Analysis of Various Software Defect Prediction Techniques

Pooja Gupta[1], Komal Ahuja[2]

[1]M.Tech Student, Computer Science Engineering Department, GRIMT,Radaur(Yamunanagar), Haryana, India
[2]Assistant Professor, Computer Science Engineering Department, GRIMT,Radaur(Yamunanagar), Haryana, India

---

Abstract

Software defect along with an intrinsic element of software product is also an important aspect of software quality. Software defects are an unavoidable co product of the developed software. In addition to this, the guarantee of software quality assurance is not so easy and requires a lot of time too. There are different ways to define defects, such as in terms of quality. The defects may be present in all products whether it is a small program or large-scale software system. A number of defects can be discovered without any complexity. The various steps are applied for the software defect prediction which includes pre-processing, feature extraction and classification. The various techniques which are proposed in the previous time for the software defect prediction are reviewed and analysed in this paper. The techniques are reviewed in terms of technique description and their findings in terms of various parameters

Keywords - Software defect, Prediction, Machine learning, Pre-processing, Feature extraction, Classification

1. Introduction

Because of the increasing complexity of today's software and enhance chances of failures, ensuring reliability has become an important concern. Organisations such as Google employ code review and unit testing to identify problems in new code and increases reliability. Meanwhile, testing each code unit is not practical and human code reviews are labour- intensive [1]. With less funding for software projects, it is fruitful to detect potential issues immediately. As a result, software defect prediction algorithms are commonly employed to automatically identify possible mistakes, enabling developers to make optimal consumption of their resources. Software defect prediction includes, creating classifiers that analyse data such as change history and complexity of code to identify code segments with potential flaws. This practice ensures code reviewers to allocate their efforts with more strategy and receive warnings about potentially buggy code regions based on the prediction outcomes [2]. These code sections may include alterations, files, or processes. In the typical fault prediction process, there are two main stages: feature extraction from source files and the creation of a classifier using various machine learning techniques. Previous research emphasises on enhancing the precision of predictions has mainly involved manually crafting discriminative features or combining characteristics [3]. Halstead features based on operators and operands, McCabe features based on dependencies, and CK features for object-oriented programs are some of the examples. However, traditional hand-crafted features often overlook the intricate semantics and clearly-defined syntax concealed in the Abstract Syntax Trees (ASTs) of programs.

Prediction models, important to Software Defect Prediction (SDP), are crucial in anticipating software faults. Number of methods and algorithms have been used to increase the correction of Software Defect Prediction (SDP) models, till the fundamental stages of SDP can be succinctly outlined. Primarily, collection of data includes collecting both pure and defective code samples from software repositories [4]. Next to this, a dataset is constructed by extracting related qualities from the compiled samples, with forthcoming steps including dataset balancing if any imbalance is identified, model training utilizing the prepared dataset, and the upcoming predictions of problematic components within new software datasets using the trained model. Consequently, the performance of the SDP model is evaluated. This innovative process allows for ongoing refinement and improvement over time.

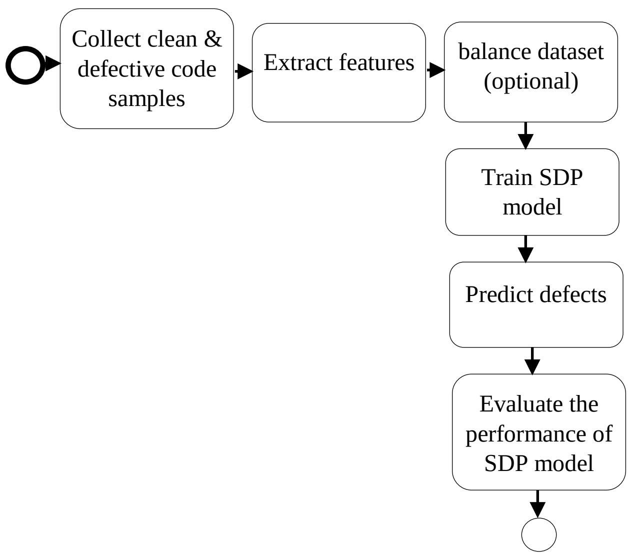
Collect clean & defective code samples
Extract features
balance dataset (optional)
Train SDP model
Predict defects
Evaluate the performance of SDP model

Figure 1: Software Defect Prediction Process

As described in Figure 1, the first step of the process involves the collection of both functional and faulty code examples [5]. Software data, creating source codes, commit messages, bug reports, and other artifacts exists in different formats and is typically sourced from repositories and archives. Next to this, the second step in the Software Defect Prediction (SDP) process is feature extraction. During this phase, various data sources such as commit logs, messages, source codes, and software artifacts are changed into metrics that serve as input for model training. During feature extraction, it's important to consider the nature of the input data which may encompass McCabe metrics, CK metrics, convert histories, assembly code, and source code, along with representation of data [6]. Along with metric-based data, ample of deep learning techniques presently provide automated feature extraction capabilities and can handle the data with high-dimension and more complexity. An optional step sometime follows after this. Given that defect datasets typically consist of a dramatically lowest percentage of defective components as compare to fine ones, this stage involves balancing of data. The unbalanced structure of class can cause misleading outcomes for different performance evaluation metrics used in assessing Software Defect Prediction (SDP) [7], therefore badly affect most SDP strategies. Numerous methods like oversampling can be employed to minimize this concern and increase the performance of SDP.

The fourth phase of the Software Defect Prediction (SDP) process involves analysing the problematic areas within the software. At this stage, mainly focus is on selecting optimal Deep Learning (DL) algorithms and techniques, which may encompass various topologies such as Convolutional Neural Networks [8], and machine learning categories like supervised or unsupervised learning. In addition to this, during this step an important consideration is determining the level of granularity at which the sections of problems are to be detected, which could range from module and file levels to class [9], function, or even phrase levels. The upcoming step entails predicting the faulty parts within fresh (test) data by using the model trained in the previous stage. Lastly, the prediction made in this step acts as input for the final phase of the SDP process. The final phase of the Software Defect Prediction (SDP) process including the evaluation of created model. Different alternatives, such as the area under the curve and the F-measure, can be employed to increase the performance of SDP model [10]. Then prediction models are evaluated and compared them with other pertinent studies by using one or more of these measures. In the realm of software defect prediction, Machine learning classifiers have garnered a lot of attention due to their ability to predict the mistakes of software by analysing code and features extraction from raw data sources. Over the past few decades, ample of machine learning-based methods have been developed specifically to identify software issues [11]. One prominent machine learning technique for tasks involving reduction of features, detection, regression, and classification is the support vector machine. Locate a hyperplane (or series of hyperplanes for more classes) is main focus that splits two classes. The plane is selected in a way by increasing the distance from the two data clusters, that separates the two classes of data points. Because it offers some possibility that future data can be classified more precisely, it is meaningful to increase this kind of distance metric. Support vectors are the data points that are nearest to the hyperplane [12]. The position of hyperplane is adjusted in according to the support vector positions to preserve the maximum distance between the data points of both classes. A supervised learning method called Random Forest defines the concept of merging learning models to gain a superior performance superior to that of a single model. Specifically, it integrates several Decision Trees to produce a prediction with more accuracy. In order to construct a lot of decision trees from the dataset and merge them into a final judgment, this approach called randomness [13]. However, the variables with maximum depth allows you to set the tree's depth, and n_estimators enable us to limit the count of decision trees in the tree. The ability to calculate a significance score for every feature—which gives us some insight into the features that are important to get an accurate prediction—is one benefit of employing random forest. The number of neighbours such as k is the primary determining feature in k-nearest neighbour. This classifier is used for both regression and classification problems [14]. In both circumstances, the output relies on whether k-NN is applied to the regression or classification model, and the k closest training examples are taken into consideration as input [15]. The algorithm work on the presumption that the new and present data is same. When a new case or set of data is received, the algorithm places it in the category that shares the lot of similarities with the existing categories [16].

2. Literature Review

M. Ali, et.al (2024) suggested an intelligent ensemble framework that integrated various classification techniques to forecast software defects [17]. Their approach involved a two-phase procedure designed for detecting defective modules. Initially, the focus was on utilizing four supervised machine learning (SML) methods: Random Forest (RF), Support Vector Machine (SVM), Naïve Bayes (NB), and Artificial Neural Network (ANN). They employed an iterative parameter optimization technique to enhance the accuracy of these methods. Subsequently, the second phase aimed to amalgamate the accuracy of each method into a voting ensemble to predict defects in software effectively. This framework significantly improved the accuracy and reliability of software defect prediction. The proposed framework was evaluated using seven datasets sourced from the NASA MDP repository. The experimental outcomes demonstrated the superiority of their approach over existing methods in predicting software defects.

R. Haque, et.al (2024) introduced an innovative technique called heterogeneous cross-project defect prediction (HCDP), employing encoder networks and a transfer learning (ENTL) model to anticipate software defects [18]. The encoder networks (ENs) were leveraged to extract significant features from both the source and target datasets. Furthermore, negative transfer in transfer learning (TL) was alleviated by incorporating an augmented dataset containing pseudo-labels and the source dataset. The model was trained on a single dataset and evaluated on sixteen datasets derived from four public projects. A comparative analysis was conducted against traditional methods, employing cost-sensitive learning (CL) to address imbalanced class issues. Experimental results demonstrated the robustness of the proposed method in predicting software defects, outperforming traditional methods across metrics such as PD, PF, F1-score, G-mean, and AUC.

Y. Al-Smadi, et.al (2023) introduced an innovative approach for predicting software defects by implementing 11 machine learning (ML) techniques across 12 diverse datasets [19]. They utilized four diverse meta-heuristic algorithms: particle swarm optimization (PSO), genetic algorithm (GA), harmony algorithm (HA), and ant colony optimization (ACO) to select features. Additionally, they applied the synthetic minority oversampling technique (SMOTE) to address imbalanced data issues. Moreover, they used the Shapley additive explanation (SAE) framework to identify decisive features. Experimental results indicated that gradient boosting (GB), stochastic gradient boosting (SGB), decision trees (DTs), and categorical boosting (CB) algorithms performed exceptionally well, achieving an accuracy and ROC-AUC of over 90%. Furthermore, the superiority of the categorical boosting (CB) algorithm was demonstrated against other methods for predicting software defects.

A. Wang, et.al (2023) devised a Federal Prototype Learning leveraging Prototype Averaging (FPLPA) method and integrating federated learning (FL) with prototype learning (PL) to forecast heterogeneous defects in software [20]. They employed the one-sided selection (OSS) algorithm to eliminate noise from local training data and utilized the Chi-Squares Test algorithm to select the optimal subset of features. Subsequently, they introduced the Convolution Prototype Network (CPN) model to create local prototypes, which exhibited greater robustness against heterogeneous data compared to convolutional neural networks (CNNs), mitigating the impact of class imbalances in software data. The prototype served as the communication subject between clients and the server, with the local prototype developed irreversibly to safeguard privacy in the communication process. The final task involved updating the presented model using the loss of local and global prototypes as regularization. They evaluated the developed method using the AEEEM, NASA, and Relink datasets, with simulation outcomes demonstrating its superiority over traditional methods for defect prediction.

S. Kwon, et.al (2023) projected a function-level just-in-time (JIT) software defect prediction (SDP) technique aimed at addressing the challenge of predicting software defects [21]. They prioritized limited testing resources for defect-prone functions. Their approach relied on a pre-trained method, specifically a transformer-based deep learning (TDL) model trained on a large corpus of code snippets to provide defect proneness for altered functions at a commit level. They evaluated the CodeBERT, GraphCodeBERT, and UniXCoder methods on edge-cloud systems, with a primary focus on analyzing their efficacy in this environment. Results indicated that the UniXCoder method outperformed others in the WPDP environment. Furthermore, the projected technique demonstrated stability in predicting software defects.

W. Wen, et.al (2022) introduced a Class Code Similarity-based Cross-Project Software Defect Prediction (CCS-CPDP) method for software defect prediction. Initially, the focus was on converting the code set extracted from the Abstract Syntax Tree (AST) into a vector set [22]. To achieve this, they employed a Doc2Bow and TF-IDF (DTI) method. The second step involved computing similarity between the vector sets of target projects and training projects. Finally, they utilized the principle of the majority decision subordinate category in K-Nearest Neighbor (KNN) to determine the number of occurrences of the same class in the training project. Based on this, the class instance was selected to refine the source project, and predictions were made regarding software defects. The developed method was evaluated against traditional methods, and experimental results indicated its effectiveness, offering higher recall and F1-score in predicting software defects compared to other methods.

S. Kassaymeh, et.al (2022) developed a Salp Swarm Algorithm (SSA) coupled with a backpropagation neural network (BPNN) to predict software faults. Their integrated method, SSA-BPNN, aimed to enhance accuracy in defect prediction by optimizing optimal metrics [23]. They applied various datasets to evaluate the presented method across different parameters including AUC, confusion matrix, sensitivity, specificity, accuracy, and error rate, varying in size and complexity. Simulation results demonstrated that the presented method outperformed conventional methods, achieving higher accuracy in software defect prediction. Moreover, this technique proved to be an effective tool in addressing challenges within software engineering.

W. Wen, et.al (2022) developed a Cross-Project Defect Prediction (CPDP) model named BSLDP to anticipate software defects. They utilized a Bidirectional Long Short-Term Memory (Bi-LSTM) algorithm combined with a self-attention (SA) method to extract semantic information from source code files [24]. The ALC model was employed to extract source code semantics based on source code files, and a classifier, termed BSL, was generated using the semantic information from both the source and target projects to create a predictive framework. To enhance the model, they adopted an equal meshing method to extract semantic information from small code fragments by splitting the numerical token vector. They evaluated the designed model using the PROMISE dataset and found that it effectively predicted defects, improving the F1 score by up to 14.2%, 34.6%, 32.2%, and 23.6% compared to four other techniques, respectively.

C. Yu, et.al (2021) highlighted the use of homomorphic encryption (HE) for defect prediction, introducing a novel technique called HOPE. They presented an algorithm approximation (AA) method to approximate the sigmoid function and selected the Paillier homomorphic encryption (PHE) algorithm to execute logistical regression (LR) [25]. Real-time projects served as experimental subjects for generating the MORPH dataset to evaluate the formulated technique. Subsequently, three control groups were deployed to simulate three different scenarios based on whether the client transmitted encrypted data to the server and whether the formulated technique was employed on the server. The results indicated that when the original LR was deployed on the server to generate the technique using encrypted data, the trained model achieved similar performance, thus preserving data privacy. Moreover, the formulated technique demonstrated greater efficiency in terms of minimal computing cost.

L. Yang, et.al (2021) conducted a study on a hybrid approach combining particle swarm optimization (PSO) and Salp Swarm Algorithm (SSA), termed SSA-PSO, aimed at improving convergence prior to individual updates in SSA [26]. Additionally, they introduced a novel maximum likelihood estimation (MLE)-based fitness function (FF) of metrics, used to initialize metrics. They evaluated this approach using five sets of actual datasets, comparing it against individual techniques. Experimental results demonstrated the stability of the SSA-PSO approach over others, showing higher convergence speed and accuracy. Moreover, the novel FF helped address issues related to slow convergence speed and lower solution accuracy. The investigated approach proved to be more effective in estimating and predicting software defects.

2.1 Comparison Table

Author & Year	Technique used	Dataset	Findings	Limitations
M. Ali, et al. (2024)	Random Forest, Support Vector Machine, Naïve Bayes, and Artificial Neural Network, voting ensemble	Seven datasets from datasets from the NASA MDP repository, namely CM1, JM1, MC2, MW1, PC1, PC3, and PC4	Accuracy= 87.14, Misclassification Rate= 12.86, False Positive Ratio= 0.003, False Negative Ratio= 0.905	This study does not explore sophisticated methods for selecting features to boost the reliability of the proposed model in software defect prediction.
R. Haque, et al. (2024)	Encoder Networks and Transfer Learning	Sixteen datasets from AEEM, NASA, Promise and JIRA.	PD=0.711, PF=0.0911, F1-score=0.475, G-mean=0.770, AUC=0.663	This evaluates the presented model using 16 datasets from four publicly available software defect projects, leaving benchmark datasets like Relink and SOFTLAB for future investigation, while considering methods like SMOTE to address class imbalance more accurately in upcoming research.
Y. Al-Smadi, et al. (2023)	Machine learning and metaheuristic algorithms	Twelve datasets from NASA	Accuracy= 0.931 ROC-AUC=above 90%	This work does not compare meta-heuristic algorithms with filter methods for

				feature selection or evaluate the performance of SMOTE against other resampling techniques.
A. Wang, et al. (2023)	Federated Prototype Learning	AEEEM, NASA and Relink	AUC= 0.8098, G-mean= 0.5978	Numerous metric variations significantly increase the complexity of constructing software defect prediction models.
S. Kwon, et al. (2023)	Function-level justin-time (JIT) SDP) model	CodeSearch	AUC= 0.548, F-measure= 0.667	This work creates SDP models through refining pre-trained models, with the exception of batch size and maximum sequence length, adjusted due to memory constraints.
W. Wen, et al. (2022)	CCS-CPDP	Promise	Recall increases by 41.8% and f1-score increases by 15.9% than baseline models	The proposed approach utilizes limited project source code datasets for evaluation and does not conduct data validation tests from various aspects to verify the performance of the proposed method.
S. Kassaymeh, et al. (2022)	Salp swarm optimizer (SSA), SSA-BPNN	Ant-1.7, Ar, CM, Jedit, KC, PC, MC	AUC= 0.79, Sensitivity= 0.997, Specificity= 0.900, Accuracy= 0.9964,	A limitation observed in the proposed method is its elevated computational expense across most datasets.
W. Wen, et al. (2022)	Deep Learning With Self-Attention	PROMISE	The performance of cross-project defect prediction in terms of F1 by 14.2%, 34.6%, 32.2% and 23.6%, respectively against four baseline (DBN-CP,TCA+, AST-LSTM,DP-CNN) methods.	The proposed model solely extracts semantic information at the level of source code files.
C. Yu, et al. (2021)	HOPE	MORPH	Accuracy=79%, AUC=0.5	Due to the inherent constraint of homomorphic encryption, only integers can be encrypted during the data encryption process
L. Yang, et al. (2021)	Hybrid Particle Swarm Optimization and Sparrow Search Algorithm	PROMISE	Accuracy=92%	With an increasing number of iterations, the optimal, worst, and average values of y gradually decrease.

Conclusion

Software defect along with an intrinsic element of software product, is also an important aspect of software quality. Software defects are an unavoidable co product of the developed software. In addition to this, the guarantee of software quality assurance is not so easy and requires a lot of time too. There are different ways to define defects, such as in terms of quality. However, the defects are generally defined in the form of deviations from specifications or expectations which may be the reason of failure in functioning. It is analyzed that major content is published in the conferences and in the latest trends journals are preferred. In future deep learning approach will be applied for the software defect prediction.

References

[1] R. R. Panda and N. K. Nagwani, "Classification and intuitionistic fuzzy set-based software bug triaging techniques," J. King Saud Univ., Comput. Inf. Sci., vol. 34, no. 8, pp. 6303–6323, Sep. 2022.

[2] P. Oliveira, R. M. C. Andrade, I. Barreto, T. P. Nogueira, and L. M. Bueno, "Issue auto-assignment in software projects with machine learning techniques," in Proc. IEEE/ACM 8th Int. Workshop Softw. Eng. Res. Ind. Pract. (SER IP), Madrid, Spain, Jun. 2021, pp. 65–72, doi: 10.1109/SER-IP52554.2021.00018.

[3] M. Panda and A. T. Azar, "Hybrid multi-objective Grey Wolf search optimizer and machine learning approach for software bug prediction," in Handbook of Research on Modeling, Analysis, and Control of Complex Systems. 2020, doi: 10.4018/978-1-7998-5788-4

[4] Z. J. Szamosvölgyi, E. T. Váradi, Z. Tóth, J. Jász, and R. Ferenc, "Assessing ensemble learning techniques in bug prediction," in Computational Science and Its Applications—ICCSA 2021 (Lecture Notes in Computer Science), vol. 12955. Springer, 2021,

[5] S. F. A. Zaidi, H. Woo, and C.-G. Lee, "A graph convolution networkbased bug triage system to learn heterogeneous graph representation of bug reports," IEEE Access, vol. 10, pp. 20677–20689, 2022, doi: 10.1109/ACCESS.2022.3153075.

[6] A. Goyal and N. Sardana, "An empirical study of non-reproducible bugs," Int. J. Syst. Assurance Eng. Manage., vol. 10, no. 5, pp. 1186–1220, Oct. 2019.

[7] D. Zhan, X. Yu, H. Zhang and L. Ye, "ErrHunter: Detecting Error-Handling Bugs in the Linux Kernel Through Systematic Static Analysis," in IEEE Transactions on Software Engineering, vol. 49, no. 2, pp. 684-698, 1 Feb. 2023

[8] J. Wang, Y. Huang, S. Wang and Q. Wang, "Find Bugs in Static Bug Finders," 2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC), Pittsburgh, PA, USA, 2022, pp. 516-527

[9] J. Lu, F. Li, C. Liu, L. Li, X. Feng and J. Xue, "CloudRaid: Detecting Distributed Concurrency Bugs via Log Mining and Enhancement," in IEEE Transactions on Software Engineering, vol. 48, no. 2, pp. 662-677, 1 Feb. 2022

[10] K. Foss, I. Couckuyt, A. Baruta and C. Mossoux, "Automated Software Defect Detection and Identification in Vehicular Embedded Systems," in IEEE Transactions on Intelligent Transportation Systems, vol. 23, no. 7, pp. 6963-6973, July 2022

[11] K. Tameswar, G. Suddul and K. Dookhitram, "A hybrid deep learning approach with genetic and coral reefs metaheuristics for enhanced defect detection in software", International Journal of Information Management Data Insights, vol. 2, no. 2, pp. 12-20, 19 August 2022

[12] A. Perera, "Using Defect Prediction to Improve the Bug Detection Capability of Search-Based Software Testing," 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, VIC, Australia, 2020, pp. 1170-1174

[13] R. Amankwah, J. Chen, A. A. Amponsah, P. K. Kudjo, V. Ocran and C. O. Anang, "Fast Bug Detection Algorithm for Identifying Potential Vulnerabilities in Juliet Test Cases," 2020 IEEE 8th International Conference on Smart City and Informatization (iSCI), Guangzhou, China, 2020, pp. 89-94

[14] D. Zhang, P. Qi and Y. Zhang, "GoDetector: Detecting Concurrent Bug in Go," in IEEE Access, vol. 9, pp. 136302-136312, 2021

[15] Y. Li, "Improving Bug Detection and Fixing via Code Representation Learning," 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Seoul, Korea (South), 2020, pp. 137-139

[16] J. Lee, J. Choi, D. Ryu and S. Kim, "Holistic Parameter Optimization for Software Defect Prediction," in IEEE Access, vol. 10, pp. 106781-106797, 2022

[17] M. Ali et al., "Software Defect Prediction Using an Intelligent Ensemble-Based Model," in IEEE Access, vol. 13, no. 4, pp. 127-134, 2024, doi: 10.1109/ACCESS.2024.3358201.

[18] R. Haque, A. Ali, S. McClean, I. Cleland and J. Noppen, "Heterogeneous Cross-Project Defect Prediction Using Encoder Networks and Transfer Learning," in IEEE

Access, vol. 12, pp. 409-419, 2024, doi: 10.1109/ACCESS.2023.3343329.

[19] Y. Al-Smadi, M. Eshtay and A. A. Abd El-Aziz, “Reliable prediction of software defects using Shapley interpretable machine learning models”, Egyptian Informatics Journal, vol. 24, no. 3, pp. 386-394, 31 July 2023, doi: 10.1016/j.eij.2023.05.011.

[20] A. Wang, L. Yang, H. Wu and Y. Iwahori, "Heterogeneous Defect Prediction Based on Federated Prototype Learning," in IEEE Access, vol. 11, pp. 98618-98632, 2023, doi: 10.1109/ACCESS.2023.3313001.

[21] S. Kwon, S. Lee, D. Ryu and J. Baik, "Pre-Trained Model-Based Software Defect Prediction for Edge-Cloud Systems," in Journal of Web Engineering, vol. 22, no. 2, pp. 255-278, March 2023, doi: 10.13052/jwe1540-9589.2223.

[22] W. Wen et al., "Cross-Project Software Defect Prediction Based on Class Code Similarity," in IEEE Access, vol. 10, pp. 105485-105495, 2022, doi: 10.1109/ACCESS.2022.3211401.

[23] S. Kassaymeh, S. Abdullah and M. Alweshah, “Salp swarm optimizer for modeling the software fault prediction problem”, Journal of King Saud University - Computer and Information Sciences, 11 February 2022, vol. 34, no. 6, pp. 3365-3378, June 2022, doi: 10.1016/j.jksuci.2021.01.015.

[24] W. Wen et al., "A Cross-Project Defect Prediction Model Based on Deep Learning with Self-Attention," in IEEE Access, vol. 10, pp. 110385-110401, 2022, doi: 10.1109/ACCESS.2022.3214536.

[25] C. Yu, Z. Ding and X. Chen, "HOPE: Software Defect Prediction Model Construction Method via Homomorphic Encryption," in IEEE Access, vol. 9, pp. 69405-69417, 2021, doi: 10.1109/ACCESS.2021.3078265.

[26] L. Yang, Z. Li, D. Wang, H. Miao and Z. Wang, "Software Defects Prediction Based on Hybrid Particle Swarm Optimization and Sparrow Search Algorithm," in IEEE Access, vol. 9, pp. 60865-60879, 2021, doi: 10.1109/ACCESS.2021.3072993.