Faculty of Informatics and Computer Science

Software Engineering

Software defect prediction

By: Abdelrahman Khaled Mohamed

Supervised By

Dr. Abeer hamdy

June 2020

Abstract

Defects are common in software systems and can potentially cause various problems to software users, as software systems are getting larger and more complex, more defects can occur. Despite meticulous planning, well documentation and proper process control during software development, occurrences of a certain defects are inevitable. Many software development activities are performed by individuals, which may lead to different software defects over the development to occur, causing disappointments in the not-so distant future. Software defect prediction identifies the modules that are defect-prone and require extensive testing by capturing the syntax and different levels of semantics of source code. Software defect prediction plays an important role in improving software quality and it helps to reduce cost, time, and resources. Different methods have been developed to quickly predict the most likely locations of defects in large code bases. Most of them focus on designing features (e.g. complexity metrics) that correlate with potentially defective code. Those approaches however do not sufficiently capture the syntax and different levels of semantics of source code. However, to overcome this issue, a new technique has been introduced using source code to generate an abstract syntax tree to capture syntax and semantics. In addition to powerful deep learning techniques. This paper used recurrent neural networks and convolutional neural networks to achieve an accuracy of 78%, and 91% respectively.

Acknowledgements

I want to thank God most of all, because without God I would not be able to do any of this.

First, I must start by thanking my supervisor, professor Abeer Hamdy. Thank you for your patience, your help, your presence when needed, your faith, your leadership, your guidance with all the passion throughout the whole year, your support from the early drafts of choosing an idea to develop this system and giving the sense of motherhood. Without your experience and your precision, we would not have gone this far. Thank you.

I want to thank my family, for the support you are giving at home, your faith in me, your presence, and patience ever since teaching me how to hold a pen to become your graduate son. Thank you.

At last, I want to thank all of professors and teaching assistants of the faculty of informatics and computer science of The British University in Egypt. Thank you for the support you have always been giving to us, your teaching experience since we were preparatory students to become senior students.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

As software systems has been playing a critical role in the society in all areas, everything can be controlled from software systems nowadays. Thus, reliability has become a critical issue. This causes that modern software systems became more complex and complicated to satisfy the users' requirements, exposure to be defect prone. Therefore, identifying defects in software systems became difficult due to significant grow of software codebase in complexity and size. In many companies, they employ code review and unit testing to find defects in fresh code. However, manual defect review is labour-intensive, and all code unit testing is impractical. Since the software development budget is limited, it would be beneficial to check the buggy code first. Early defect detection helps to reduce cost, time, and resources. Therefore, software defect predictions are arising nowadays, and made it active research area in software engineering.

Software defect prediction is the process of building classifier to determine if a defect exist in a particular area of the source code files. Software defect prediction plays an important role in improving software quality and it helps to reduce cost, time, and resources. It can assist developers to prioritize their testing effort. Software defect prediction can be either file-level, method-level, package-level, and class-level defect prediction. In this paper we will be focusing on file-level defect prediction. Typical software defect prediction consists of two phases: (1) extracting features from source code files. (2) building classifier model using deep learning algorithms to create model for training and validation.

1.2 Problem Statement

40% of the developments' budget in software development life cycle (SDLC) is spent on testing phase only [5]. It is recommended to check the defected code only to reduce cost, time, and resources. In addition to manual defect review is labour-intensive and impractical. On the other hand, to increase quality attributes as reliability, maintainability, and performance efficiency. As technology increase, clients' requirements and expectations are getting above the sky, as they aim to develop an idea that meets with modern software systems. Thus, complexity is getting higher which may lead to defects in software systems

1.3 Scope and Objectives

Our goal is to extract textual features from java source code files and place it in a correct form, in order to build an accurate software defect prediction system using deep learning techniques to classify whether fresh code is defect prone or not, to help in reducing cost, time, and resources on testing in software development life cycle. Building this system will also improve quality attributes such as reliability, performance efficiency, and to satisfy user's requirements.

1.4 Report Organization (Structure)

This report is structured as the following. Section 1 discusses overview of software defect prediction process, problem statement, scope and objectives, and work plan. Section 2 discusses background information of related work, literature survey, and analysis of related work. Section 3 discusses the proposed solution methodology, pre-processing, proposed convolutional neural network, and proposed recurrent neural network. Section 4 discusses implementation. Section 5 discusses results. Section 6 discusses summary, conclusion, and future work.

1.5 Work Methodology

The work methodology is distributed as the following: (1) Searching for a proper idea. (2) Read previous work done in the problem domain. (3) Choosing proper dataset. (4) Writing interim report (5) Pre-processing. (6) Building deep learning models. (7) Train each model with different hyperparameters. (8) Test each model. (9) Compare results. (10) Write thesis. Each of these phases will be explained individually below.

We will start by choosing our dataset, our dataset is subset of the original dataset which was simplified PROMISE source code (SPSC) and PROMISE source code (PSC). This dataset contains source code files, and labels to be able to differentiate whether it is bugged or not. Secondly, pre-processing to the source code files to make it extract features, then building deep learning network. After building the network, the pre-processed dataset should be divided into 70% training, and 30% for validation. Thereafter, a report will be generated to analyse our report with other researchers.

For more detailed information on simplified PROMISE source code (SPSC) and PROMISE source code (PSC) you can check Table.1

Project	Number of Defects	Number of Files	Avg Buggy rate
Poi	529	818	64.7
jEdit	134	547	24.5
Xerces	138	882	15.6
Synapse	141	461	30.6
Xalan	790	1629	48.5
Lucene	234	420	55.7
Camel	333	1781	18.7
Total	2299	6538	35.2

Table 1. Simplified PROMISE Source Code (SPSC) dataset description.

1.6 Work Plan (Gantt Chart)

Figure 1. Gantt chart

2 Related Work (State-of-The-Art)

2.1 Background

Software systems have different types of errors, such as compilation error, run time error, logical defects. In our paper we will be focusing on logical defects. Logical defects mean errors written in the source code that will deliver unexpected result/behaviour to the user. For example, in figure [2], shows a sample of logical errors which may occur, two simple java code files. Both contains a *for* statement, *add* function and *remove* function to a queue. The only difference between then is the order of the *add* and *remove* functions. File2.java will rise *NoSuchElementException*, when it calls remove function if the queue is already empty.

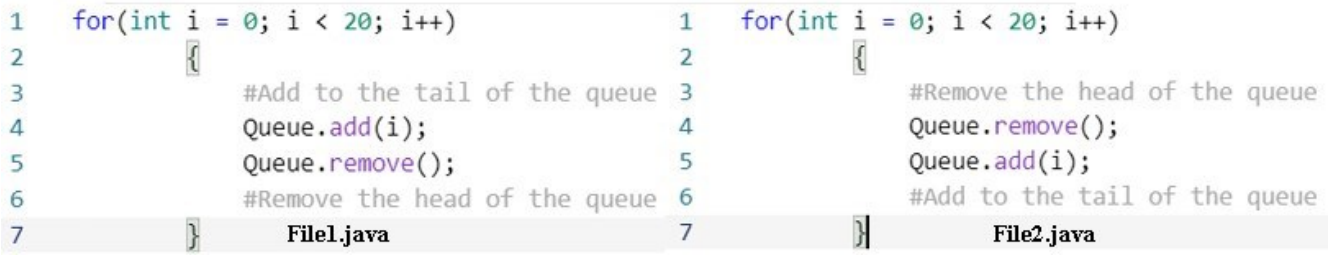

Figure 2. Motivating example

There are different types of source code defect predictions which can be categorized to be file-level, method-level, package-level, and class-level defect prediction. In this paper we focus on File-level defect prediction. File-level means that each test instances is a source code file. Most of the modern techniques for software defect prediction problem uses deep learning techniques. Deep learning is a machine learning technique based on artificial neural networks. Deep learning techniques have shown better results that other machine learning algorithms as it has powerful ability to run non-linear model and has the ability to generalize. Deep learning is a lead in machine learning nowadays. There are three different types of neural networks is deep learning, (1) Artificial neural networks (ANNs), (2) Convolution neural networks (CNNs), (3) Recurrent neural networks (RNNs). Each of these networks will be explained individually below.

2.1.1 Convolutional neural network

Convolutional neural networks (CNNs) are special type of neural networks for processing data that has a known, grid-like topology. For example, time-series data one-dimensional (1D) as well as two-dimensional data (2D) and three-dimensional image data (3D). Convolutional neural networks have been shown to be successful in many practical areas, including facial recognition, image classifications, speech recognition and natural language processing (NLP). In this paper, we leverage a convolutional neural network (CNN), and recurrent neural network (RNN) models to extract features and classify java source code files. In a basic fully connected network, which we call dense network, neuron units are fully connected to all neuron units of its neighbouring layers. Which means each neuron can influence every single neuron in the next layer (L + 1). At the same time, the neuron itself is dependent on every single neuron from the previous layer (L - 1) as stated in Figure [3].

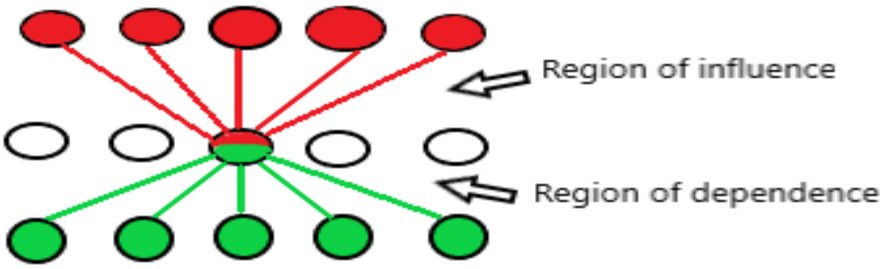

Figure 3. Region of influence/dependence in simple ANN

Figure [4] depicted the architecture of a basic convolutional neural network. A basic convolutional neural network consists of convolutional layers and pooling layers. In convolutional neural network, neurons units connected to these two layers are sparsely connected, This connection is determined the kernel size and pooling size. Pooling layers in convolutional neural network is usually combined between two convolutional layers. The pooling layers are used to reduce the number of parameters by down-sampling the representation. Pooling helps to make data representation more robust. There are different types of pooling layers. For example, max-pool layers, and average-pool layers. The convolutional layers are used to extract features from the given dataset. Convolutional layers contain a set of filters whose parameters need to be learned. A convolutional neural network has two characteristics that makes them powerful,

that are parameter weight sharing and sparse connectivity. These two features help to reduce model's capacity and gain the ability to learn local patterns rather than global patterns which are offered by basic fully connected network.

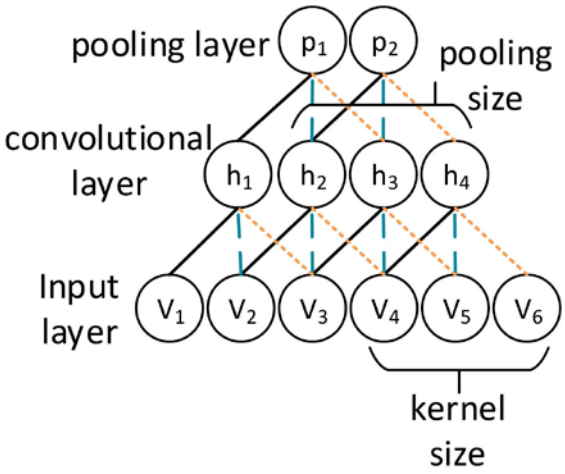

Figure 4. Simple convolutional neural network architecture.

Sparse connectivity means that every single neuron unit is only connected to a specific limited number of other neurons in neighbouring layers. In convolutional neural network, spare connectivity is being controlled by pooling size and kernel size. For example, node V_4 in Figure [3], when kernel size = 3, it is connected to three nodes only in the convolutional layer which are h_3, h_4, and h_2, although, h_1 is not affected by V_4 node. Furthermore, h_3 is affected only v_4, v_5, and v_3 nodes. It means that convolutional neural network employs a local connectivity pattern between nodes of adjacent layers. Each subset that connects to the next layer of the convolutional neural network is called a local filter, and it captures a particular type of pattern. In order to be able to calculate the output from the previous layer to the next layer, multiply each local filter with the output from the previous layer then add bias and perform a non-linear transformation. In figure [3], if we symbolize that i^{th} node in the n^{th} convolutional layer as h_i^n, the weights of the i^{th} node in the $(n - 1)^{th}$ layer as $W^{m-1}{}_i$, the bias in the previous layer $(n - 1)^{th}$ layer as b^{n-1}, and the ReLU or Sigmoid as an activation function, thus, final equation will be calculated as the following:

$$h_i^n = \mathrm{ReLU}(w^{n-1}{}_i * x^{n-1}{}_i + b^{n-1})$$

$$h_i^n = \text{sigmoid}(w^{n-1}_i * x^{n-1}_i + b^{n-1})$$

For the other characteristic that convolutional neural network has is shared weight. Shared weight means that each filter in the network share the same parameters. For example, in Figure [3] blue lines, orange lines, and the black lines connecting input nodes in the input layer and convolutional nodes in the convolutional layer share the same parameters (weights and bias), Shared weight enable convolutional neural network to detect features independent of their position in the input vector. Moreover, weight sharing has a great impact to reduce model capacity.

Dense neural networks	Convolutional neural networks
Dense	Sparse
Learns all parameters every time	Parameter sharing
Not equivariant	Equivariant representations

Table 2. Comparison between DNN and CNN

2.1.2 Recurrent neural network

Recurrent neural networks (RNNs) is a another of artificial neural networks (ANNs) where they are good at modelling sequence data. Recurrent neural networks take the input as the same output that came from the previous one. In other words, if next data point depends a lot on previous ones, then recurrent neural networks are the best to use as represented in figure [4]. Recurrent neural networks are widely used in stock predictions, text generation, voice recognition. Recurrent neural networks are learning though the gradient decent. During back propagation where weights are being updated, recurrent neural networks suffer from vanishing gradient problem. The gradient is the value used to update the weight of the neural network. The vanishing gradient problem is when gradient strength becomes extremely small, thus it does not contribute too much learning. because layers do not learn. Recurrent neural network can forget what is seen and longer sequences does have short term memory.

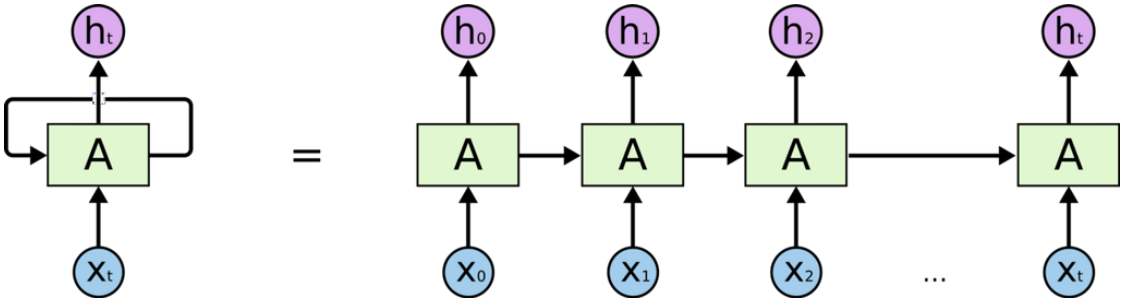

Figure 5. Sample RNN structure and unfolded representation

Long short-term memory (LSTM) is the more evolved version of recurrent neural networks. Long short-term memory was generated to act as the solution to short-term memory that recurrent neural network has. They have an internal mechanism called a gate that can regulate the flow of information. These gates can learn what data in the stream is important to keep or delete. This allows you to learn related information and make predictions.

The main idea of LSTMs are cell state and the gates. The cell state transfers relative information sequence chain. Cell state can be thought as the memory of the neural network. For The Reason That, cell state can carry data throughout the sequence processing even information from earlier times steps could be carried all the way to the last time step. Thus, reducing the effect of short-term memory that appeared in recurrent neural networks. Information gets added or removed to the cell states through gates. The gates are just different neural networks that decides which information is allowed on the cell state. The gates learn what information is relevant to keep or forget during training. Gates in long short-term memory contains sigmoid activation. Sigmoid squeezes the input values to be between 0 & 1. We can take the advantages of 0's and 1's that the sigmoid offers to update/forget the data, thus, any number getting multiplied by 0 is 0 this means the value is going to be thrown and any number getting multiplied by 1 does not change, therefore that value is kept. The network can learn what data should be forgotten or what data is important to keep.

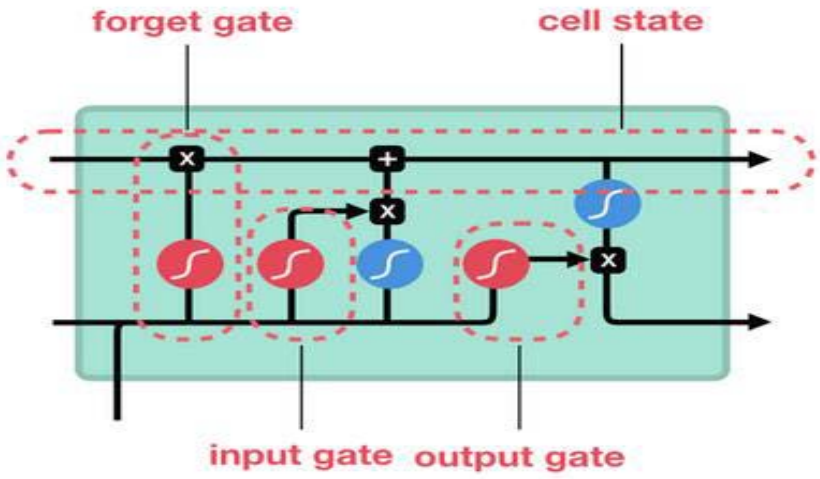

Figure 6. LSTM gates

There are different gates in the LSTMs, Forget gate, Input gate, and output date. The Forget gate is the one that decides what information should be neglected/dropped. Information from previous hidden state and information from current input is passed to the sigmoid function, values come out between 0 and 1, the closer to 0 means neglect/drop and closer to 1 means to keep. Input gate is used to update the cell state, there are many steps to calculate the input gate, (1) passing current input value and the previous hidden state value to the sigmoid function that decides which values will be updated by transforming values to be between 0 and 1. (2) Passing current input value and hidden state value to the tanh activation function to squeeze values between -1 and 1, this helps to control the flow of the network. (3) multiplying the output from the sigmoid activation function with the output of the tanh activation function, the sigmoid output will decide which information is important to keep from the tanh output. Output gate is the last one, this gate decides what the next hidden state should be, hidden state contains information of previous inputs. (1) Passing value from previous hidden state in the current input to sigmoid function. (2) Passing the newly modified cell state to the tanh function. (3) Multiplying sigmoid output with tanh output in order to decide what information the hidden state should carry. The output is the hidden state. At last, the new hidden state and new cell state are then carried over to the next time step.

There are many hyperparameters in the convolutional neural network and recurrent neural network to discuss, for example, pooling size, filter size, batch size, learning rate, number of epochs. These hyperparameters must be tuned in order to achieve better accuracy for the model. Hyperparameters and experiments will be discussed in section 5.

2.1.3 Dropout

Dropout is a network model that is used to handle overfitting problems[10]. Dropout main concept is to drop some neural units, weights and connections during training time and they are randomly chosen, to prevent complex co-adaptation of neural units and to reduce model generalization error. Srivastava, Hinton, Krizhevsky, and Sutskever are the ones who proposed the concept of co-adaptation.[10] During backward propagation, weights of neural units are being updated depending on what other neural units are doing, hence, the weight may possibly be updated to compensate for the errors of other neural units, which is called co-adaptation. Therefore, adding a dropout layer, a neural unit is unreliable because it may be randomly chosen to be the dropped unit. Hence, each neural unit learns better features instead of correcting errors in other neighbouring neural units.

Figure [7] demonstrates the mechanism of dropout. For example, the dropout probability is 0.5, thus, when adding a dropout layer in the hidden layer, neural units in the hidden layer are chosen randomly to be dropped with half of the neural nodes, as we stated above the dropout probability is 0.5, as the figure shows, V_2 and V_3 has been dropped. The connections between the two neural nodes and the output/input layers with the weights are also dropped. In such a case, the weight update of V_1 and V_4 would be independent of V_2 and V_3, as a result of the dropout, it will reduce model generalization error and prevent co-adaptation of neural units.

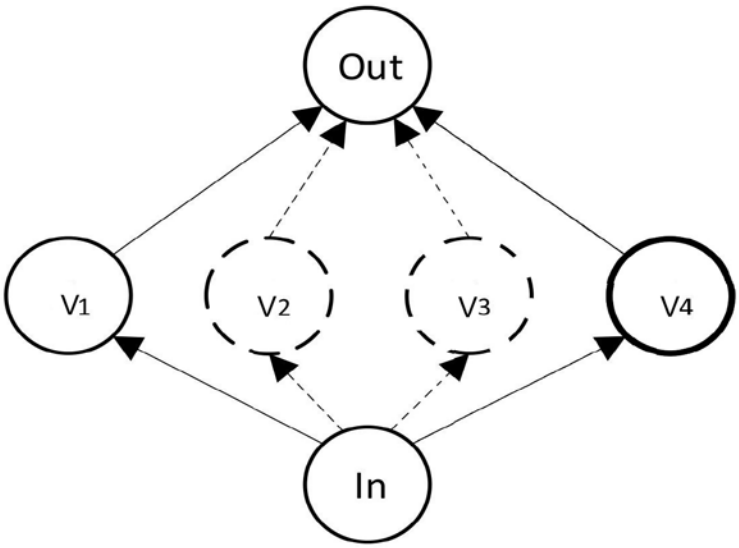

Figure 7. Mechanism of dropout

2.1.4 Overfitting

Overfitting is one of the biggest problems in training a neural network model[11]. Overfitting means that the neural network model at a certain time during the training period does not improve its ability to solve problem anymore. Overfitting is a problem because the statistical models are hard to generalize. Overfitting is the variance between training error and test error is large. The neural network model just starts to learn some random regularity contained in the set of training patterns. Overfitting means that the neural network model has high variance and low bias.

Figure [8] demonstrates the idea of overfitting, where the neural network model is not able to separate the data, the model does exactly know each instance of the data to specify which class the instance should be held in either red or blue class.

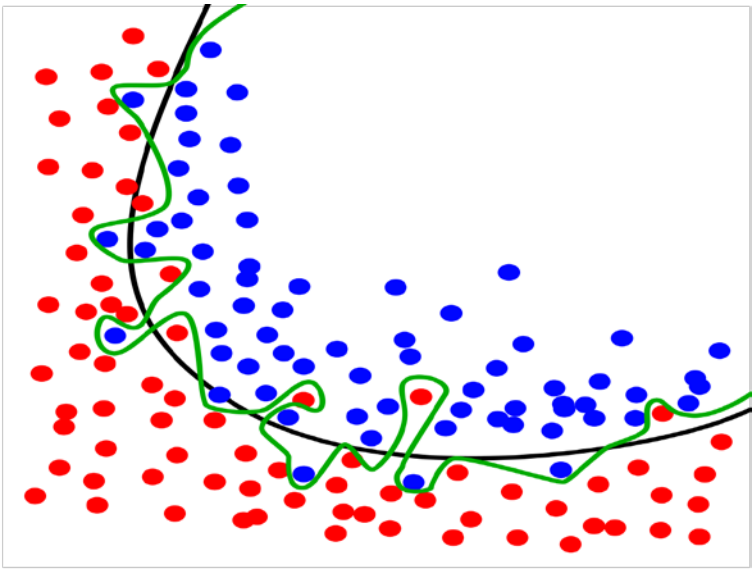

Figure 8. Overfitting

Moreover, there are many techniques used to overcome overfitting. For example, dropouts, data augmentation, L1 regularization, L2 regularization, reduce neural network capacity, early stopping [11]. Dropouts have been introduced in section 2.1.3, noise injection is effective because artificial neural networks are not robust to noise. Data augmentation is very effective for classifications, on the other hand, data augmentation is not effective in predictions domain.

2.1.5 Underfitting

Underfitting is the opposite of overfitting [11]. Underfitting occurs when neural network model is not capable to capture the underlying features of the data. Underfitting means the neural network model has high bias, low variance. Underfitting results in low model generalization and unreliable predictions/classifications. Some techniques are used to solve the underfitting problem as, increasing model complexity, increase number of features to learn, remove noise from data, and increase number of epochs in training model.

Figure [9] demonstrates the main concept of underfitting. Where the neural network model is not able to separate the data in a good fit, where the variance between the line and the instance is high.

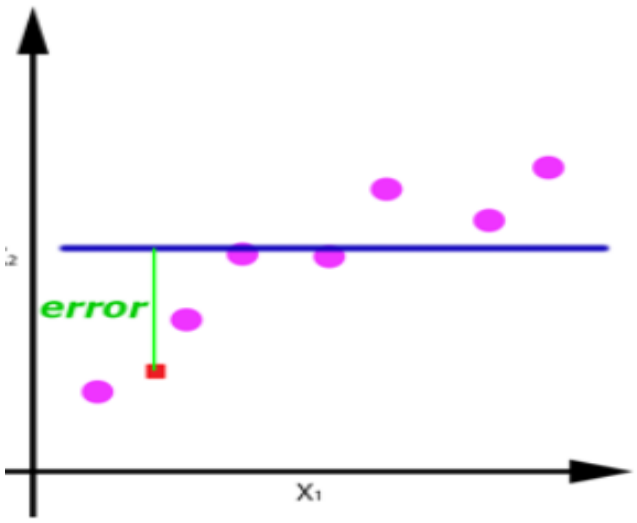

Figure 9. Underfitting

.

2.2 Literature Survey

Previous research approaches on software defect prediction has taken the researchers to one of these two directions: one is creating new features or using combinations of existing features for better accuracy. Scientists and researchers have designed hand-crafted metrices to detect defect characteristic for example, Halstead features based on operator and operand counts, McCabe features based on dependencies, and other comprehensive feature sets including Chidamber and Kemerer (CK) features, metrics for object-oriented design (MOOD) features, and code change features. As for the second direction, machine learning techniques, as many machine learning models have been designed to classify data for example, decision tree (DT), Support vector machine (SVM), Random forest (RF), Logistic regression (LR), Naïve Bayes, and dictionary learning models. Those approaches however do not sufficiently capture the *syntax* and different levels of *semantics* of source code as software metric features work well in some projects and may not perform well in other projects. [3] [2]

Natural language processing techniques also has been introduced to software defect prediction to extract defect prediction from code tokens in source code files. For example, Bag-of-words (BoW). It treats code tokens as terms and represents source file as term-frequencies. Although, bag-of-words techniques did perform well in software defect prediction as they're unable to detect differences in the semantics of source code due to syntactic structure or different in code order for example (e.g. $z \geq q$ vs. $q \geq z$). [2]

Unlike natural language, they can define the syntax well in Backus-Naur form, then it can be structured in abstract syntax tree (AST). Abstract syntax tree is a tree to represent the syntactic structure of source code files. Abstract syntax tree has proven the capability to capture the syntax and different level of semantics of source code.

Deep learning techniques have been also introduced to software defect prediction, as Cong's conventional neural network (CNN) [3], Recurrent neural networks (RNN), and Long-short term memory (LSTM) [2]. Conventional neural networks and Recurrent neural networks shows amazing results on dataset, but it differs from one dataset to another. Thus, we cannot tell which is better than the other yet.

In Li's research paper [1], he introduced Conventional neural networks, Abstract syntax tree (AST), word embeddings and Deep belief network (DBF) to create his modified model called (DP-CNN). This approach helped Li to learn semantic and structural features of source code. Li's dataset was seven open source projects. And at the end he managed to increase in F-measure performance by 16% better than deep-belief-network-based which was 14.2% in Song's research paper [7].

In Cong's research paper [3], he improved Li's conventional neural network on the same dataset and platform. Cong added more hidden layers, more pooling layers, and dense layer to improve the accuracy in Li's conventional neural network [1]. Added skip-gram model and dropout layer to the CNN. The result shows that Cong's CNN had improved the F-measure by 6%.

2.3 Analysis of the Related Work

As discussed in the previous section, many researchers worked in the field of software defect predictions, as it is important to have a software free of bugs, and reliable. Each of them implemented his project on his own way having a different technique, different models. Nevertheless, the dataset has been the same across each project done by each researcher in order to be able to compare the work done by each one.

Table [3] demonstrates the performance between each project made by researchers and specific files. Therefore, you will see in the table the project's names, dataset files, and the technique used to solve software defect prediction problem. Traditional logistic regression had the lowest performance among other techniques with an average of 0.524. Deep-belief-network (DBN) was the third to compete upon other techniques/models, Deep-belief-network managed to achieve the highest performance in Poi file with a performance of 0.789. Nevertheless, Deep-belief-network still had a low average performance with 0.543 which is not a great improvement against the traditional logistic regression. Li's CNN introduced the concept of abstract syntax tree to capture syntax and different levels of semantics, used convolutional neural network to achieve an average performance measure of 0.596, and to be the lead to achieve the best in Camel, jEdit, and Lucene files. At the end, Cong's convolutional neural network improved version of Li's convolutional neural network. Cong managed to achieve highes performance in Xalan, Xerces, and Synapse to achieve an average performance of 0.618 to be the hights among other researchers.

Project	Traditional (LR)	DBN	Li's CNN	Cong's CNN
Camel	0.329	0.335	**0.505**	0.487
jEdit	0.573	0.480	**0.631**	0.590
Lucene	0.618	0.758	**0.761**	0.701
Xalan	0.627	0.681	0.676	**0.780**
Xerces	0.273	0.261	0.311	**0.667**
Synapse	0.500	0.503	0.512	**0.655**
Poi	0.748	**0.780**	0.778	0.444
Average	0.524	0.543	0.596	**0.618**

Table 3. Performance comparison.

3 Proposed solution

3.1 Solution Methodology

Figure [10] demonstrates a file-level software defect prediction process. As depicted in the figure, the first step of building software defect prediction model is to extract file instances/modules (source code files) from java source code repositories. The second step is to extract textual features from each java source code files. There are many traditional features defined in the past studies such as, CK features, and McCabe features as explained in section 2.2. Nevertheless, in this paper, other features will be used for our defect prediction process which will be discussed in section 3.2. The third step will be labelling the java file instances as clean or buggy files. The labelling process is based on post-release defects from several bug tracking systems. A source code file is labelled as bugged if it contains at least one bug report. Otherwise, it is labelled as a clean file. The fourth step is to generate a training instances from the labelled dataset. The fifth step is to build the deep learning classifier model and use the training instances to train the model. Finally, fifth step a new instance are fed to the classifier for evaluation which will be discussed in section 5.

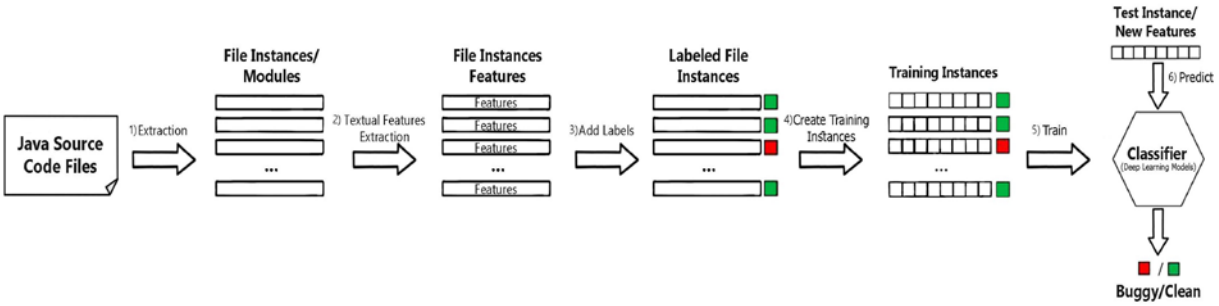

Figure 10. Solution methodology

3.2 Pre-processing

As shown in figure [11], the pre-processing process consists of five parts. First, we take java source files and pass it through a *lexer* to generate tokens, this operation is called *tokenization.* Afterwards, tokens generated from previous operation will be passed to a *parser* generate an abstract syntax tree (AST) with specified nodes following the state-of-the-art method as stated in figure [11]. Finally, the generated abstract syntax tree will be passed to a vectorizer to map the string token abstract syntax tree into input vectors to be fed to the proposed convolutional neural network and recurrent neural network.

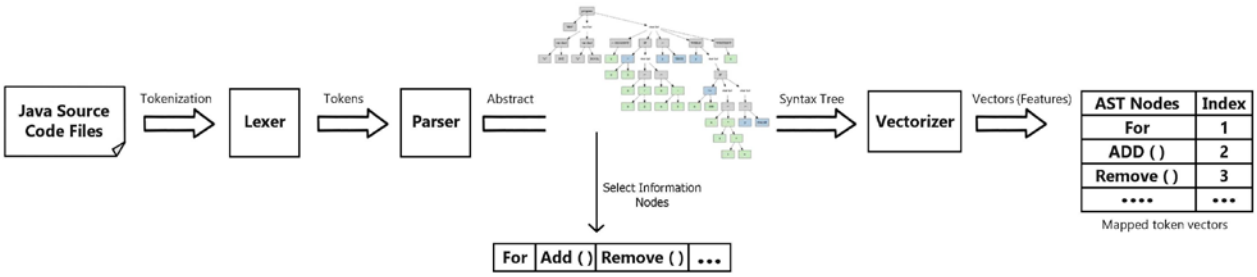

Figure 11. Workflow of the pre-processing

3.3 Parsing source code

Considering that the java source code is used as an input for the pre-processing part, a proper code representation was deemed to be beneficial for parsing source code. Code representation include tree-level, character-level, graph-level, token-level, path-level, AST-node-level. The abstract syntax tree node level (AST-node-level) code representation have demonstrated to be better to knock token-level, character-lever, and representations of higher granularities in classification tasks [12]. Therefore, to build a classifier which preserve syntax and different level of semantics, abstract syntax tree node level should be the used as a code representation.

According to Li[1], following the state-of-the-art method[7], only three types of abstract syntax tree nodes are extracted as tokens: (1) declaration nodes, such as type declarations, enumeration declarations, and method declarations, which are represented by their values. (2) nodes of method invocations and class instance creations, which are represented as class names or method names. (3) control-flow nodes including while statements, for statements,

catch clause, throw statements, if statements. Control-flow nodes are represented by node type. There are certain nodes types of the abstract syntax tree which are excluded for example, assignment and package declarations. The nodes were excluded for different reasons: (1) Frequency of the node is low. (2) Information in the nodes is untraceable. (3) Some of these nodes are class-specific or method-specific, which does not have a meaning throughout the whole project. The selected abstract syntax tree nodes are demonstrated in the following figure [12].

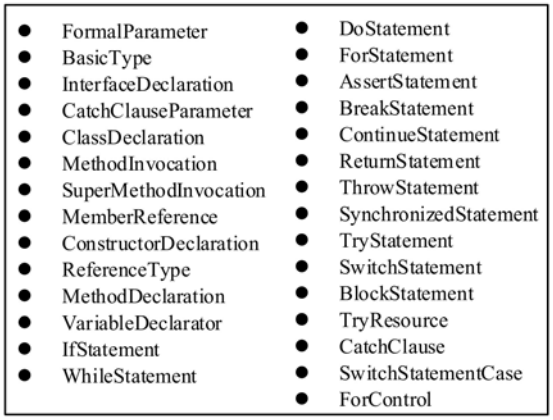

Figure 12. Selected abstract syntax tree nodes

In these experiments, an open-source python package called *javalang* to parse the java source code files to abstract syntax tree, as javalang provides a lexer and a parser targeting java files. Javalang has a limited functionality, hence, many java files may not be parsed correctly. In order to solve this issue three strategies are applied, (1) delete the file. (2) fix the syntax error in the java file where javalang could not parse. (3) delete only the part that javalang could not parse. For simplicity, the first strategy where adopted.

A motivating example of java source code demonstrated in figure [12]. After parsing the piece of code using javalang, 39 nodes were generated, starting with CompilationUnit and ends with MemberReference. After selected the specified nodes which are listed above, a list including 26 nodes was generated, which starts with ClassDecleration and ends with MemberReference. Nevertheless, Assigment node, BinaryOperation node, and CompilationUnit node are excluded according to the criteria listed above.

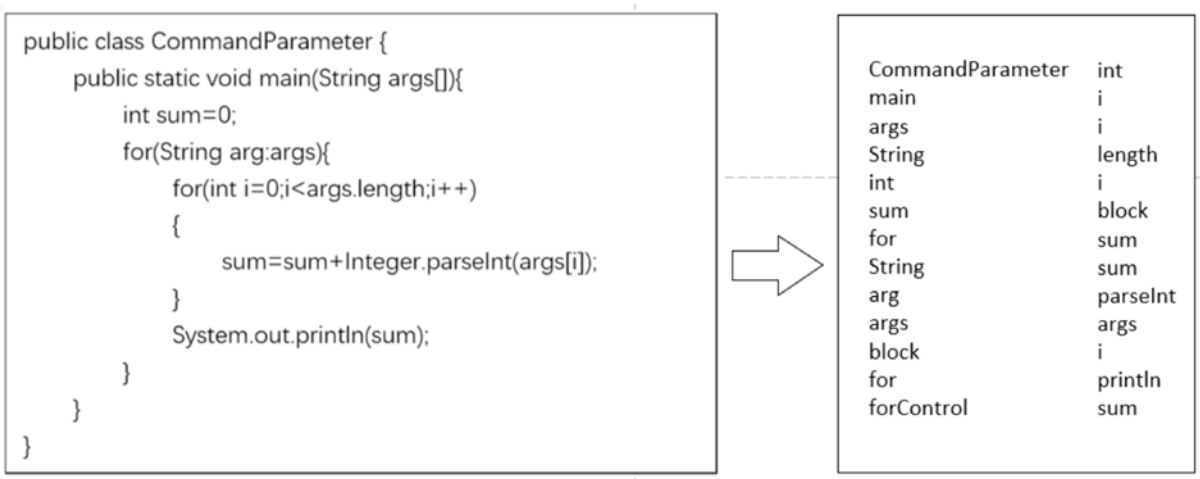

Figure 13. Java source code and the extracted abstract syntax tree nodes

Another example of to illustrate the mechanism of java source code parsed to abstract syntax tree (AST) in the figure below. The piece of code contains while loop followed by variable declaration.

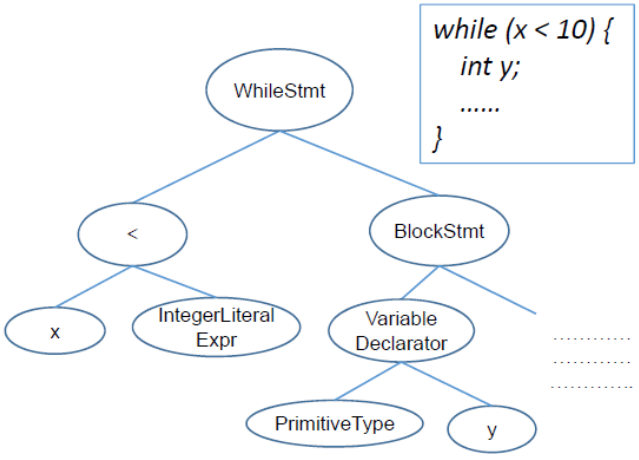

Figure 14. Java source code to abstract syntax tree (AST)

3.4 Mapping tokens

The next step after parsing source code and generating the abstract syntax tree (AST) with the specified nodes, a token vector for each java source code file was obtained. Nevertheless, these token vectors could not be fed directly as an input for the deep learning models as deep learning models can have only integers as an input. Therefore, a mapped version should be generated to map tokens from strings to integers. Moreover, the conversion that mapped each string token to an integer should range from one to the total number of token types, each token string will be represented by a unique integer. In addition, the deep learning models requires an input vectors with same dimensions/length. However, the length of the input vectors which are extracted from the abstract syntax tree will vary after conversion. To overcome this problem, a digit zero should be appended to the integer vectors to make their lengths equal to the longest vector. As stated above, mapping starts from one, hence, adding a zero will not affect mapping space. The used vectorizer in this paper is term frequency-inverse document frequency (TF-IDF), this is used with the purpose of evaluate the significance of each word present on java source code files.

3.5 Convolutional neural network

According to Li [1] and Cong [3], convolutional neural network has proven to be good for defect prediction, Although Li[1] achieved a good accuracy, Cong improved Li's model to achieve better accuracy. In this paper, a new convolutional neural network architecture have been introduced to achieve better results than any, which will be discussed in section 5. The overall architecture of the convolutional neural network is shown in figure [15]. Moreover, for the proposed convolutional neural network consists of four convolutional layers followed by a max-pooling layers in between to extract global patterns, followed by a flatten layer, six dense layers with dropouts in between with a probability of 0.6, to achieve better generalization, and finally a logistic regression classifier to predict whether java source code is clear/bugged.

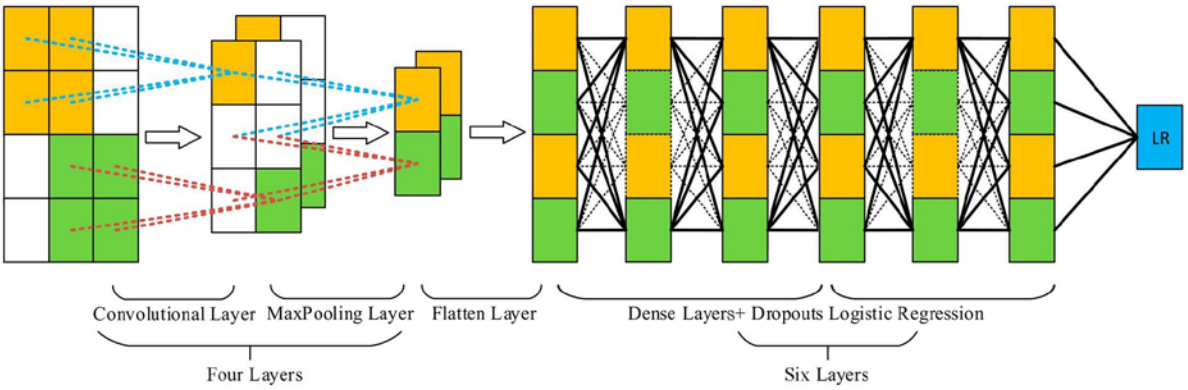

Figure 15. Proposed convolutional neural network architecture

Below is other detailed information about the architecture.

- Implementation framework: Keras where it is available at (https://keras.io/). Keras is a high-level API based on tensorflow. Tensorflow used is version 2.0
- Google Colab: could service with high CPUs and GPUs to improve your performance and develop deep learning projects.
- Parameter initialization: 85 epochs, 16 batch size
- Dropouts: Were added to prevent overfitting and to have better generalization, dropout with a value 0.6
- Four convolutional layers and in between max-pooling layers: It is known universally to increase the depth of the model to have greater powers and to have better results.
- Activation functions: ReLU activation functions everywhere, except for the last dense layer, a sigmoid activation function used.
- Optimizer: Adam with a learning rate of 0.0001

The following table is a comparison between Li's architecture, Cong's architecture and this improved CNN. Parameters are listed above to apply to the same dataset which is simplified PROMISE source code dataset.

	Li's CNN	Cong's CNN	Improved CNN
Convolutional layers	1	3	4
Dense layers	1	3	6
MaxPooling layers	1	3	4
Activation functions	sigmoid + ReLU	sigmoid + ReLU	sigmoid + ReLU
Dropouts	None	Between dense layers	Between dense layers (probability of 0.6)
Regularization	None	L2 regularizations	None
Optimizer	Adam	Adam	Adam with learning rate 0.0001
Loss function	-	Binary cross-entropy	Binary cross-entropy

Table 4. Architectural comparison

3.6 Recurrent neural network

Recurrent neural networks have proven to be the lead in modelling sequence data such as speech recognition and text translation [9]. Hence, I had to build a recurrent neural network to compare results with the improved convolutional neural network model. The overall architecture of the proposed recurrent neural network is shown in the figure below. My recurrent neural network consists of LSTM layer with 512 cells, followed by flatten layer and finally a dense logistic regression classifier to predict whether java source code is clear/bugged.

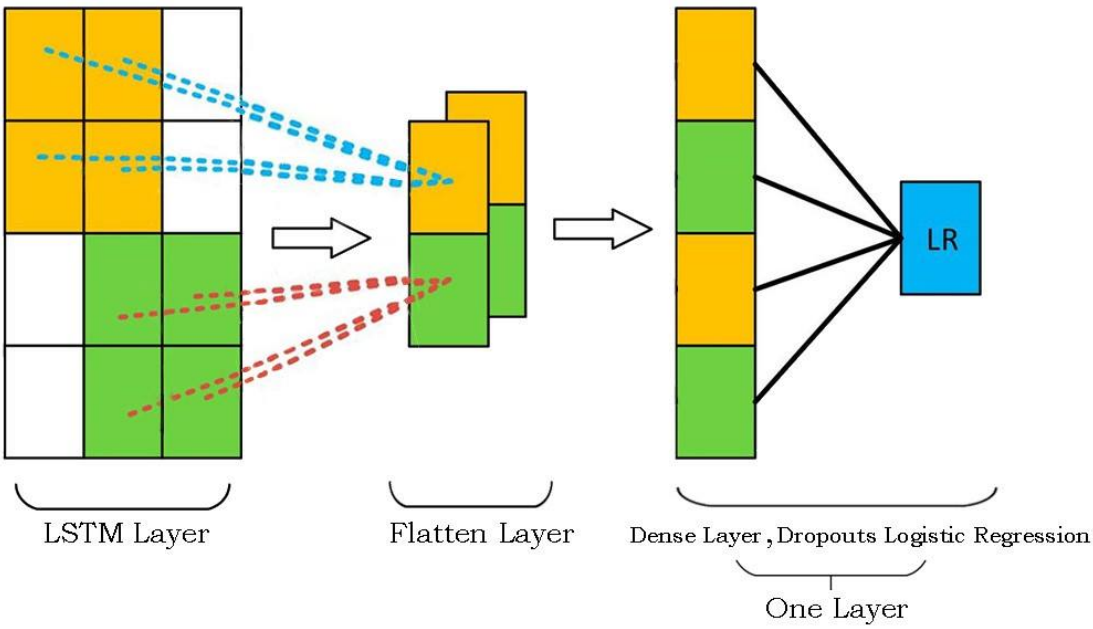

Figure 16. Proposed recurrent neural network architecture

Below is other detailed information about the architecture.

- Implementation framework: Keras where it is available at: (https://keras.io/). Keras is a high-level API based on tensorflow. Tensorflow used is version 2.0
- Google Colab: could service with high CPUs and GPUs to improve your performance and develop deep learning projects.
- Parameter initialization: 15 epochs, 4 batch size
- Activation functions: Sigmoid activation functions.
- Optimizer: Adam with a learning rate of 0.0001

4 Implementation

The software defect prediction has been written in python 3 language, as python is the lead language in machine learning and deep learning techniques, Moreover, python offers libraries which will be used to build a software defect prediction model. Pre-processing part has been written on PyCharm IDE, and the deep learning models are written on Google Colab. The environment included installing javalang library, pandas library, NumPy library, sklearn library and ntlk library. Furthermore, to build a deep learning model on google colab it requires to include tensorflow, keras models, keras layers, keras optimizers.

First of all, my pre-processing has been the following: (1) Gathering all the dataset in one file to be able to locate the path of the file. (2) loop on the dataset's size to get each source code file from its path. (3) Take each file and pass it through lexer and parser of javalang library. (4) mapping token vectors into integer vectors. (5) Save vectors in excel sheet to be fed to the deep learning mode. As shown in Figure [17], filesName variable has all file names from 'defectPrediction.csv', while path variable has the path to the source code file to capture. Inside the for loop, we call parser class, which has method called preprocessing().

```
if __name__ == '__main__':
    parser = Parser('')  # Create object of the class parser

    filesNames = pd.read_csv('defectprediction.csv')  # Read all files names
    path = filesNames.iloc[:, 1]  # select column 1, all rows
    filesNames = filesNames.iloc[:, 0]  # select column 0, rows from 0 to length
    filesNames = np.array(filesNames)  # Convert to numpy array
    path = np.array(path)  # Convert to numpy array
    foundsrc = 0
    notdound = 0
    fileNum = 1
    for i in range(path.shape[0]):
        fileNum+=1
        #print("CURRENT\n")
        #print(filesNames[i]," File number: ", fileNum)
        try:
            fh = open(path[i] + ".java", 'r')
            foundsrc += 1  # Increment found counter
            parser = Parser(path[i] + ".java")
            parser.pre_processing()

        except FileNotFoundError:
            notdound += 1  # Increment not found counter
            data = {'Name': [filesNames[i]],
                    'Path': [path[i]],
                    'Status': ['NotFound']}
            df = DataFrame(data)  # add them to data frame
            df.to_csv('NotFoundDefectPrediction.csv', mode='a', index=False, header=False)  # Write missing files in csv

    print("\n\nfound\t\t", foundsrc)
    print("notfound\t", notdound)
```

Figure 17. Pre-processing code 1

In method pre-processing, we declare and initialize a variable AST, AST will hold the abstract syntax tree of the given source code file, abstract syntax tree is generated by javalang.parse.prase(). Then we initialize the vectorizer, TF-IDF vectorizer and send a parameter for the sopping words. Finally the vectorizer is fed to the abstract syntax tree to map token vectors of the tree to integer vectors that could be fed to the deep learning model.

```
        # Source parsing
        try:
            AST = javalang.parse.parse(src)  # This will return AST
            for path, node in AST:  # Index, Element
                if 'ReferenceType' != node:
                    AST.remove(node)
                print(node, "\n")
                # print(path,"\n")
        except:
            pass

    vectorizer = TfidfVectorizer(stop_words='english')  # Create the vectorize/transform

    vectorizer.fit([str(AST)])  # Learns vocab " CompilationUnit, Imports, path, static, true, util, io "
```

Figure 18. Pre-processing code 2

5 Results and Discussions

As convolutional neural network-based deep model, and recurrent neural network-based deep model requires hyperparameters tuning. Thus, a various hyperparameters were tuned as deep learning does not have a literature rules that might indicate which hyperparameters tuning are the best, because it changes from one project to another. Nevertheless, deep learning does rely on testing to determine the best hyperparameters for each project. The tables below will demonstrate the different hyperparameters which were tested on our simplified PROMISE source code (SPSC) dataset.

Layers	Optimizer	Batch size	Dropouts	Accuracy
1 convolutional 1 dense	SGD	256	0.5	45%
1 convolutional 2 dense	SGD	128	0.3	48%
1 convolutional 3 dense	ADAM	16	0.3	45%
3 convolutional 3 dense	RMSPROP	32	0.2	35%
3 convolutional 2 dense	RMSPROP	128	0.6	50%
4 convolutional 3 dense	RMSPROP	256	0.6	55%
4 convolutional 4 dense	SGD	256	0.2	50%

5 convolutional 6 dense	ADAM	16	0.5	60%
3 convolutional 3 dense	ADAM	32	0.5	68%
5 convolutional 5 dense	ADAM	32	0.4	65%
5 convolutional 4 dense	ADAM	64	0.5	70%
2 convolutional 6 dense	SGD	64	0.5	69%
4 convolutional 7 dense	SGD	128	0.3	70%
4 convolutional 5 dense	SGD	64	0.3	80%
5 convolutional 5 dense	ADAM	32	0.5	80%

Table 5. CNN Experiments

Layers	Optimizer	Batch size	Accuracy
3 LSTM **3 dense**	SGD	16	52%
1 LSTM **3 dense**	SGD	32	53%
2 LSTM **3 dense**	ADAM	8	58%
2 LSTM **4 dense**	ADAM	4	60%
1 LSTM **3 dense**	ADAM	4	65%
1 LSTM **4 dense**	SGD	2	60%
2 LSTM **2 dense**	SDG	8	55%
2 LSTM **1 Flatten** **2 dense**	ADAM	8	61%
2 LSTM **1 Flatten** **3 dense**	ADAM	4	68%

Table 6. RNN Experiments

To calculate defect prediction performance, two metrics were used, F-measure and Accuracy. The two metrics were used under different experimental scenarios to achieve best performance. These metrics have been used widely in different software defect predictions studies [1][2][3]. Google colab GPUs were used for deep learning and normal CPUs were used for the pre-processing.

Before getting into F-measure calculations, the confusion matrix should be introduced first. A confusion matrix is a table often used to show the performance of a deep learning model on the test set data which are labels are known. The confusion matrix generates a summary report of the prediction results of the deep learning model. It shows how the model gets confused when it predicts when comes to the test data.[13][14] The figure below demonstrates the confusion matrix.

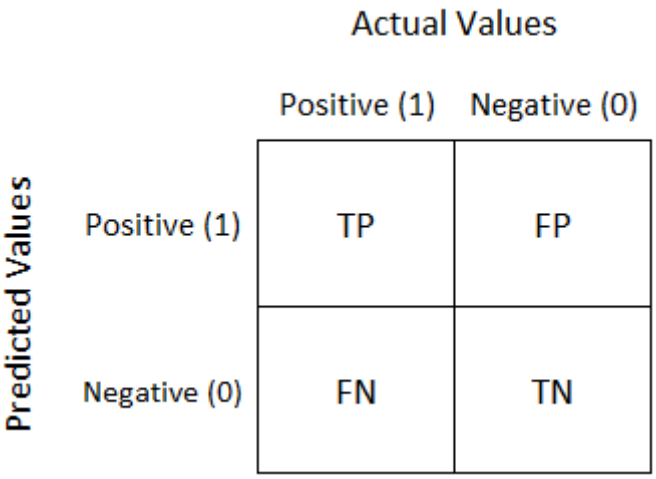

Figure 19. Confusion matrix

Confusion matrix can either be true positive (TP), false positive (FP), false negative (FN), and true negative (TN). True positive means model predicted positive and it is true. True negative means model predicted negative and it is true. False positive means model predicted positive and it is false. Finally, false negative means model predicted negative and it is false. For better understanding check the table below for better understanding.

True positive (TP)	Model predicted positive and it is true
True negative (TN)	Model predicted negative and it is true
False positive (FP)	Model predicted positive and it is false
False negative (FN)	Model predicted negative and it is false

Table 7. Confusion matrix classes

In order to calculate the F-measure, two equations must be calculated first, which are recall and precision. Recall is the ratio of total number of correctly classified positive examples over total number of positive examples. High recall value indicates the prediction is correctly categorized. Furthermore, precision is the ratio of total number of correctly classified positive over total number of files classified as negative. The F-measure is the mean of recall and precision. F-measure ranges from 0 to 1, if the number is getting closer to 1 it means that the model has better prediction performance.

$$recall = TPR = \frac{TP}{TP + FN}.$$

$$precision = \frac{TP}{TP + FP}.$$

$$\text{F-measure} = \frac{2 * precision * recall}{precision + recall}$$

As you can observe in the below charts, the proposed convolutional neural network has proven to achieve better accuracy, and F-measure than the proposed recurrent neural network. Furthermore, both of the proposed deep learning model managed to achieve better performance than Cong's model and Li's model. Cong achieved an average of 0.61 F-measure, while Li' achieved an average of 0.59 F-measure. Moreover, the proposed recurrent neural network achieved 0.66, while the proposed convolutional neural network achieved 0.78 F-measure.

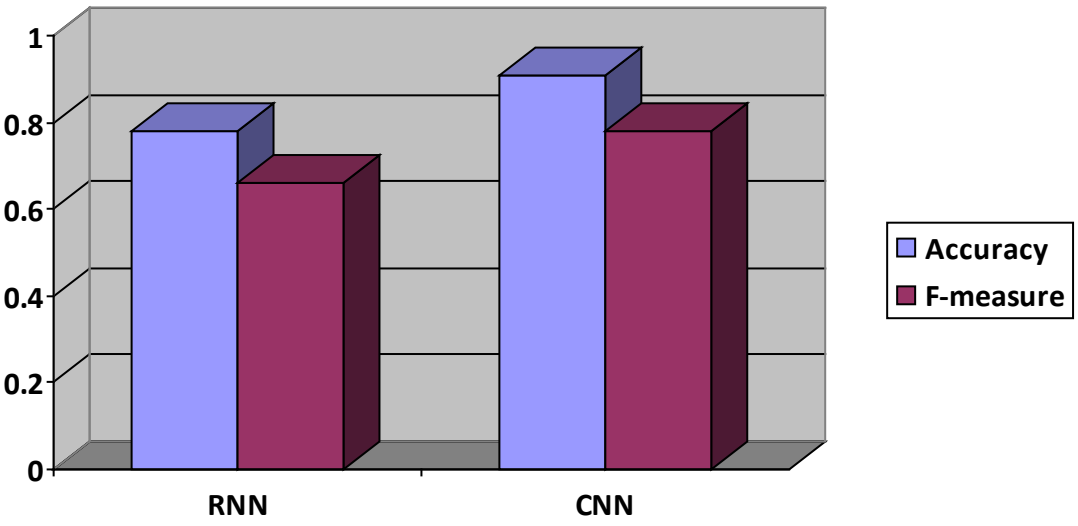

Figure 20. Proposed CNN vs proposed RNN performance

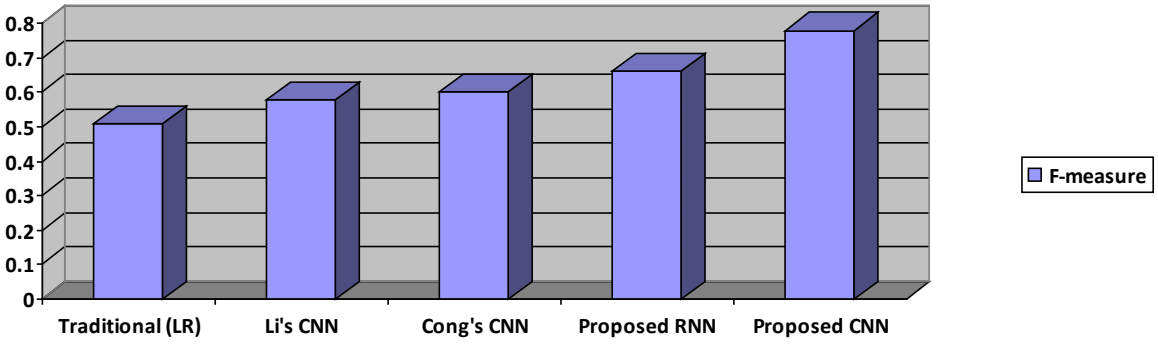

Figure 21. Performance comparison

6 Conclusions and Future Work

6.1 Conclusion

To conclude, software defect prediction has been an inspiration to software engineers as scale and complexity of projects are increasing with modern technology, defects will increase. Defect prediction has been implemented using different well-known machine learning algorithms that has performed well in software defect predictions, however it was not sufficient to capture the syntax and different level of semantics. In this paper, I proposed an approach to predict defects using mapped tokens of an abstract syntax tree which was extracted from source code to capture the syntax and different level of semantics. Nevertheless, two deep learning models have been proposed, a convolutional neural network with an accuracy of 78% and recurrent neural network with an accuracy of 91%

6.2 Future Work

To make software defect prediction have more generalization, dataset should be extended to be able to make a deep learning models able to generalize more. Moreover, a feature should be added in the dataset that represents the specific location of the defect in the java source code. Finally, I recommend using other programming languages other than python, build deeper models which may have better results. Nevertheless, adding defect predictions as a plugin for the compilers to make the software project more reliable and increase the quality attributes.

References

[1] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu (2017). Software Defect Prediction via Convolutional Neural Network, Available at: https://ieeexplore.ieee.org/document/8009936 (Accessed: November 2019).

[2] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim (2018). A deep tree-based model for software defect prediction, Available at https://arxiv.org/abs/1802.00921 (Accessed: November 2019).

[3] Cong Pan, Minyan Lu, Biao Xu, and Houleng Gao (2019). An Improved CNN Model for Within-Project Software Defect Prediction, Available at: https://www.mdpi.com/2076-3417/9/10/2138 (Accessed: November 2019).

[4] Ritu Kapur, and Balwinder Sodhi (2018). Estimating defectiveness of source code: A predictive model using GitHub content, Available at: https://arxiv.org/abs/1803.07764 (Accessed: November 2019).

[5] T. Rajani Devi (2016). Importance of Testing in Software Development Life Cycle, Available at: https://www.ijser.org/researchpaper/Importance-of-Testing-in-Software-Development-Life-Cycle.pdf (Accessed: November 2019).

[6] Abdullah Alsaeedi, and Mohammad Zubair Khan (January 2019). Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study, Available at: https://www.researchgate.net/publication/333216078_Software_Defect_Prediction_Using_Supervised_Machine_Learning_and_Ensemble_Techniques_A_Comparative_Study (Accessed: November 2019).

[7] Song Wang, Taiyue Liu and Lin Tan (2016). Automatically Learning Semantic Features for Defect Prediction, Available at: https://ece.uwaterloo.ca/~lintan/publications/deeplearn-icse16.pdf (Accessed: November 2019).

[8] Markus Thom, and Gunther Palm (2013). Sparse Activity and Sparse Connectivity in Supervised learning, Available at: http://www.jmlr.org/papers/volume14/thom13a/thom13a.pdf (Accessed: November 2019).

[9] Denny Britz (2015). Recurrent Neural Networks, Available at: https://www.easy-tensorflow.com/tf-tutorials/recurrent-neural-networks (Accessed: November 2019).

[10] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov (2014). A simple way to prevent neural network from overfitting, Available at: https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf (Accessed: November 2019).

[11] Haider Khalaf (2014). Methods to avoid over-fitting and under-fitting in supervised machine learning (Comparative study), Available at: https://www.researchgate.net/publication/295198699_METHODS_TO_AVOID_OVER-FITTING_AND_UNDER-FITTING_IN_SUPERVISED_MACHINE_LEARNING_COMPARATIVE_STUDY (Accessed: November 2019).

[12] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, Lu Zhang (2014). Building Program Vector Representations for Deep Learning, Available at: https://arxiv.org/abs/1409.3358 (Accessed: November 2019).

[13] Sarang Narkhede (2018). Understanding confusion matrix, Available at: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62 (Accessed: November 2019)

[14] GeeksforGeeks (2020). Confusion matrix in Machine Learning, Available at: https://www.geeksforgeeks.org/confusion-matrix-machine-learning/#:~:text=A%20confusion%20matrix%20is%20a,the%20performance%20of%20an%20algorithm. (Accessed: December 2019)

[15] Hoa Dam, Trang Pham, Shien Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim (2018). A deep tree-based model for software defect prediction, Available at: https://arxiv.org/abs/1802.00921 (Accessed: December 2019)

[16] Xiang Zhang, Junbo Zhao, Yann LeCun (2015). Character-level convolutional networks for text classifications, Available at: https://papers.nips.cc/paper/5782-character-level-convolutional-networks-for-text-classification.pdf (Accessed: December 2019)