Use of Deep Learning Model with Attention Mechanism for Software Fault Prediction

Ting-Yan Yu
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
eva52525@gmail.com

Chin-Yu Huang
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
cyhuang@cs.nthu.edu.tw

Neil C. Fang
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan
chihchiang0113@seed.net.tw

***Abstract*—Software defect prediction is a skill in software engineering that can increase program reliability. In the past, most defect prediction studies have been based on size and complexity metrics. In recent years, machine learning based predictive studies have been conducted. To build an accurate prediction model, choosing effective features remains critical. In this paper, we constructed a deep learning model called Defect Prediction via Self-Attention mechanism (DPSAM) to extract semantic features and predict defects automatically. We transferred programs into abstract syntax trees (ASTs) and encoded them into token vectors. With input features, we trained a self-attention mechanism to extract semantic features of programs and predict defects. We evaluated performance on 7 open source projects. In Within-Project Defect Prediction (WPDP), DPSAM achieved 16.8% and 14.4% performance improvement compared to state-of-the-art deep belief network (DBN)-based method and defect prediction via convolutional neural network (DP-CNN)-based method in F1 score, respectively. Besides, in Cross-Project Defect Prediction (CPDP), DPSAM achieve 23% and 60% performance improvement in F1 score compared to DBN-based method and DP-CNN-based method.**

***Keywords*: Software engineering, Defect prediction, Deep learning, Convolutional Neural Network, Attention mechanism, Self-Attention mechanism.**

I. INTRODUCTION

In recent years, advances in science and technology has led to software development growing more complex and projects increasing in scale. In order to meet and ensure quality standards and complete schedules on time, an increasing number of methodologies have been developed for software engineering technology. Currently, a number of methods to enhance software quality exist. During the software development process, software testing is generally used to verify whether or not the developed software satisfies user's or project's requirements. Software failure data can be collected and recorded during testing and operational phases, and they are assumed to provide additional information about the failure process [1]. From the test results, project managers and engineers can then objectively make an assessment about the quality or the acceptability of the software.

However, decreased project development time and a wide variety of program languages have increased the difficulty of testing. Therefore, assuring quality and reliability are not only necessary but also increase efficiency. Software defect prediction is a skill in software engineering which can add reliability to programs [2][3][4][5][6][7][8]. Practically, based on the results of early prediction of fault distribution, project managers are able to make required changes to the development approach; that is, managers can choose to revise schedules, reallocate testing resources. Additionally, developers may have to reevaluate the criteria used to determine which program modules or components should receive corrective actions and institute any needed changes.

Software defect prediction identifies the modules that are defect prone and require extensive testing. For engineering programs, it can condense time, reduce overheads and wherewithal, and provide assurance quality of products. Therefore, a variety of methods have been developed. Most defect prediction studies are based on size and complexity metrics in the past. Many studies based on data fitting models or linear models which use different metrics to derive estimations are currently available. In recent years, with the development of machine learning, more and more machine learning based predictive studies that use regression models or classification models to predict software defects.

Most machine learning based processes use features extracted from labeled historical defect data, including code features and process features. By using weighted algorithms and extracted features, the machine learning based methods produce features and put generated features into the machine learning classifier model to enhance performance. Today, deep learning skill has grown considerably in popularity and is applied widely in many research areas [9]. Already, many methods currently use deep learning to conduct feature generation [10][11][12][13]. Programs include not only structures but also syntaxes and semantics, which are hidden deeply in the source code [14]. Deep learning models can capture complicated non-linear features [15]. However, features generated from machine learning based processes are unable to obtain semantic features from programs. Deep learning models are used to perform feature generation, which in turn enhances reliability. Abstract syntax trees (ASTs) have well defined syntaxes from programs [16][17][18]. Therefore, deep learning models can be used with input token vectors extracted from the ASTs of programs to learn semantic features.

There are two kinds of defect predictions: WPDP and CPDP [19][20]. Traditional defect prediction methods gain effective performance in WPDP. However, when training dataset and testing dataset are in different projects, namely CPDP, the performance is unsatisfactory. Using the features generated from deep learning methods can also overcome this problem. Currently, Attention mechanism is very popular and widely used in many fields, such as machine translation, speech recognition, image caption, and so on [21][22][23][24]. Compared with other deep learning models, attention mechanism can parallelize training in efficiency, reduce training time, and achieve optimal performance with minimum training cost. Consequently, we used attention mechanism to generate more reliable features and proposed an approach called DPSAM to perform defect prediction.

In addition, to produce the proper inputs for deep learning model, we had to process ASTs from programs. Therefore, we used word embedding to exploit the program context. Word embedding maps each AST token into a numerical vector, which is trained regarding the context of each token. After finishing pre-process to target programs, the generated features are put into the machine learning classifier model, such as Logistic Regression or Naive Bayes, to predict programs as buggy or clean. However, connecting the classifier and tuning classifier model increases the time consumed in defect prediction process. Therefore, our proposed approach generates features from programs and predicts defects automatically without connecting the machine learning classifier. In addition, this process is able to assess whether the connecting classifier is worth it or not and validate the trade-off between time consumption and performance.

The contributions of our paper are as follows:

- We propose a deep learning based model, DPSAM, to generate features from programs in ASTs and predict buggy or clean from programs automatically.
- We leverage the semantic features learned automatically by Self-Attention mechanism to improve both WPDP and CPDP.
- Our evaluation results on 7 open source Java projects show that using Self-Attention mechanism to generate features improves both WPDP and CPDP. In WPDP, the performance of DPSAM performs as well as the state of the art deep belief network (DBN)-based and the defect prediction via convolutional neural network (DP-CNN) method in F1 score and accuracy. Furthermore, the results in CPDP demonstrate that the automatically learned features by DPSAM outperforms the state-of-the-art DP-CNN method.

The rest of this paper is organized as follows. In Section II, we provide a brief background on software defect prediction, including machine learning models, deep learning models, and Attention mechanism. Then, we present our proposed approach in Section III. In Section IV, we show the result of our experiment and some observations and discussions. Finally, some conclusions and future works are described in Section V.

II. RELATED WORKS

A. *Overview of Software Defect Prediction*

Software defect prediction, which is a skill in software engineering, can discover the regions of buggy code. In addition, it helps developers allocate their testing efforts by first checking potentially buggy code and saving testing time. Due to its importance, software defect prediction has been the focus of researchers for a long time. A number of software defect prediction methods have been published in the past. For example, many test case prioritization techniques aimed to schedule test cases in a manner that increased the rate of fault detection for regression testing. They usually prioritized test cases according to information acquired by analyzing the source code. Huang et al. [25] proposed a Modified Cost-Cognizant Test Case Prioritization (MCCTCP) method based on the use of historical records. They gathered the historical records from the latest regression testing and then proposed a genetic algorithm to determine the most effective order. Their experiments showed that the MCCTCP method could effectively improve the effectiveness of cost-cognizant test case prioritization without analyzing the source code, even when test case costs and fault severities were uniform.

Additionally, Huang et al. [26] also proposed a bounded generalized Pareto distribution (BGPD) model to investigate the fault distributions of open source software. Their proposed BGPD model could eliminate certain issues that occurred in the classical Pareto distribution model and exhibited impressive performance on modeling the distribution of software faults. Luan and Huang [27] once proposed a single change-point 2-parameter generalized PD (SCP-2GPD) model with a very flexible structure and which could model a wide spectrum of software development environments. Their experimental results showed that the Pareto principle could be applied to describe the fault distribution of OSS, and their proposed SCP-2GPD model can be used to depict various OSS fault distributions.

However, most defect prediction studies before the development of machine learning were based on size and complexity metrics. A number of studies based on data fitting models or linear models used different metrics to arrive at estimations. However, an increasing number of machine learning based predictive studies using regression models or classification models to predict have emerged. In the past, most defect prediction methods used traditional hand-crafted features to make estimations. Traditional hand-crafted features are static code features or process features, including Halstead features-based on the number of operators and operands, McCabe features-based on dependencies, CK features for object-oriented programs, and so on [28][29][30]. Code metrics include LCOM, CBO, and other similar metrics. Process features contain the number of revisions, authors, past fixes, and so on. Halstead proposed a number of size metrics, which have been interpreted as complexity metrics, and used these as predictors of program defects [31]. Compton and Withrow of UNISYS derived many polynomial equations to optimize module sizes and discovered that small-size software components often had an extremely high fault density [32].

In machine learning, defect prediction techniques use features to train classifiers to predict defect. And there are various code areas, including method, file, and change. Our proposed approach is file-level defect prediction. File level means that each of the training instances or testing instances is a source code file. Machine learning based defect prediction process consists mainly of three parts. First, data needs to be split as training data and testing data. Then, the machine learning classifier needs to be trained with generated features from training programs for defect prediction. Finally, the target program needs to be entered into the pre-trained classifier to predict whether the program is buggy or clean. The entire process of defect prediction is depicted in Fig.1. Hassan et al. [33] used the entropy of features from code change processes instead of code to predict defects and demonstrate that predictors based on change complexity models were better predictors of future faults in large software systems. Furthermore, Lee et al. [34] proposed 56 novel micro interaction metrics that leverage developers' interaction information stored in the Mylyn data to perform defect prediction. Their experimental results revealed that MIMs were able to improve defect classification and regression accuracy.

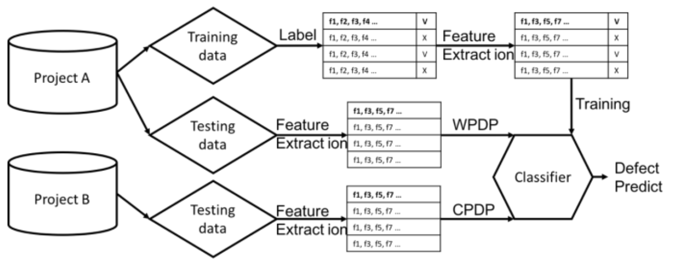

Fig. 1. Process flow of the defect prediction

There are two kinds of defect prediction: WPDP and CPDP. In WPDP, training data and testing data are in the same project. Conversely, in CPDP, training data and testing data are in the different project. Because the features of source projects and target projects often have different distributions, it remains challenging to attain good performance in CPDP [19][34]. Data are derived from the same project in WPDP. Wang et al. [35] proposed defect prediction models based on the C4.5 model to reduce the size of the decision tree model and increase performance by using two modules from Eclipse. Khoshgoftaar and Seliya [36] examined the performance of machine learning algorithms in regression tree types, including CART-LS, S-PLUS, and CART-LAD on defect prediction by using telecommunications software system data which was programmed in PROTEL. Data were derived from the different project in CPDP. TCA made feature distributions in the source and target projects similar. And Nam et al. [37] proposed TCA+ optimized TCA's normalization process to improve the performance of CPDP by using eight open-source projects, including projects of ReLink and projects of AEEEM, which are traditional hand-crafted features. Their experimental result showed that TCA+ improved cross-project prediction performance significantly. Turhan et al. [38] applied analogy-based learning (i.e, nearest neighbor filtering) to cross-company data and within-company data to tune models for defect prediction. They improved performance in CPDP by using 40 kinds of static code features.

B. *Deep Learning in Defect Prediction*

However, programs not only contain structures but also syntaxes and semantics, which are hidden deeply in the source code [14]. The deep learning model can capture complicated nonlinear features. Therefore, many current methods use the deep learning model to generate features. Yang et al. [39] used deep belief network algorithms to generate features from initial change features and connect logistic regression classifiers for just-in-time defect prediction. Their method was able to discover more bugs and achieve higher F1 scores. Wang et al. [12] proposed a feature extraction model which learned the semantic representation of programs automatically from source code to optimize defect prediction performance. They used the deep belief network (DBN) model to learn semantic features from token vectors extracted from the ASTs of programs, and this method bridged the gap between programs' semantics and defect prediction features. Li et al. [10] adopted and improved Wang et al.'s [12] method. They also extract token vectors from the ASTs of programs, then used CNN to extract semantic features for WDPD. Dam et al. [11] also followed and improved Wang et al.'s [12] method by using tree-structured Long Short Term Memory (LSTM) network which directly matches with the ASTs representation of source code for defect prediction.

With the development of deep learning, CNN and Recurrent Neural Network (RNN) have become the representative models for all deep learning courses and books [40]. CNN is very powerful in image recognition. Many models for pattern recognition are based on the CNN architecture [41][42][43]. CNN consists of one or more convolutional layers, a fully connected layer at the top, associated weights, and a pooling layer. In this structure, there are two key characteristics in CNN: local connections and shared weights. These characteristics can benefit defect prediction by capturing the local structural information of programs. Local connections generate local correlation of the inputs. Shared weights help defect prediction to detect features wherever the detect is located in the input and reduce the number of free parameters to increase learning efficiency. In addition, max-pooling reduces the dimensionality of representations and adds robustness to defect prediction.

RNN is used in many topics such as machine translation, sentiment analysis, image caption, and it is especially widely used in the field of NLP research [44][45][46]. RNN uses internal memory to process input sequences. This structure processes text and remembers important words in the sentence. Because RNN is able to understand the semantics of the input data, it can perform defect prediction to capture the semantics of the programs. Xu et al. [47] used Latent Dirichlet Allocation (LDA) in NLP for topic extraction of contextual information. However, RNN is unable to handle the exploding gradient problem, and this makes it difficult for RNN to capture long term dependency. Different kinds of LSTMs are combined to solve this problem [48][49]. LSTM can do defect prediction because it is able to capture the long-term dependencies which exist between code elements.

Attention is a mechanism which improves the effect of the RNN. Currently, attention mechanism is very popular and widely used in many fields such as machine translation, speech recognition, image caption, and so on [50][51][52]. Because attention mechanism assigns different weights to different parts of the input data, it is able to determine which part of the input requires more attention and extract features from key parts to attain information. Attention mechanism has a number of advantages, including helping models assign different weights to each part of the input data, helping models extract more critical information, and making model judgments more accurate. In addition, attention mechanism saves computing and storage overhead. These advantages explain its extensive use, and for this reason, we used attention mechanism for defect prediction in our experiments.

Cho et al. [53] proposed attention mechanism to solve the problem of traditional encoder-decoder model lacking discrimination on the input sequence. The model structure they proposed is depicted in Fig.2. Attention mechanism is able to overcome these problems because it assigns different weights to different parts of the input data.

$$p(Y_i|Y_1, \dots, Y_{i-1}, X) = g(Y_{i-1}, s_i, c_i) \quad (1)$$

$$s_i = f(s_{i-1}, Y_{i-1}, c_i) \quad (2)$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \quad (3)$$

$$\alpha_{ij} = softmax(e_i) = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})} \quad (4)$$

$$e_{ij} = a(s_{i-1}, h_j) \quad (5)$$

In Attention mechanism, the conditional probability is defined as Eq. (1). s_i is the hidden state of the RNN in the decoder at time *i* in Eq. (2), and c_i represents the context vector at time *i*. In the traditional Encoder-Decoder structure, the encoder encodes an input sequence *X* into a fixed-length context vector *c*. *c* is used as the initial vector to initialize the decoder model and predict output sequence Y_1. In addition, the decoder model use context vector *c* and Y_{t-1} decoding to get the Y_t at time *t*, and values in Eq. (3) are weighted. In Eq. (4), *i* indicates the *i* -th word of the encoder, and h_j indicates the hidden vector of the *j* -th word of the encoder; namely, the hidden state of the RNN in the encoder at time *j*. In addition, α_{ij} indicates the weight between the *j* -th word of the encoder and the *i* -th word of the decoder. Furthermore, α_{ij} is a softmax model; its output is the sum of the probability whose value is 1; e_{ij} ndicates an alignment model, and this model is used to measure the influence of the position of the *j* -th word of the encoder on the position of the *i* -th word of the decoder in Eq. (5). There are many methods to calculate e_{ij} and different calculation methods represent different Attention mechanism such as soft Attention, hard Attention, global Attention, local Attention and so on. The simplest and most common alignment model is the dot product.

$$a(s_{i-1}, h_j) = \begin{cases} \overrightarrow{s_{i-1}}^{\mathsf{T}} \vec{h} & dot \\ \overrightarrow{s_{i-1}}^{\mathsf{T}} W \vec{h} & weight \end{cases} \quad (6)$$

Eq. (6) contains different kinds of e_{ij} which indicates the degree of alignment between the source and the target word. Common alignment calculations are dot product and weight, which are the most general. After completing the above calculation, we can get the alignment vector $a_{i,j}$, namely, $score(h_t, \overline{h_s})$, which is the weight of the context vector. Then, context vector c_i can be obtained by weighted averaging, and we are able to get results of *Y*.

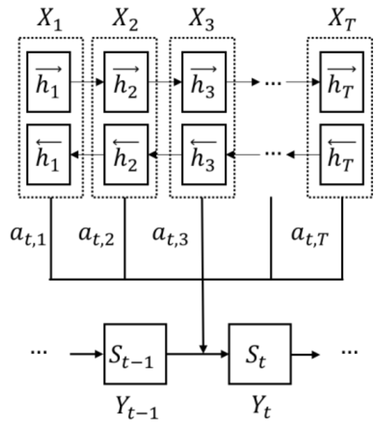

Fig. 2.Attention Mechanism.

III. DEFECT PREDICTION VIA SELF-ATTENTION MECHANISM

In this Section, we elaborate our proposed DPSAM approach that can generate features from programs and predict defects. In order to build and evaluate our model, we mapped target programs into ASTs and split the dataset into training data and testing data. To uniform preprocessing steps with state of the art methods for comparison, we also mapped token vectors into integer vectors. We used integer vectors to train DPSAM, attain effective features, and predict buggy automatically from the training dataset. Our proposed approach contained four major steps as illustrated in Fig. 3:

1. Input source code file
2. Parsing source code into AST tokens
3. Mapping tokens vector to integer vector
4. DPSAM generates semantic features and predict defects automatically

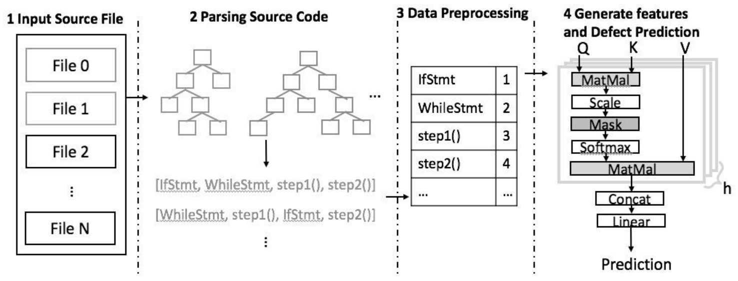

Fig. 3. Overview of our proposed approach.

A. *Parsing Source Code*

There are many types of granularities for the symbol for software programs such as character-level, token-level, nodes on ASTs, and so on. [17] proves nodes on ASTs are good granularity. Therefore, in order to attain effective input features, we selected the proper granularity from ASTs. We used open source tools to transfer source code into ASTs to build program representation[54]. Referring to other works [10][12], we selected three main types of nodes as depicted in TABLE I:

- Method invocations and class instance creations: we record their method names or class names.
- Declaration nodes: we record their node types.
- Control-flow nodes: we also record their node types.

TABLE I. THE SELECTED AST NODES

Method invocations and class instance creations	
ClassOrInterfaceDeclaration	MethodDeclaration
Declaration nodes	
AnnotationDeclaration	AnnotationMemberDeclaration
ConstructorDeclaration	EnumDeclaration
FieldDeclaration	ImportDeclaration
InitializerDeclaration	
Control-flow nodes	
AssertStmt	BlockStmt
BreakStmt	CatchClause
ContinueStmt	DoStmt
ENAplicitConstructorInvocationStmt	ENApressionStmt
ForeachStmt	ForStmt
IfStmt	LabeledStmt
ReturnStmt	SwitchEntryStmt
SwitchStmt	SynchronizedStmt
ThrowStmt	TryStmt
WhileStmt	

B. *Data Preprocessing*

1) Encoding Tokens

Many machine learning or deep learning models require input data in the form of integer vectors. Many methods exist to represent text, which is referred to as word representation. Word representation can convert the text into a computer readable output. Bag-of-Words(BoW) , Vector Space Model, and TF-IDF are common word representation methods. However, these methods have difficulty in presenting semantic in programs. For example, high dimension and high sparsity are two weaknesses of BoW [55]. These weaknesses take a great quantity of space and are not able to present the syntactic of programs. Therefore, we used token mapping to represent words to obtain semantic from programs effectively.

We built a map between integers and tokens and encoded token vectors to integer vectors. Different class names, method names, and node types were mapped to different numbers starting from 1. We could get integer vectors converted from token vectors after this process. In addition, the order of the tokens remained unchanged, and the structure information of the program was retained. We filtered out AST nodes that were not frequently used since these AST nodes may be designed for specific files and were difficult to generalize to other files. This technique is a common practice in the natural language processing (NLP) research field [56].

2) Handling Imbalance

Because the number of clean instances exceeded considerably the number of buggy instances in software defect data, the imbalanced data reduced the performance of our model [57]. Two methods can solve this problem in training data. The first method called downsampling is reducing the number of instances in the majority class, which could lead to information loss. The second method called oversampling includes increasing the number of instances in the minority class by duplicating buggy instances; we used this second method. Consequently, we were able to obtain a balanced dataset.

C. *Training Self-Attention Mechanism and Predict Defect*

After completing the data preprocessing, we could use models to extract features. The model we used in our proposed approach was self-attention mechanism, which is widely used in NLP research, especially in machine translation. Self-Attention mechanism differs from traditional Attention mechanism because traditional Attention mechanism calculates the base of hidden states of the source and the target to obtain the dependencies on each word of the source and each word of the target. This method ignores the dependencies on words of the source or the target. However, Self-Attention works on the source and the target, and also gets the dependencies on words of the source or the target, respectively. It can obtain the dependencies not only on the words of the source and the target, but also on words of the source or the target. Therefore, it is able to capture syntactic features or semantic features between words in sentences. Furthermore, it is better at capturing the internal correlation of data or features and is able to extract important features of data quickly. Therefore, we used Self-Attention mechanism to extract features for defect prediction.

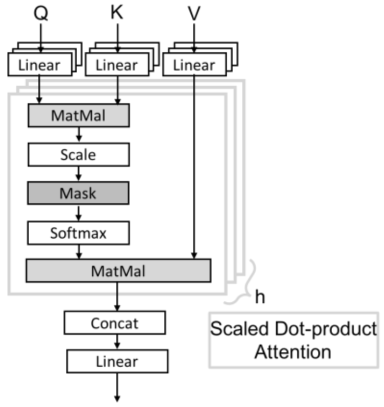

Fig. 4. Multi-head Attention architecture.

Self-Attention mechanism was implemented using scaled dot-product Attention unit. Fig. 4 presents a multi-head Attention architecture, and the labeled parts are Scaled dot-product Attention units. First, the inputs are linearly transformed into Q, K, and V, respectively. Q indicates query, K indicates key, and V indicates value. Q, K, and V are all converted from the inputs; however, the weights of the linearly transformed matrix are different.

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad (7)$$

$$MultiHead(Q,K,V) = Concat(head_1, \ldots, head_h)W^O \qquad (8)$$

The expression for scaled dot-product Attention is in Eq. (7) and it is the weighted expression equation. Scaling is used to prevent excessive input from making training unstable, and softmax is used to normalize its results into probability distributions. Mask is used to mask future information to ensure time alignment. Its output is a weighted result of *V*. d_k indicates the dimension of a *Q* and *K* vector. In Fig. 4, *Q*, *K*, and *V* make a linear transformation, and then enter the scaled dot-product Attention. In Eq. (8), multi-head Attention makes different projections for *Q*, *K*, and *V* for *h* times. $head_i$ is equal to $Attention(QW_i^Q, KW_i^K, VW_i^V)$. W_i^Q belongs to $R^{d_{model}*d_k}$; W_i^K belongs to $R^{d_{model}*d_k}$; W_i^V belongs to $R^{d_{model}*d_v}$. The parameter *W* is different every time when *Q*, *K*, and *V* make linear transformations. Then we concat the results of scaled dot-product Attention for *h* times and conduct linear transform again to obtain the result of multi-head Attention. Because the multi-head Attention calculates for *h* times not only once, the model can learn relevant information in different representation subspaces.

Attention mechanism completed feature extraction and generated new features. Then, we connected a pooling layer to reduce the dimensionality of intermediate representations and provide additional robustness. Considering the whole defect prediction process time, we did not connect classifiers such as logistic regression to get results when obtaining new features. We connected a sigmoid layer to convert the score for each word into a probability value. In the structure, our model was able to extract the semantic of programs and predict defects automatically. In addition, in order to have good feature extraction performance, model tuning is necessary. The training hyper-parameters for Self-Attention mechanism are depicted in TABLE II.

TABLE II. VALUES OF HYPER-PARAMETERS

Hyper-parameter	**Value**
batch size	36
optimization	Adam
loss function	Binary crossentropy
dropout	0.5
embedding dimension	20
multi-head number	3

IV. EXPERIMENT AND DISCUSSION

A. *Dataset*

We used open source data from the PROMISE dataset and chose 7 Java projects as our source code [58]. All of the data were labeled as clean or buggy. Furthermore, most code were accessible from Github. As depicted in TABLE III, the buggy rates of the projects had a minimum value of 22.4% and a maximum value of 58.2%. Furthermore, TABLE IV reveals that the file numbers of projects ranged from 108 to 964.

TABLE III. SUMMARY of PROMISE DATASET

Dataset	Description	Avg File	Avg Buggy (%)
camel	Enterprise integration framework	814	22.4
log4j	Logging library for Java	150	58.2
lucene	Text search engine library	269	54.2
poi	Java library to access Microsoft format files	344	51.3
synapse	Data transport adapters	211	25.5
Xalan	A library for transforming NAML files	830	54.3
Xerces	NAML parser	493	38.9

TABLE IV. DATASET VERSIONS

Dataset	Version				Dataset	Version			
camel	1.2	1.4	1.6		synapse	1.0	1.1	1.2	
File number	607	871	964		File number	157	222	256	
Buggy file	216	145	188		Buggy file	16	60	86	
Buggy (%)	35.6	16.6	19.5		Buggy (%)	10.2	27.0	33.6	
log4j	1.0	1.1	1.2		Xalan	2.4	2.5	2.6	2.7
File number	134	108	204		File number	723	803	885	909
Buggy file	34	37	189		Buggy file	110	387	411	898
Buggy (%)	25.4	34.3	92.6		Buggy (%)	15.2	48.2	46.4	98.8
lucene	2.0	2.2	2.4		Xerces	1.2	1.3	1.4	
File number	194	274	339		File number	440	453	588	
Buggy file	91	144	203		Buggy file	71	69	437	
Buggy (%)	46.9	52.6	59.9		Buggy (%)	16.1	15.2	74.3	
poi	1.5	2.0	2.5	3.0					
File number	237	314	385	442					
Buggy file	141	37	248	281					
Buggy (%)	59.5	11.8	64.4	63.6					

B. *Evaluation Metrics*

In this paper, we plan to use four metrics, including accuracy, precision, recall, and F1 score, which are widely used in research related to machine learning and deep learning [10][12][59][60]. Accuracy is the common baseline to verify performance for the classification problem. However, software defect data has a data skew problem such as data imbalance, where the number of clean instances outnumbers buggy instances considerably. Therefore, we used the F1 score which is widely adopted to evaluate model performance as metrics.

TABLE V is a confusion matrix table with 4 different combinations of predicted values and actual values, including true positive, true negative, false positive, and false negative [61]. True positive is the number of predicted defective files which are truly buggy. In contrast, false positive is the number of predicted defective files which are clean. True negative is the number of predicted non-defective files which are clean in fact, whereas false negative is the number of predicted non-defective files which are actually buggy.

Precision reflects the ability of classification models to distinguish negative samples. The higher the precision, the stronger the distinguishing ability for negative samples. Furthermore, recall reflects the ability for distinguishing positive samples. F1 score combines both of them. Therefore, F1 reflects the robustness of classification models. The higher the value of all the metrics, the better.

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive} = \frac{N_{b\to b}}{N_{b\to b} + N_{c\to b}} \quad (9)$$

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative} = \frac{N_{b\to b}}{N_{b\to b} + N_{b\to c}} \quad (10)$$

$$\text{F1 score} = \frac{2 * Precision * Recall}{Precision + Recall} \quad (11)$$

TABLE V. CONFUSION MATRIX

	Actual Values: Positive	Actual Values: Negative
Predicted Values: Positive	True Positive	False Positive
Predicted Values: Negative	False Negative	True Negative

C. *Baseline Methods*

We compared our proposed DPSAM approach with the following baseline methods in defect prediction displayed in TABLE VI.

- DBN: the state-of-the-art method which employs DBN on source code to extract semantic features and use DBN-learned features into classifier models such as Logistic Regression or Naive Bayes for prediction [12].
- DP-CNN: the state-of-the-art method which employs CNN on source code to extract semantic features and use CNN-learned features into Logistic Regression for prediction [10].
- DP-CNN+: an extended version of DP–CNN proposed by us to complete CPDP performance not in DP-CNN, which use CNN-learned features into Logistic Regression or Naive Bayes for prediction.

Because there is no result of CPDP in DP-CNN+, we implemented DP-CNN to get the baseline method and complete the experiment data. When we implemented DP-CNN+, we used the same network architecture and parameter. We followed the same procedure to preprocess source code, including parsing program, encoding tokens, and handling imbalance. In DP-CNN+, we used logistic regression and naive Bayes for defect prediction. In addition, to save classifier training time, we did not connect the classifier, but used the pooling layer and sigmoid layer to calculate the probability of buggy automatically in DPCNN+ and DPSAM.

TABLE VI. The Description of Four Different Models

Models	Classifier	Method name
DBN	Logistic Regression	DBN-LR
DBN	Naive Bayes	DBN-NB
DP-CNN	Logistic Regression	DPCNN-LR
DP-CNN+	Logistic Regression	DPCNN+LR
DP-CNN+	Naive Bayes	DPCNN+NB
DP-CNN+	NA	DPCNN+
DPSAM	NA	DPSAM

D. *Experimental Result*

In our experiments, we compared our proposed approach with deep learning models, including DBN and CNN. The experimental results revealed the performance for software defect prediction in accuracy, precision, recall, and F1 score. There are two kinds of defect prediction: WPDP and CPDP. In this section, we demonstrate defect prediction results in both WPDP and CPDP.

1) *Within-Project Defect Prediction (WPDP)*

The performance of DPSAM are displayed from TABLE VII to TABLE X. The datasets of DBN may contain experiment results because it implemented cross version defect prediction. Therefore, we selected the results demonstrating the best performance, which is indicated by a star (*) in TABLE VII. However, the average performance is the value of the original paper. DP-CNN is missing results of three datasets, including ant, ivy, and log4j. In addition, only one metric was used to record the performance of the results; namely, F1-score in DBN and DP-CNN. We marked the missing data as NA.

First, discussing with F1 score, the average performance of our proposed approaches was 70%. As depicted in TABLE VII, DPSAM achieved 16.8% performance improvement compared to the average performance of DBN, achieved 14.4% performance improvement compared to DP-CNN, and achieved 56.5% performance improvement compared to the average performance of DP-CNN+. Referring to the original paper, parameters of DP-CNN were variable when constructing the feature extraction model. However, we implemented DP-CNN+ with consist parameters. This may explain performance not being as good as that of DP-CNN.

As for the other metrics, such as accuracy, precision, and recall, our proposed approach achieved performance improvement 1.38 times higher than the average accuracy of DP-CNN+. However, the precision of our proposed approach was not as good as the average precision of DP-CNN+. Recall of DPSAM was 2.08 times higher than the average recall of DP-CNN+. Precision reflects the ability of classification models to distinguish negative samples. The higher the precision, the stronger the distinguishing ability for negative samples. Recall reflects the distinguishing ability in positive samples. In defect prediction, positive means the data is buggy and vice versa. Finding out files that contain defects is more important than predicting files that are clean. Therefore, the performance of DPSAM was better than that of DP-CNN+. The experimental results demonstrated that our proposed approach was useful for WPDP. Our proposed approach outperforms the four metrics to varying degrees.

TABLE VII. F1 scores in WPDP

F1 score	DBN -LR*	DBN-NB*	DPCNN -LR	DPCNN +LR	DPCNN +NB	DPCNN +	DPSAM
camel	59.8	48.1	50.8	16.7	32.9	29.1	38.9
log4j	68.2	72.5	NA	40.2	43.5	49.0	97.9
lucene	63.0	73.8	76.1	67.4	50.3	75.6	75.6
poi	78.3	77.7	78.4	64.8	52.2	56.3	78.2
synapse	54.1	57.9	55.6	45.0	55.3	49.4	59.0
Xalan	56.5	45.2	69.6	50.8	54.1	45.7	99.9
Xerces	47.5	38.0	37.4	36.8	23.9	1.9	41.5
Average	61.1	59.0	61.3	46.0	44.6	43.9	70.1

TABLE VIII. Accuracy in WPDP

Accuracy	DBN -LR	DBN -NB	DPCNN-LR	DPCNN+ LR	DPCNN+ NB	DPCNN+	DPSAM
camel	NA	NA	NA	77.5	75.0	78.0	66.7
log4j				27.1	29.7	33.9	95.8
lucene				62.9	55.7	60.8	60.8
poi				61.9	56.2	58.9	64.2
synapse				72.0	73.2	33.1	65.0
Xalan				34.1	37.1	29.7	99.9
Xerces				48.5	43.3	36.2	48.8
Average				54.9	52.9	47.2	71.6

TABLE IX. Precision in WPDP

Precision	DBN -LR	DBN-NB	DPCNN-LR	DPCNN+ LR	DPCNN+ NB	DPCNN+	DPSAM
camel	NA	NA	NA	32.3	35.6	41.2	30.7
log4j				94.0	94.5	93.8	95.8
lucene				72.3	78.9	60.8	60.8
poi				79.4	86.8	88.5	64.2
synapse				67.4	63.6	33.2	48.9
Xalan				100.0	100.0	100.0	99.9
Xerces				87.5	87.9	100.0	78.7
Average				76.1	78.2	73.9	68.4

TABLE X. Recall in WPDP

Recall	DBN -LR	DBN-NB	DPCNN-LR	DPCNN+ LR	DPCNN+ NB	DPCNN+	DPSAM
camel	NA	NA	NA	11.2	30.5	22.5	52.9
log4j				25.5	28.3	33.2	100.0
lucene				63.1	36.9	100.0	100.0
poi				54.8	37.4	41.3	100.0
synapse				33.7	48.8	96.5	74.4
Xalan				34.0	37.1	29.6	100.0
Xerces				23.3	13.8	1.0	28.1
Average				35.1	33.3	46.3	79.3

2) *Cross-Project Defect Prediction(CPDP)*

CPDP means training data and testing data are located in different projects. We used the last version in every dataset as testing data and older versions as training data. For example, in the camel dataset, our training data were 1.2, and 1.4, and testing data was 1.6; in CPDP, training data were the same; however, testing data was 1.2 from the log4j dataset, 2.4 from the lucene dataset, and so on. We used 7 Java projects as our source code. Therefore, for the 7 datasets multiplied by 4 metrics, there was 28 tables in total. To save layout and increase legibility, we consolidated the experiment results from TABLE XI to TABLE XV.

DBN did not verify all the datasets in CPDP. For example, when the ant dataset served as training data, only the results that used camel dataset and poi dataset as testing data were available. Therefore, we indicated this with a star (*) in TABLE XI. There was no result of CPDP in DP-CNN [10] and no accuracy, precision and recall result for CPDP in DBN [12]; therefore, we marked the missing data as NA. First, regarding F1 score, average performance of the proposed methods was 71.9%. As depicted in TABLE XI ,we can perceive that, DPSAM achieved 23% performance improvement compared to DBN, and achieved 60% performance improvement compared to average DP-CNN+ performance. In addition, the performance of other methods was inconsistent in different datasets; however, our proposed approach maintained the performance in different datasets. Obviously, DPSAM is a stable model. As for the other metrics, our proposed approach achieved performance improvement 1.29 times higher than the average accuracy of DP-CNN+. Although the precision of our proposed approach is not as good as the average precision of DP-CNN+, recall of DPSAM was 2.14 times higher than the average recall of DP-CNN+. The phenomenon and reason were as mentioned earlier for WPDP. Therefore, it can be concluded that the performance of DPSAM was better than DP-CNN+.

We set a baseline where the model was applicable to a tested dataset if the F1 score was above 70%. For each dataset, TABLE XV reports the number of the other datasets to which the corresponding models can be applied. Our proposed approach improved the general applicability of prediction models and was more applicable than other methods. Each dataset was successfully applicable to at least 3 other datasets and some of them were even applicable to 4 other datasets. Generally, the performance of CPDP was poor because feature distribution differed between the source projects and the target projects. Altogether, our proposed approach was more reliable and more accurate than DBN and DP-CNN models. Furthermore, our proposed approach improved performance in CPDP.

E. *Observation and Discussion*

Defect prediction faces two main problems in the twenty-first century. The first problem relates to building a precise prediction model for new projects or projects having less historical data. Therefore, many CPDP models have been proposed. Our proposed approach overcame the first problem. The second question relates to applying defect prediction models in industry. Fortunately, numerous studies, including case studies and proposed practical applications have been conducted. Rahman et al. [62] demonstrated that defect prediction was able to help prioritize warnings reported by static bug finders. Another application involved using defect prediction results to prioritize or select test cases, such as saving testing cost in regression testing.

In addition, testing in system development life cycle (SDLC) helps developers to ensure functionality and reliability of software systems. However, it accrues considerable software development costs. Therefore, having a good testing strategy to find and fix defects is crucial for any industry. Predicting buggy files, modules, or functions supports managing the limited test resources and avoid releasing software with critical defects. Our proposed approach can predict buggy files in the system. Therefore, accurate prediction of defect-prone files aids developers to direct test efforts, reduce costs, and improve the software testing process by focusing on defect-prone files.

Our proposed approach is able to predict buggy files in the overall system automatically. Therefore, developers or maintenance personnel can understand which files are defect-prone and test these files first in the testing phase. The test phase includes white box testing and black box testing. In white box testing, developers can use the AST tokens parsed by our proposed approach to understand the structure and process of the program. This method supports developers in writing test cases. In black box testing, developers are able to allocate more testing effort to the buggy files and reduce testing effort on the clean files. This approach can aid in reducing time consumption in black box testing. Therefore, our proposed approach can help reduce errors in industrial applications, reduce overall testing time, and avoid outflow of defects to ensure product quality.

TABLE XI. F1 SCORE IN CPDP.

F1 SCORE	DBN-LR/NB*	DPCNN +LR	DPCNN +NB	DPCNN+	DPSAM
camel	NA	42.6	44.1	40.1	78.8
log4j	69.2	49.0	46.0	50.6	71.5
lucene	58.4	65.1	48.4	73.1	73.1
poi	51.4	52.4	48.2	48.5	72.9
synapse	66.1	33.8	36.7	39.7	69.2
xalan	49.0	50.3	50.0	46.3	74.8
xerces	57.2	29.4	28.9	20.7	63.2
Average	58.6	46.1	43.2	45.6	71.9

TABLE XII. ACCURACY IN CPDP.

Accuracy	DBN-LR/NB	DPCNN +LR	DPCNN +NB	DPCNN+	DPSAM
camel	NA	51.3	47.2	45.3	77.3
log4j		56.6	55.3	58.2	65.1
lucene		64.0	52.3	63.0	63.0
poi		52.9	52.5	52.2	66.8
synapse		42.5	45.1	47.4	67.2
xalan		56.0	56.4	55.9	70.0
xerces		43.9	45.2	41.1	58.5
Average		52.5	50.6	51.9	66.8

TABLE XIII. PRECISION IN CPDP.

Precision	DBN-LR/NB	DPCNN +LR	DPCNN +NB	DPCNN+	DPSAM
camel	NA	84.0	86.0	85.7	82.9
log4j		73.1	73.9	72.6	61.0
lucene		72.3	74.9	63.0	63.0
poi		71.6	74.9	73.2	63.0
synapse		79.5	79.8	80.5	76.3
xalan		73.0	73.8	72.3	64.8
xerces		73.6	79.4	72.6	65.1
Average		75.3	77.5	74.3	68.0

TABLE XIV. RECALL IN CPDP.

Recall	DBN-LR/NB	DPCNN +LR	DPCNN +NB	DPCNN+	DPSAM
camel	NA	35.8	31.0	27.2	76.4
log4j		40.3	37.1	42.9	94.7
lucene		64.4	40.0	100.0	63.0
poi		47.7	39.6	40.5	95.5
synapse		21.8	26.0	26.6	67.2
xalan		37.3	36.5	32.0	87.7
xerces		21.7	19.3	15.4	73.5
Average		38.4	32.8	40.7	79.7

TABLE XV. APPLICABLE DATASET IN CPDP

Applicable dataset	DBN-LR/NB	DPCNN +LR	DPCNN +NB	DPCNN+	DPSAM
camel	1	0	0	0	3
log4j	0	0	0	0	4
lucene	0	3	0	4	4
poi	0	0	0	0	4
synapse	0	0	0	0	4
Xalan	0	0	0	0	4
Xerces	0	0	0	0	3
Average	0.1	0.4	0.0	0.6	3.7

V. CONCLUSIONS

Assuring the quality of projects is not only necessary but also increases efficiency when developing software. Generally, developers might need to find the main causes of these detected faults and then eliminate them in order to reduce the occurrence of faults and the risks of software-failure. For instance, a cause and effect diagram (CED), also called the fishbone diagram, is typically designed to sort and determine the potential causes of the observed problems and effects [63]. Practically, in order to identify the causes, engineers and project managers have to group the causes into some major categories, such as the product, process, people, development environment, tool, training, etc. The root causes analysis can be implemented and proposed solutions (i.e., details of fault prevention activities, responsible person, implementation start/end dates, etc.) can also be developed through some brainstorming meetings or direct suggestions or instructions from senior engineers or managers. In addition, fault history classifications or change history classification will usually be created, maintained, and updated [64].

Software defect prediction is a skill in software engineering which can add reliability to programs. In this paper, we propose DPSAM as an approach to predict defects. We use self-attention mechanism to extract feature from programs. Our proposed approach was able to save the semantic of programs and predict defects automatically. We implemented our experiment on 7 open source datasets which are also used in [12] and [10]. In WPDP, DPSAM achieved 16.8% and 14.4% performance improvement compared to state-of-the-art DBN-based method and DP-CNN-based method in F1 score, respectively. We verified our experiment in more data sets and enhanced the reliability of our proposed approach. The features of source projects and target projects often have different distributions, so it is challenging to attain good performance in CPDP. However, in CPDP, DPSAM achieve 23% and 60% performance improvement in F1 score compared to DBN and DP-CNN+ respectively. In addition, compared to state-of-the-art methods, our proposed approach demonstrated the best performance and was not the most time-consuming. Therefore, our proposed approach was more efficient. Altogether, the quality of software would be significantly increased and the risk of project-failure can be greatly reduced if defects are detected as early as possible.

ACKNOWLEDGMENT

The work described in this paper was supported by Ministry of Science and Technology, Taiwan, under Grants MOST 108-2221-E-007-033-MY3 and MOST 110-2221-E-007-035-MY3.

REFERENCES

[1] W.E. Wong, X. Li, and P.A. Laplante, "Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures," *Journal of Systems and Software*, vol. 133, pp. 68-94, 2017.

[2] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," *Proceeding of 2008 ACM/IEEE 30th International Conference on Software Engineering (ICSE'08)*, pp. 181–190, 2008.

[3] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software defect prediction," *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pp. 414–423, 2014.

[4] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," *Proceedings of the 37th international Conference on Software Engineering (ICSE)*,Vol. 2, pp. 99–108, 2015.

[5] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 35th International Conference on Software Engineering*, pp. 382–391,2013.

[6] J. Nam, "Survey on software defect prediction," Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Tech. Rep, 2014.

[7] X. Bai, H. Zhou, and H. Yang, "An HVSM-based GRU Approach to Predict Cross-Version Software Defects," *International Journal of Performability Engineering*, vol. 16, no. 6, pp. 979–990, June 2020.

[8] Y. Li, W.E. Wong, S.Y. Lee, and F. Wotawa, "Using tri-relation networks for effective software fault-proneness prediction," *IEEE Access*, vol. 7, pp. 63066-63080, 2019.

[9] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," Science'06, Vol.313, pp.504–507,2006.

[10] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," *Proceedings of 2017 IEEE International Conference on Software Quality, Reliability and Security*, Prague, Czech Republic, pp. 318–328, 2017.

[11] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, "A deep tree based model for software defect prediction," [Online]. Available: https://arxiv.org/abs/1802.00921. Accessed: Mar. 7, 2018.

[12] S. Wang, T. Liu and L. Tan, "Automatically learning semantic features for defect prediction," *Proceedings of the 38th International Conference on Software Engineering*, Austin, TX, USA, pp.297-308,2016.

[13] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose "Automatic feature learning for vulnerability prediction," [Online]. Available: https://arxiv.org/abs/1708.02368. Accessed: Mar. 7, 2018.

[14] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. "Toward deep learning software repositories," *Proceedings of the 12th Working Conference on Mining Software Repositories*, Florence, Italy, pp. 334–345,2015.

[15] H. K. Dam, T. Tran, T. Pham, "A deep language model for software code," [Online]. Available: https://arxiv.org/abs/1608.02715. Accessed: Mar. 7, 2018.

[16] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *International Conference on Knowledge Science, Engineering and Management*, Springer, pp. 547-553, 2015

[17] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," *Proceedings of the 34th International Conference on Software Engineering*, Zurich, Switzerland, pp. 837–847, 2012.

[18] C. J. Maddison, and D. Tarlow, "Structured generative models

of natural source code," [Online]. Available: https:// arxiv.org/abs/1401.0514. Accessed: Mar. 7, 2018.

[19] J. Nam, W.Fu, S. Kim, T.Menzies, and L.Tan. "Heterogeneous defect prediction," *IEEE Transactions on Software Engineering*, Vol. 44, no.9, pp.874–896, 2018

[20] F. Li, Y. Qu, J. Ji, D. Zhang, and L. Li, "Active learning empirical research on cross-version software defect prediction datasets," *International Journal of Performability Engineering*, vol. 16, no. 4, pp.609-617, April 2020.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, Long Beach, CA, USA, 2017.

[22] M. T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2015

[23] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio, "Show, attend and tell: neural image caption generation with visual attention," [Online]. Available: https://arxiv.org/abs/1502.03044. Accessed: Mar. 7, 2018.

[24] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent model of visual attention," *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[25] Y. C. Huang, K. L. Peng, and C. Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," *Journal of Systems and Software*, Vol. 85, Issue 3, pp. 626–637, March 2012.

[26] C. Y. Huang, C. S. Kuo, and S. P. Luan, "Evaluation of bounded generalized pareto model for the analysis of fault distribution of open source software," *IEEE Trans. on Reliability*, Vol. 63, No. 1, pp. 309-319, March 2014.

[27] S. P. Luan and C. Y. Huang, "An improved pareto distribution for modeling the fault data of open source software," *Software Testing, Verification and Reliability*, Vol. 24, Issue 6, pp. 416–437, Sept. 2014.

[28] H. H. Maurice, "Elements of software science (operating and program-ming systems series)," 1977.

[29] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, Vol: SE-2 ,No. 4, pp. 308–320, 1976.

[30] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, 1994.

[31] M. H. Halstead, "Elements of software science." Elsevier, North-Holland, 1977.

[32] B.T. Compton, and C. Withrow, "Prediction and control of ada software defects," *Journal of Systems and Software*, Vol. 12, No. 3, pp. 199-207, 1990.

[33] A. E. Hassan. "Predicting faults using the complexity of code changes," *Proceedings of IEEE 31st International Conference on Software Engineering*, Vancouver, BC, Canada ,pp. 78–88, 2009

[34] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. "Micro interaction metrics for defect prediction, " *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 311–321, 2011.

[35] J. Wang, B. Shen, and Y. Chen. "Compressed c4. 5 models for software defect prediction," *Proceedings of the 12th International Conference on Quality Software*, Xi'an, China, pp. 13-16, 2012

[36] T.M. Khoshgoftaar and N. Seliya. "Tree-based software quality estimation models for fault prediction," *Proceedings of Eighth IEEE Symposium on Software Metrics* , Ottawa, ON, Canada, pp. 203-214, 2002.

[37] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," *Proceedings of 35th International Conference on Software Engineering (ICSE)*, pp.382–391,2013.

[38] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. "On the relative value of cross-company and within-company data for defect prediction," Empirical Software Engineering, 14(5), pp. 540–578, 2009.

[39] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. "Deep learning for just-in-time defect prediction." *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*, Vancouver, BC, Canada, pp.17–26, 2015.

[40] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[41] A. Krizhevsky, I. Sutskever, and G. Hinton. "Imagenet classification with deep convolutional neural networks". *Communications of the ACM*, Vol. 60, No.6, pp.84–90, 2017.

[42] Xu, L., Ren, J.S., Liu, C., and Jia, J. "Deep convolutional neural network for image deconvolution," *Proceedings of the 27th International Conference on Neural Information Processing Systems*, pp.1790–1798, 2014.

[43] S. Ren, K. He, R. Girshick, and J. Sun. "Faster R-CNN: Towards real-time object detection with region proposal networks." In NIPS, 2015.

[44] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, 2013

[45] K. Cho, B. V. Merrie ̈nboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. "Learning phrase representations using rnn encoder-decoder for statistical machine translation." *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp.1724–1734,2014.

[46] T. Mikolov, M. Karafia ́t, L. Burget, J. Cernocky`, and S. Khudanpur. "Recurrent neural network based language model." In INTERSPEECH, pp. 1045–1048, 2010.

[47] J. Xu, L. Yan, F. Wang, and J. Ai, "A github-based data collection method for software defect prediction," *Proceedings of the 6th International Conference on Dependable Systems and Their Applications (DSA)*,Harbin, China, pp.100–108, 2019.

[48] S. Hochreiter and J. Schmidhuber. "Long short-term memory." Neural computation, 9(8), pp.1735–1780, 1997.

[49] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jo ́zefowicz, and S. Bengio., "Generating sentences from a continuous space." *In Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, pp. 10-21, 2016.

[50] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, Vol. 1, pp. 599–609, 2016.

[51] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, Vol. 1, pp.2073–2083, 2016.

[52] P. Yin, and G. Neubig, "A syntactic neural model for general-purpose code generation," *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, Vol. 1, pp. 440–450, 2017.

[53] K. Cho, B. V. Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *Proceedings of the Empirical Methods in Natural Language Processing (EMNLP)*, pp.1724–1734, 2014.

[54] "JavaParser," [Online]. Available: https://github.com/donnchadh/JavaParser/tree/master/JavaParser. Accessed: Mar. 7, 2018.

[55] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, “Predicting vulnerable software components via text mining.” *IEEE Transactions on Software Engineering*, Vol. 40, No. 10, pp. 993–1006, 2014.

[56] C. D. Manning and H. Schutze. Foundations of statistical natural language processing. MIT press, 1999.

[57] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online defect prediction for imbalanced data,” *Proceedings of the 37th International Conference on Software Engineering*, Vol. 2, pp. 99–108, 2015.

[58] “PROMISE dataset,” [Online]. Available: http://openscience.us/repo/defect/. Accessed: Mar. 7, 2018.

[59] D. M. W. Powers, “Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation,” *International Journal of Machine Learning Technology*, Vol. 2, No. 1, pp. 37–63, Dec. 2011.

[60] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. “Defect prediction from static code features: current results, limitations, new approaches,” *Automated Software Engineering*, 17(4), pp. 375–407, 2010.

[61] J. T. Townsend., “Theoretical analysis of an alphabetic confusion matrix,” *Psychonomic Journals*, Vol. 9, No. 1, pp. 40–50, 1971.

[62] F. Rahman, S. Khatri, E.T. Barr and P. Devanbu,“ Comparing static bug finders and statistical prediction,” *Proceedings of the 2014 International Conference on Software Engineering*, pp. 424–434, 2014.

[63] P. Jalote, *Software Project Management in Practice*, Pearson Education, 2002.

[64] C. Ebert, R. Dumke, M. Bundschuh, and A. Schmietendorf, *Best Practices in Software Measuremen*t, Springer, 2005.