	prec1	0.9146516509275390	0.868879354822765
2	accuracy	0.8811057545488590	0.8592559893226560
3	f1	0.6725450388553330	0.6215304499278860

Table 8.1

8.2 The experimental results of training SLDeep on subject programs when using the LSTM learning model.

Fold	Train/ Test	Re-call	Precision	Accuracy	F-Measure	TP	FN	FP	TN
1	Train	0.7824	0.84227	0.91328	0.81222	30380.00000	8358.00000	5689.00000	117555.00000
	Test	0.69430	0.81072	0.88056	0.74801	3191.00000	1405.00000	745.00000	12659.00000
2	Train	0.30027	0.97388	0.83561	0.45901	11297.00000	26326.00000	303.00000	124056.00000
	Test	0.29119	0.8577	0.77378	0.44958	1663.00000	4048.00000	24.00000	12265.00000
3	Train	0.6667	0.9660	0.9148	0.7889	25791.	12891.	906.	12239

		4	6	2	87	00000	00000	00000	4.00000
	Test	0.60619	0.87172	0.87517	0.71510	2820.00000	1832.00000	415.00000	12933.00000
4	Train	0.22653	0.98493	0.81368	0.36718	8756.00000	30047.00000	134.00000	123045.00000
	Test	0.18716	0.98376	0.79461	0.31448	848.00000	3683.00000	14.00000	13455.00000
5	Train	0.58270	0.92462	0.88802	0.71488	22740.00000	16285.00000	1854.00000	121103.00000
	Test	0.51984	0.89243	0.87006	0.65699	2240.00000	2069.00000	270.00000	13421.00000
6	Train	0.53939	0.91211	0.87628	0.67790	21089.00000	18009.00000	2032.00000	120852.00000
	Test	0.54202	0.92693	0.88217	0.68405	2296.00000	1940.00000	181.00000	13583.00000
7	Train	0.76230	0.87861	0.91704	0.81634	29864.00000	9312.00000	4126.00000	118680.00000
	Test	0.76816	0.85355	0.91600	0.80861	3194.00000	964.00000	548.00000	13294.00000
8	Train	0.30701	0.97314	0.83238	0.46676	11883.00000	26823.00000	328.00000	122948.00000
	Test	0.28630	0.96715	0.81400	0.44181	1325.00000	3303.00000	45.00000	13327.00000
9	Train	0.76987	0.86881	0.91569	0.81635	30352.00000	9073.00000	4583.00000	117974.00000
	Test	0.7035	0.7819	0.8930	0.7406	2750.0	1159.0	767.00	13324.

		0	2	0	4	0000	0000	000	00000
10	Train	0.79023	0.82208	0.90426	0.80584	32186.00000	8544.00000	6966.00000	114304.00000
	Test	0.70315	0.61484	0.89323	0.65604	1831.00000	773.00000	1147.00000	14231.00000

Table 8.2

8.3 Training Result -LSTM

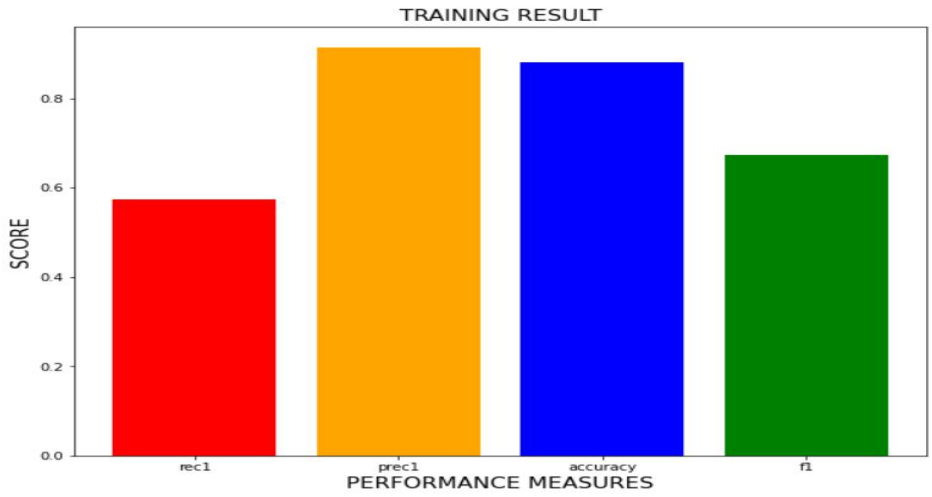


Fig 8.1

8.4 Testing Result- LSTM

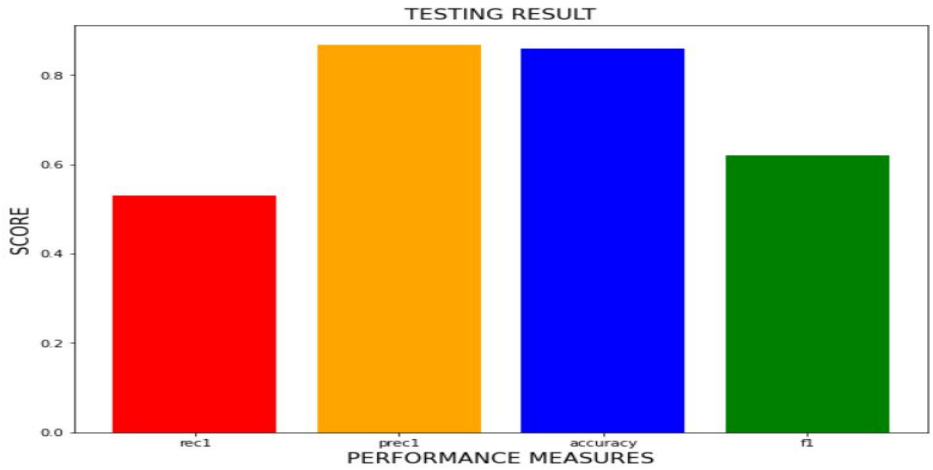


Fig 8.2

8.5 Train and Test Accuracy Graphs

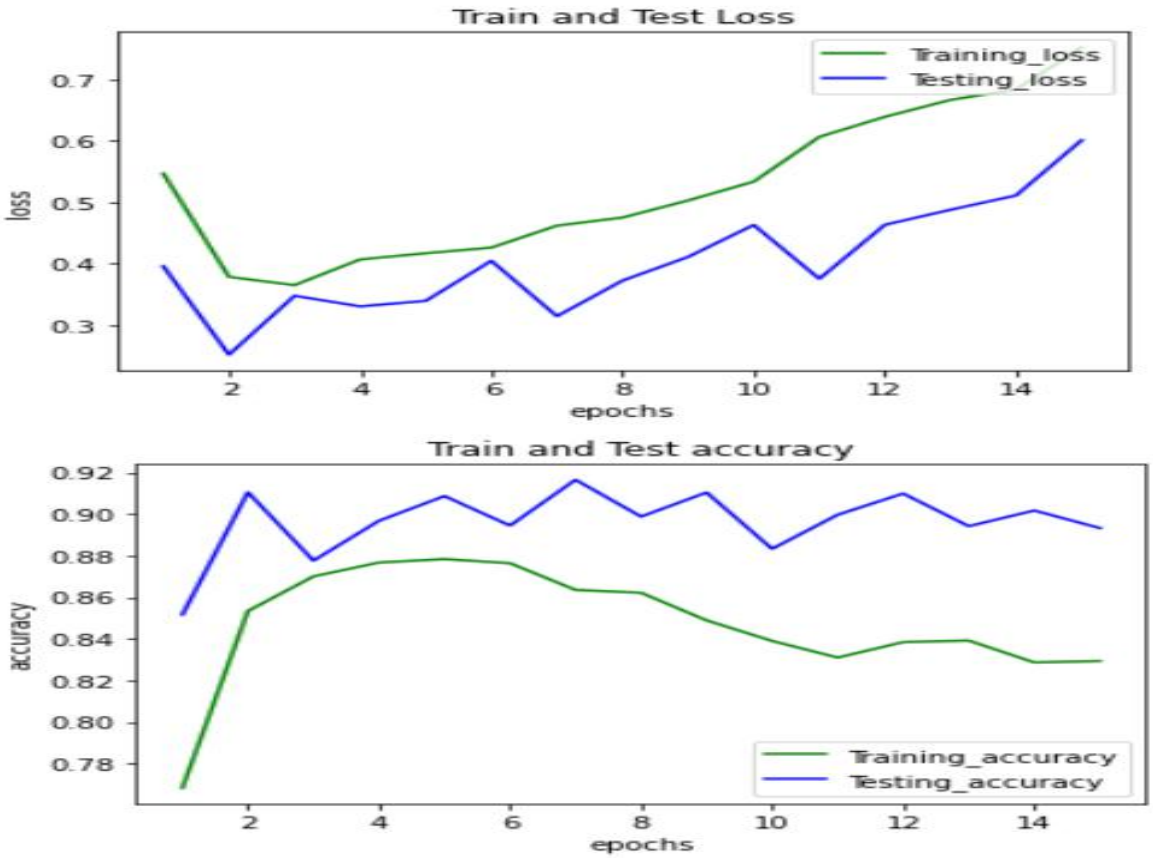


Fig 8.3

8.5 K-fold RF Average Results

	Names	train_res	test_res
0	rec1	0.204931303	0.205714868
1	prec1	0.583012795	0.583836476
2	accuracy	0.639141681	0.638606184
3	f1	0.302825421	0.302504018

Table 8.3

8.6 The experimental results of training SLDeep on subject programs when using the RF learning model.

Fold	Train/ Test	Re-call	Precision	Accuracy	F-Measure	TP	FN	FP	TN
1	Train	0.24494	0.58669	0.64231	0.34560	15299.000000	47161.00000	10778.00000	88744.00000
	Test	0.26136	0.52049	0.64567	0.34799	1702.00000	4810.00000	1568.00000	9920.00000
2	Train	0.199588	0.58607	0.64100	0.29362	12086.00000	49616.00000	8536.00000	91744.00000
	Test	0.19807	0.56206	0.61378	0.29292	1440.00000	5830.00000	1122.00000	9608.00000
3	Train	0.22902	0.57914	0.64163	0.32823	14182.00000	47744.00000	10306.00000	89750.00000
	Test	0.23134	0.60482	0.63994	0.33467	1630.00000	5416.00000	1065.00000	9889.00000
4	Train	0.19787	0.57826	0.63877	0.29485	12233.00000	49590.00000	8922.00000	91237.00000
	Test	0.19787	0.62522	0.63383	0.29711	1393.00000	5756.00000	835.00000	10016.00000
5	Train	0.19503	0.58234	0.63799	0.29220	12104.00000	49958.00000	8681.00000	91239.00000
	Test	0.18842	0.60446	0.64111	0.28729	1302.00000	5608.00000	852.00000	10238.00000
6	Train	0.19837	0.57793	0.63552	0.29536	12374.00000	50003.00000	9037.00000	90568.00000
	Test	0.21365	0.62456	0.66483	0.31838	1409.00000	5186.00000	847.00000	10558.00000

7	Train	0.19250	0.58969	0.63794	0.29025	11992.00000	50303.00000	8334.00000	91343.00000
	Test	0.21312	0.55804	0.64550	0.30844	1423.00000	5254.00000	1127.00000	10196.00000
8	Train	0.18634	0.58354	0.63701	0.28248	11574.00000	50538.00000	8260.00000	91610.00000
	Test	0.20787	0.58354	0.63701	0.28248	11574.00000	50538.00000	8260.00000	91610.00000
9	Train	0.20787	0.56835	0.63915	0.31462	13416.00000	48787.00000	9664.00000	90115.00000
	Test	0.20564	0.58340	0.64606	0.30410	1392.00000	5377.00000	994.00000	10237.00000
10	Train	0.19368	0.53400	0.64010	0.29103	11967.00000	49821.00000	8483.00000	91729.00000
	Test	0.14282	0.58696	0.61740	0.22974	1026.00000	6158.00000	722.00000	10076.00000

Table 8.4

8.7 Training Result -RF

Fig 8.4

8.8 Testing Result -RF

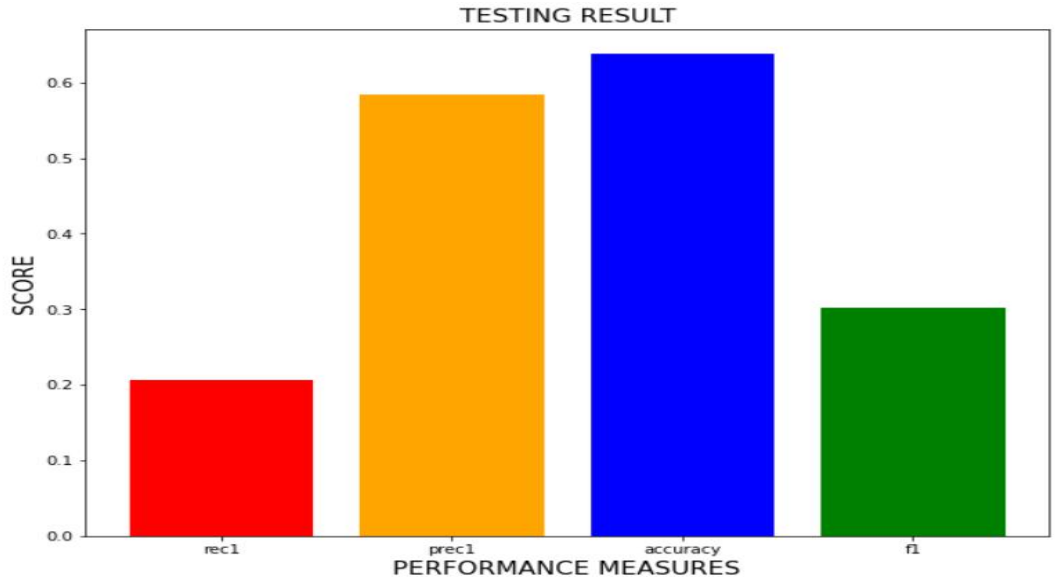


Fig 8.5

8.9 ROC Curve

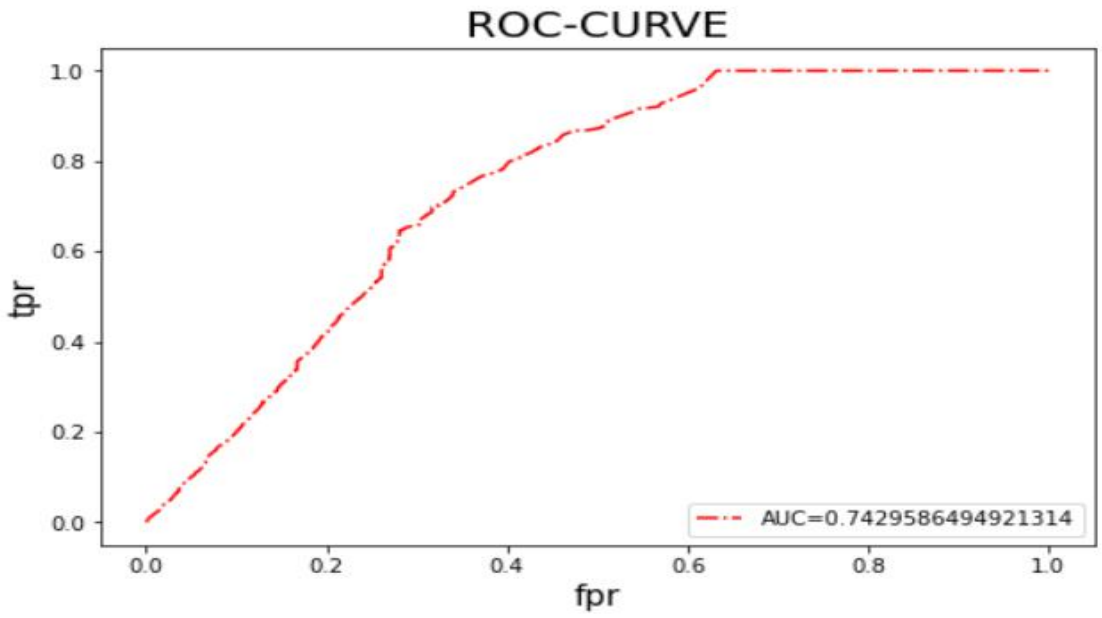


Fig 8.6

8.10 Precision-Recall Graph

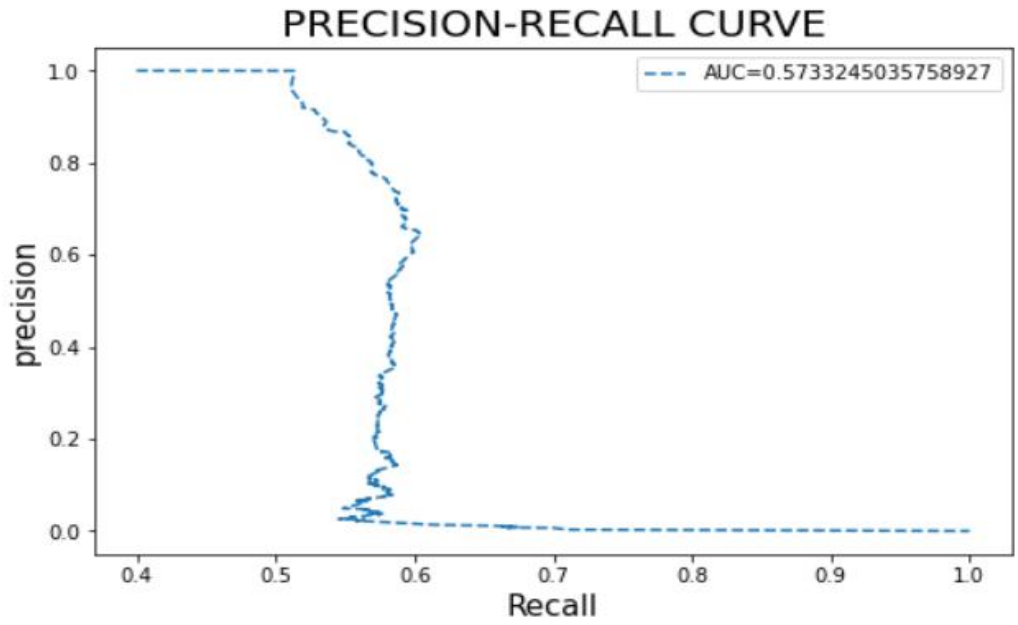


Fig 8.7

8.11 Results

```
[ ]  train_avg=np.mean(trs[:, : 4], axis=0)
     #print(np.mean(trs[:, : 4], axis=0))
     print(train_avg)
     print("avg test : rec1, prec1, accuracy, f1")
     test_avg=np.mean(ts[:, : 4], axis=0)
     #print(np.mean(ts[:, : 4], axis=0))
     print(test_avg)

     avg train : rec1, prec1, accuracy, f1
     [0.57284051 0.91465165 0.88110575 0.67254504]
     avg test : rec1, prec1, accuracy, f1
     [0.53018131 0.86887935 0.85925599 0.62153045]

[ ]

[ ]  train_res=train_avg
     test_res=test_avg

     names=['rec1', 'prec1', 'accuracy', 'f1']
     dict_pd={'names':['rec1', 'prec1', 'accuracy', 'f1'],'train_res':train_res,'test_res':test_res}
     data=pd.DataFrame(dict_pd)
     data
```

	names	train_res	test_res
0	rec1	0.572841	0.530181
1	prec1	0.914652	0.868879
2	accuracy	0.881106	0.859256
3	f1	0.672545	0.621530

Fig 8.8

CHAPTER- 9

CONCLUSION AND FUTURE WORK

9. CONCLUSION AND FUTURE WORK

- The goal of this project is to alleviate the developer's burden in pinpointing the locations of faults, hence providing high-quality software with less time and effort.
- To achieve this goal, we proposed SLDeep, a technique for statement-level software defect prediction.
- This work can be extended in several ways. First, we can explore to define additional metrics for C-based languages.
- Second, we can adapt the current metrics for other common programming languages such as Python and Kotlin.
- Third, new variants of LSTM can be used to exploit their potential.
- Fourth, Empirical studies can be conducted to investigate the effects of layers and nodes of LSTM on the overall performance of the model.
- Fifth, some experiments can be carried out on programs in other application domains, for example, Android applications.

CHAPTER-10

BIBLIOGRAPHY

BIBLIOGRAPHY

[1]Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers, principles, techniques. second edition*. Pearson Education Inc.

[2] Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., et al. (2009). Fair and balanced?: Bias in bug-fix datasets. in *proceedings of the the 7th joint meet- ing of the European software engineering conference and the acm sigsoft sympo- sium on the foundations of software engineering. ACM*, 121–130.

[3] Bo¨ hme, M., & Roychoudhury, A. (2014). Corebench: Studying complexity of regression errors. in *proceedings of the 2014 international symposium on software testing and analysis. ACM*, 105–115.

[4] Boucher, A., & Badri, M. (2018). Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology, 96*, 38–67.

[5] Dong, F., Wang, J., Li, Q., Xu, G., & Zhang, S. (2018). Defect prediction in android binary executables using deep neural network. *Wireless Personal Communications, 102*(3), 2261–2285.

[6] Canfora, G., Iannaccone, A. N., & Visaggio, C. A. (2014). Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics. *Journal of Computer Virology and Hacking Techniques, 10*(1), 11–27.

[7] Canfora, G., Lucia, A. D., Penta, M. D., Oliveto, R., Panichella, A., & Panichella, S. (2015). Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability, 25*(4), 426–459.

[8] Catal, C. (2011). Software fault prediction: A literature review and current trends. *Expert systems with applications, 38*(4), 4626–4636.

[9] Catal, C., & Diri, B. (2009). Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences, 179*(8), 1040–1058.

[10]Choudhary, G. R., Kumar, S., Kumar, K., Mishra, A., & Catal, C. (2018). Empirical analysis of change metrics for software fault prediction. *Computers & Electrical Engineering, 67*, 15–24.

[11] Dam, H.K., .Pham, T., Ng, S.W., .Tran, T., Grundy, J., Ghose, A. et al. (2018). A deep tree-based model for software defect prediction. arXiv preprint arXiv:1802. 00921.

[12]Fan, G., Diao, X., Yu, H., Yang, K., & Chen, L. (2019). *Software defect prediction via attention-based recurrent neural network*. Scientific Programming 2019.

[13]Fenton, N., & Bieman, J. (2014). *Software metrics: A rigorous and practical approach*. CRC press.

[14]Gao, K., Khoshgoftaar, T. M., Wang, H., & Seliya, N. (2011). Choosing software metrics for defect prediction: An investigation on feature selection techniques. *Software: Practice and Experience, 41*(5), 579–606.

[15] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

[16]Graves, A. (2012). *Supervised sequence labelling with recurrent neural networks*. Berlin

[17]Heidelberg:Springer-Verlag Graves, A., Mohamed, A. R., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In*2013 IEEE international conference on acoustics, speech and signal processing*. *IEEE*, 6645–6649.

[18]Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2011). A systematic literature review on fault prediction performance in software engineering. *IEEE*

[19]Han, J., Pei, J., & Kamber, M. (2011). *Data mining: Concepts and techniques*. Elsevier. Hosseini, S., Turhan, B., & Gunarathna, D. (2017). A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*.

[20]Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., & Hassan, A. E. (2016). Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering, 21*(5), 2072–2106.

CHAPTER- 11

APPENDIX

10.1 APPENDIX

S.no	code	block	Function Decl	FunctionDecl Rec	For Stmt	For StmtRec	Do Stmt	DoStmtRec	...
0	Int ld,rd,lm,rm;	FunctionDecl,false;	1	0	0	0	0	0	...
1	scanf("%d%d%d%d",&ld,&rd,&lm,&rm);	FunctionDecl,false;	1	0	0	0	0	0	...
2	if(((ld-1)<=(rm)&&((rm/2-1)<=ld))||((rd-1)<=(lm)&&((lm/2-1)<=rd)))	FunctionDecl,false;IfStmt,false;	1	0	0	0	0	0	...
3	printf("YES");	FunctionDecl,false;IfStmt,false;	1	0	0	0	0	0	...
4	else printf("NO");	FunctionDecl,false;IfStmt,false;	1	0	0	0	0	0	...
5	return 0;	FunctionDecl,false;	1	0	0	0	0	0	...

To validate the performance of the Software Defect Prediction Model we use a dataset from the website **https://zenodo.org/record/3268512#.YnIisC8RpmB**