A COMPARATIVE STUDY OF DEEP LEARNING TECHNIQUES IN SOFTWARE FAULT PREDICTION

Ha Thi Minh Phuong, Dang Thi Kim Ngan, Nguyen Thanh Binh*

The University of Danang - Vietnam-Korea University of Information and Communication Technology, Vietnam

*Corresponding author: ntbinh@vku.udn.vn

(Received: May 15, 2024; Revised: June 09, 2024; Accepted: June 11, 2024)

Abstract - Software fault prediction (SFP) is an important approach in software engineering that ensures software quality and reliability. Prediction of software faults helps developers identify faulty components in software systems. Several studies focus on software metrics which are input into machine learning models to predict faulty components. However, such studies may not capture the semantic and structural information of software that is necessary for building fault prediction models with better performance. Therefore, this paper discusses the effectiveness of deep learning models including Deep Belief Networks (DBN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and Long-Short Term Memory (LSTM) that are utilized to construct fault prediction models based on the contextual information. The experiment, which has been conducted on seven Apache datasets, with Precision, Recall, and F1-score are performance metrics. The comparison results show that LSTM and RNN are potential techniques for building highly accurate fault prediction models.

Key words - Software engineering; deep learning; software fault prediction; abstract syntax tree; software faults

1. Introduction

Software testing is an essential task but a costly action in the software development life cycle. Many reports have shown that software products contain a high number of faults occurring during the testing phase and even post-deployment of the software system. According to a report by Consortium for Information & Software Quality (CISQ) [1] published in 2021, identifying and fixing bugs account for 50% of the total expenses of software system development. Therefore, many software fault prediction (SFP) techniques have been proposed to correctly predict software faults in the early stage of the software life cycle. The software fault prediction process intends to identify faulty software components and assist developers in allocating their efforts and resources more optimally during the software development and testing phases. In the traditional software fault prediction models, the historical version of software is utilized to train learners and then use these models to predict whether new modules are faulty or not. The historical dataset contains values of many software metrics (e.g., Line of Code -LOC, Depth of Inheritance Tree - DIT, Coupling Between Objects - CBO) and their labels (faulty or non-faulty). However, the SFP models using software metrics cannot capture the syntax and semantic features of source code and minimize the performance of predictive models [2]. To handle this issue, several studies have been reported to leverage deep learning techniques to extract semantic and syntax information from source code and perform fault prediction. A semantic Long Short-Term Memory (LSTM) framework was proposed by Liang *et al.* [3] to leverage token sequences extracted from the program's Abstract Syntax Tree (AST) to build and perform fault prediction. They concluded that their proposed framework outperformed three other state-of-the-art fault prediction models in both Within-Project Fault Prediction (WPDP) and Cross-Project Fault Prediction (CPDP). Cai *et al.* [4] presented a tree-based-embedding convolutional neural network with transferable hybrid feature learning (TBCNN-THFL) for fault prediction in CPDP. Their experimental results showed that their proposed method exhibited better performance than the current models. At present, many studies used various kinds of deep learning models for building software fault prediction models and achieved promising results. In this paper, we perform a comparison of different deep-learning models including Deep Belief Network (DBN), Convolutional Neural Networks (CNN), Recurrent Neural Network (RNN), and Long-Short Term Memory (LSTM) that are utilized for building software fault prediction models by extracting syntactic information from the program's AST. Our experiment was conducted on seven Apache projects in WPDP. The results have shown that two models Long Short Term Memory and Recurrent Neural Network have high accuracy.

2. Related Work

2.1. Software Fault Prediction

In the existing literature, numerous studies have delved into software fault prediction, as evidenced in [5], [6]. Fault prediction techniques [7], [8] typically construct models using software modules and their labels (faulty or non-faulty), then employ these models to forecast whether new modules contain faults. Many fault prediction studies utilize software metrics to build machine learning classifiers. These metrics often include traditional static code measures such as Halstead metrics [9], McCabe metrics [10], CK features [11], MOOD features [12], and various others. Additionally, other information extracted from software projects is utilized for fault prediction purposes. In addition to traditional static code metrics, different machine learning models have been utilized as fault prediction classifiers, including Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), Neural Network (NN), and others. Alongside these models, additional information is often extracted from software projects to enhance fault prediction accuracy. For instance, Loyola *et al.* [13] utilized developer activity data to construct dependency graphs and automatically generated features from these graphs for fault prediction. Jiang *et al.* [14] introduced an approach that

extracts characteristic vectors, bag-of-words, and metadata features, to construct a fault prediction model separately for each developer. Furthermore, recent research has also delved into change-level fault prediction, focusing on features such as change diffusion, change size, change purpose, etc. These features are leveraged to predict changes that may introduce faults [15], [16], [17].

In WPDP, the fault prediction model is constructed by using modules from the old version of a program and then employed to forecast the faulty modules of the new version. There are many traditional models have been utilized in prior studies to conduct WPDP. Specifically, Wang and Li [8] assessed the Naive Bayes fault prediction model across 11 datasets from the PROMISE data repository. Jing *et al.* [18] introduced the dictionary learning technique for fault prediction, resulting in enhanced WPDP performance. Tong *et al.* [19] applied stacked auto-encoders and ensemble learning to construct fault prediction models. Their experimental results on NASA datasets suggesting their approach surpassed traditional fault prediction classifiers.

2.2. Deep learning and software engineering

Software fault prediction has seen the recent application of various deep learning algorithms. Wang *et al.* [20] introduced an approach that utilizes a Deep Belief Network model to automatically learn semantic features using token vectors extracted from the programs's abstract syntax trees. Their evaluation on 10 open-source projects demonstrated a superior performance compared to traditional techniques based on software metrics. Li *et al.* [21] applied Convolutional Neural Networks (CNN) to tokenized source code for fault prediction, arguing CNN's superiority in capturing local patterns over DBN. They introduced DP-CNN, a framework combining CNN and traditional handcrafted features, showing improved SDP performance on average. Besides, Dam *et al.* [22] employed a tree-based Long Short-Term Memory (LSTM) model to derive semantic features from programs' abstract syntax trees for fault prediction.

In the above studies, many deep learning techniques have been exploited to enhance the performance of software fault prediction models such as CNN, DBN, LSTM, etc. Therefore, our paper differs from the existing approaches in that we perform a comparative study to examine different deep learning model's performance in predicting software faults. Particularly, we used Continuous Bag-of-Words to convert token vectors extracted from the Abstract Syntax Tree into fixed-length numerical vectors. Then, the collected vectors are fed to four deep learning models including Deep Belief Networks (DBN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and Long-Short Term Memory (LSTM) to perform fault prediction. We perform the comparison on these deep learning models used for learning hidden semantic information, while the existing approaches are usually based on software metrics.

3. The approach

Figure 1 illustrates the proposed approach. Initially, it retrieves a token sequence from the abstract syntax tree of every file in both the training and test sets. Then, this token sequence undergoes a transformation into a vector sequence, wherein each token is converted into a fixed-length real-valued vector. The token embedding process utilizes a pre-trained mapping table generated with a CBOW model. Afterward, the vector sequences produced in the training set are utilized as inputs for four deep-learning models to perform prediction. Finally, the performance of these models is evaluated and compared.

3.1. Parsing source code and extracting features

To capture the semantic information of a program, we utilize the state-of-the-art approach in [20] to extract token sequences from its source code. This involves initially constructing an Abstract Syntax Tree (AST) for each Java file in the dataset. Subsequently, we extract the following three types of nodes from the AST: Firstly, for method invocation and class instance creation nodes, we extract and annotate them based on their method or class names; for example, the method *search()* is extracted and recorded as *search*.

Secondly, declaration nodes, including method declarations, type declarations, and enum declarations, are extracted and annotated with their respective names; Control-flow nodes, encompassing statements or clauses related to the control flow of a program (such as *if* statements, *for* loops, *while* loops, and *catch* clauses), are extracted. These nodes are annotated based on their statement types, e.g., an *if* statement is annotated as *if*, and a catch clause is annotated as *catch*. We exclude local nodes like assignment because they are confined within the method scope, and varying methods may assign different meanings to variables with identical names. The selected AST nodes are shown in Table 1. Finally, the token sequence of each Java file includes of three kinds of token.

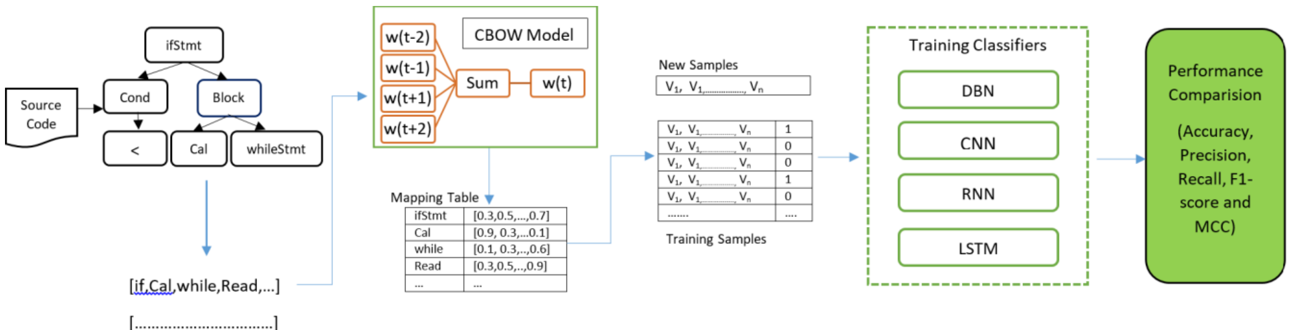

Figure 1. The proposed approach

Table 1. *The selected AST nodes*

Node Category	Node Type
Nodes associated with class nodes and method invocations	MethodInvocation, SuperMethodInvocation, ClassCreator
Declaration nodes	PackageDeclaration, InterfaceDeclaration, ClassDeclaration, ConstructorDeclaration, MethodDeclaration, VariableDeclaration, FormalParameter
Control flow nodes	IfStatement, ForStatement, WhileStatement, DoStatement, AssertStatement, BreakStatement, ContinueStatement, ReturnStatement, ThrowStatement, TryStatement, SynchronizedStatement, SwitchStatement, BlockStatement, CatchClauseParameter, TryResource, CatchClause, SwitchStatementCase, ForControl, EnhancedForControl
Other nodes	BasicType, MemberReference, ReferenceType, SuperMemberReference, StatementExpression

3.2. CBOW training

In this study, we consider tokens within a programming language as analogous to words in a natural language. Thus, we employ the Continuous Bag-of-Words (CBOW) model [23] to acquire vector representations of these program tokens. CBOW, commonly used in natural language processing (NLP), facilitates the creation of distributed representations for words. Its methodology involves initializing word vectors with random real numbers, traversing the corpus, and predicting the distributed representation of the current word based on the contextual window. To effectively illustrate CBOW, it is crucial to introduce the concept of word to vector (word2vec). In NLP, while machine learning models are indispensable for processing natural language data, they face the challenge of comprehending human language directly. CBOW [24], [25] addresses this challenge by constructing distributed representations for words. Word vectors are generated during training, the CBOW model's network structure consists of three layers: input, projection, and output. The input layer measures the distributed representation of context words by reading word vectors from the context, the projection layer sums these vectors, and the output layer predicts the target word using hierarchical softmax. All word vectors in the training vocabulary are initialized randomly and optimized alongside the model parameters using algorithms like stochastic gradient descent. After the CBOW pre-training, the mapping table between tokens and token vectors are obtained. In the training and test dataset, tokens are replaced by token vectors through the mapping table. The CBOW model is described as follows.

In a given text, let's denote the *t-th* word as w_t. We'll be using the fundamental CBOW model architecture to learn word embeddings. The model predicts the output word w_t based on the surrounding input words within a specified window. For instance, with a window size of 2, the input words are w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}, ... The input and output embeddings of a word w_t are denoted as $\overrightarrow{x_t}$ and $\overrightarrow{o_t}$, respectively. The CBOW model calculates the hidden representation as follows:

$$\vec{h} = \frac{1}{2c} \sum_{i=-c, i\neq 0}^{c} \vec{x}_{t+i} \quad (1)$$

Where c is the window size. The negative sampling [31] is used to train CBOW model maximizing the given objective function:

$$log\sigma(\overline{h}^T \vec{O}_t) + \sum_{j=1}^{k} log\sigma(-\vec{h}^T \vec{o}^j) \quad (2)$$

Where k indicates the size of negative sample, $\vec{o}^j$ indicates the *j-th* noise word embedding and σ indicates the sigmoid function.

3.3. Building deep learning models for fault prediction and performing comparative performance analysis

The collected token vector sequences for each file in both training datasets are the input of deep learning models to predict fault. In this study, four different deep learning models are examined for performing comparative performance. We choose the standard architecture of each deep learning model as a fault prediction model. Particularly, the LSTM model includes a fully connected input layer, a LSTM layer, a mean-pooling layer, and an output layer. For CNN, the architecture consists of an embedding layer, a convolution layer, a max-pooling layer, a fully-connected layer, and an output layer working as a Logistic Regression classifier. The RNN model contains an input layer, a recurrent connection, a hidden state, and an output layer. Finally, the DBN model consists of an input layer, a hidden layer, and an output layer. Tensorflow [26] is employed to implement the fault prediction models. Afterward, we use the test dataset to evaluate the performance of four deep-learning models.

4. Experiment

4.1. Experimental Design

Deep learning models have been utilized for fault prediction in recent years. In this study, we make an empirical comparison of four deep learning models when they performed fault prediction based on semantic features extracted from the program's ASTs. We evaluate the performance of four deep learning models, namely DBN, CNN, RNN and LSTM on seven Apache datasets (Camel, Jedit, Log4j, Xalan, Synapse, Lucence, Xerces). During training process, the model optimizes its parameters to minimize the loss function, which quantifies the difference between the predicted fault probabilities and the actual ground truth labels. Finally, the performance the software fault prediction models based on above deep learning techniques is evaluated on a separate test set. The output of predictive models indicates either 0 (non-faulty) or 1 (faulty).

4.1.1. Datasets

In this experiment, seven Java open-source datasets with description details such as the name of each project, the release versions, the average number of files and the average faulty rate are presented in Table 2. Based on the version numbers and class names, we obtained the source code of each file from Github [27]. The corresponding fault prediction datasets include 20 static features and fault labels (faulty or non-faulty) come from PROMISE [28]. The datasets were collected from many projects that have various sizes (ranging from 150 to 792 files) and fault proportions (a minimum value of 15.7% and a maximum value of 62.49%). For within-projects, two versions of the same project are selected for the training data and the test data (e.g. using Camel v1.2 as the training set and Camel v1.4 as the test set).

Table 2. *The used Apache projects*

Project	Versions	Avg. Files	Fault Rate
Camel	1.2, 1.4, 1.6	792	23.76
Jedit	4.0, 4.1, 4.2	296	27.63
Log4j	1.0, 1.1, 1.2	150	50.44
Xalan	2.4, 2.5	780	36.53
Synapse	1.1, 1.2	211	23.37
Lucence	2.0, 2.2	258	54.89
Xerces	1.2, 1.3	447	15.7

4.1.2. Hyperparameter setting

Optimizing hyperparameters is an important task for maximizing performance outcomes. Within this investigation, we conduct various experiments aimed at identifying optimal parameter configurations. The list of hyperparameters includes the following:

- Epoch: epoch is a single iteration through the entire training dataset. We specifically explored epochs spanning from 50 to 500. In the case of four models, 200 epochs emerged as optimal, enabling effective learning from the data while averting overfitting.

- Batch-size: To find the best balance between training stability and how efficiently our computer could use its memory, we experimented with batch sizes between 16 and 64. Considering both our dataset size and memory limitations, a batch size of 32 emerged as the most suitable choice.

- Drop-out: Our experimentation involves dropout ratios ranging from 0.2 to 0.7 in increments of 0.1. For our experiment, a higher dropout ratio of 0.5 is employed in four models, reflecting its susceptibility to overfitting due to the intricate temporal dependencies inherent in sequential data.

- Learning Rate: The learning rate regulates the extent to which the model's weights adapt in response to the loss gradient. A well-tuned learning rate facilitates steady and effective convergence. In our findings, a learning rate of four models is 0.001 which demonstrates superior effectiveness.

- Optimizer Function: The selection of optimizer plays a key activity in shaping the training dynamics. We opt for the Adam optimizer, a variant of stochastic gradient descent, for four models. Adam stands out for its ability to dynamically estimate pertinent statistics from the data, offering adaptive learning rate features alongside computational efficiency.

4.1.3. Performance measures

In this study, we choose three evaluation measures Precision (P), Recall (R) and F1- score (F1) to compare the performance of deep learning models which are utilized for fault prediction based on semantic features. All the above performance metrics are calculated by true positive (TP), false positive (FP), and false negative (FN). Here is a brief description of these metrics:

$$P = \frac{TP}{TP+FP} \qquad R = \frac{TP}{TP+FN}$$

$$F1-score = 2*P*R/(P+R)$$

- True positive is the number of faulty instances predicted correctly as faulty instances;

- False positive is the number of non-faulty instances is predicted as faulty instances;

- False negative is the number of faulty instances that is predicted as non-faulty instances.

4.2. Experimental results

This section illustrates the experimental results on seven fault datasets to compare the performance of four deep learning models that are applied for predicting faults based on semantic features. For each project, the old version with a small-size dataset is used as the training data, and the new version with a large-size dataset is used as the test data. The versions are described as P_s -> P_t where P_s and P_t are defined as the source and target project, respectively. Table 3 shows the precision, recall and F1-score values for four deep learning models. The significant values are highlighted in bold. Overall, LSTM reaches F1-score values of 48.68%, 64.29%, 72.63%, 72.65% and 46.23% on Camel, Jedit, Log4j, Xalan and Xerces, respectively. Additionally, the second highest F1-scores are indicated by RNN with values of 70.18%, 69.42%, 48.91% and 39.29% on the Log4j, Xalan, Synapse and Xerces datasets. While CNN and DBN achieve the highest F1-score values of 71.10% and 65.61% on the Lucence and Synapse datasets, respectively. The above results show that LSTM outperforms the other models on 4 of 7 datasets of the experiments. On average, LSTM achieves a F1-score value, which is 7.5% higher than the other ones. The combination of CBOW and LSTM aims to capitalize on the unique strengths of each model: LSTM's proficiency in contextual understanding and sequential modeling, complemented by CBOW's efficient embedding capabilities. LSTM outperforms traditional RNNs due to its ability to mitigate issues such as vanishing and exploding gradients, which hinder RNNs in capturing long-term dependencies [29], [30]. Moreover, CNNs excel in image recognition tasks, especially with high-dimensional data, as they are adept at capturing local patterns.

Table 3. *Performance comparison of DBN, CNN, RNN and LSTM*

Project	Version	DBN			CNN			RNN			LSTM		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
Camel	1.4->1.6	26.77	57.64	35.30	42.35	61.71	**50.05**	46.14	50.00	47.99	71.16	50.32	48.68
Jedit	4.0->4.1	54.05	71.85	61.30	74.30	58.96	62.27	51.09	71.48	59.59	59.75	67.23	**64.29**
Log4j	1.0->1.1	53.89	86.55	68.20	71.16	50.32	49.23	60.22	**75.61**	**70.18**	65.77	81.10	**72.63**
Xalan	2.4->2.5	58.31	56.27	57.80	59.15	77.81	67.60	74.21	51.16	**69.42**	67.77	78.19	**72.65**
Synapse	1.1->1.2	53.57	71.26	**65.61**	40.08	51.74	44.59	46.91	50.60	**48.91**	46.66	51.50	47.18
Lucence	2.0->2.2	64.35	60.45	63.70	64.95	80.34	**71.10**	61.40	74.50	67.40	60.84	78.02	68.36
Xerces	1.2->1.3	74.23	35.66	46.23	50.12	28.86	36.60	50.23	64.87	**39.29**	43.77	50.62	**46.23**
Average		69.29	48.05	54.96	57.53	54.60	53.85	55.81	69.68	53.34	52.30	64.32	**57.25**

5. Conclusions

Recently, deep learning techniques have been exploited to generate the syntax and semantic information that is important capability for constructing high accuracy prediction models. In this study, we conducted an empirical comparison of the performance of four deep learning models. We employed a word embedding call CBOW to convert token sequences into fixed-length numerical vectors that could be fed to deep learning models for performing prediction. The experiment was carried out on seven public datasets. From the experimental results, LSTM shows a better performance in precision, recall and F1-score compared with other deep learning models in fault prediction. In the future, more deep learning models such as Transformer and CodeBert may be employed to build predictive models that able to capture both semantic and syntactic information in source code.

Acknowledgments: This research is funded by Funds for Science and Technology Development of the University of Danang under project number B2022-DN07-02.

REFERENCES

[1] H. Krasner, "*The Cost of Poor Software Quality in the US: A 2020 Report*", Consortium for Information & Software Quality, January 2021.

[2] X. Zhou and L. Lu, "Fault prediction via lstm based on sequence and tree structure", in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 366–373.

[3] H. Liang, Y. Yu, L. Jiang, and Z. Xie, "Seml: A semantic lstm model for software fault prediction", *IEEE Access*, vol. 7, pp. 83 812–83 824, 2019.

[4] Z. Cai, L. Lu, and S. Qiu, "An abstract syntax tree encoding method for cross-project fault prediction", *IEEE Access*, vol. 7, pp. 170 844–170 853, 2019.

[5] Z. Li, X.-Y. Jing, and X. Zhu, "Progress on approaches to software fault prediction", *IET Software*, vol. 12, no. 3, pp. 161–175, 2018.

[6] J. Nam, S. J. Pan, and S. Kim, "Transfer fault learning", in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 382–391.

[7] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Using the support vector machine as a classification method for software fault prediction with static code metrics", in *Engineering Applications of Neural Networks: 11th International Conference, EANN 2009, London, UK, August 27-29, 2009. Proceedings 11*. Springer, 2009, pp. 223–234.

[8] T. Wang and W.-h. Li, "Naive Bayes software fault prediction model", in *2010 International conference on computational intelligence and software engineering*. Ieee, 2010, pp. 1–4.

[9] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[10] T. J. McCabe, "A complexity measure", *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[12] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the mood set of object-oriented software metrics", *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491–496, 1998.

[13] P. Loyola and Y. Matsuo, "Learning feature representations from change dependency graphs for fault prediction", in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 361–372.

[14] T. Jiang, L. Tan, and S. Kim, "Personalized fault prediction", in 2013 *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Ieee, 2013, pp. 279–289.

[15] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time fault prediction", in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 11–19.

[16] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "Multi: Multi-objective effortaware just-in-time software fault prediction", *Information and Software Technology*, vol. 93, pp. 1–13, 2018.

[17] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time fault prediction", *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019.

[18] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, "Dictionary learning based software fault prediction", in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 414–423.

[19] H. Tong, B. Liu, and S. Wang, "Software fault prediction using stacked denoising autoencoders and two-stage ensemble learning", *Information and Software Technology*, vol. 96, pp. 94–111, 2018.

[20] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for fault prediction", in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 297–308.

[21] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software fault prediction via convolutional neural network", in *2017 IEEE international conference on software quality, reliability and security (QRS)*. IEEE, 2017, pp. 318–328.

[22] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction", *arXiv preprint arXiv:1708.02368*, 2017.

[23] K. Duan, S. S. Keerthi, W. Chu, S. K. Shevade, and A. N. Poo, "Multicategory classification by soft-max combination of binary classifiers", in *Multiple Classifier Systems: 4th International Workshop, MCS 2003 Guildford, UK, June 11–13, 2003 Proceedings 4*. Springer, 2003, pp. 125–134.

[24] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality", *Advances in neural information processing systems*, vol. 26, 2013.

[25] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", *arXiv preprint arXiv:1301.3781*, 2013.

[26] M. Abadi *et al*., "TensorFlow: A System for Large-Scale Machine Learning", in *the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA.

[27] "The Apache software foundation", *github.com.* [Online]. Available: https://github.com/apache [Accessed June 03, 2024].

[28] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to fault prediction", in *Proceedings of the 6th international conference on predictive models in software engineering*, *PROMISE '10. Association for Computing Machinery,* New York, NY, USA, 2010.

[29] H. Wang, W. Zhuang, and X. Zhang, "Software defect prediction based on gated hierarchical LSTMs", *IEEE Transactions on Reliability*, vol. 70, pp. 711–727, 2021.

[30] T. Y. Yu, C. Y. Huang, and N. C. Fang, "Use of Deep Learning Model with Attention Mechanism for Software Fault Prediction," in *Proceedings of 8th International Conference on Dependable Systems and Their Applications (DSA)*, China, 2021, pp. 161-171.

[31] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and Their Compositionality", in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, 2013, pp. 3111–3119.