Predicting the Number of Software Faults using Deep Learning

Wahaj Alkaberi
Computer Science and Artificial Intelligence Department, College of Computer Science and Engineering, University of Jeddah, Saudi Arabia
2100295@uj.edu.sa (corresponding author)

Fatmah Assiri
Software Engineering Department, College of Computer Science and Engineering, University of Jeddah, Saudi Arabia
fyassiri@uj.edu.sa

Received: 22 December 2023 | Revised: 20 January 2024 | Accepted: 28 January 2024

 DOI: https://doi.org/10.48084/etasr.6798

ABSTRACT

The software testing phase requires considerable time, effort, and cost, particularly when there are many faults. Thus, developers focus on the evolution of Software Fault Prediction (SFP) to predict faulty units in advance, therefore, improving software quality significantly. Forecasting the number of faults in software units can efficiently direct software testing efforts. Previous studies have employed several machine learning models to determine whether a software unit is faulty. In this study, a new, simple deep neural network approach that can adapt to the type of input data was designed, utilizing Convolutional Neural Networks (CNNs) and Multi-Layer Perceptron (MLP), to predict the number of software faults. Twelve open-source software project datasets from the PROMISE repository were used for testing and validation. As data imbalance can negatively impact prediction accuracy, the new version of synthetic minority over-sampling technique (SMOTEND) was used to resolve data imbalance. In experimental results, a lower error rate was obtained for MLP, compared to CNN, reaching 0.195, indicating the accuracy of this prediction model. The proposed approach proved to be effective when compared with two of the best machine learning models in the field of prediction. The code will be available on GitHub.

Keywords-deep learning; MLP; CNN; software testing; prediction; fault; class imbalance

I. INTRODUCTION

Quality attributes are among the most important desired attributes in developed software, and faults have a significant negative impact on software quality. The cost of software projects rises as they progress towards the deployment phase, whereas the presence of faults entails additional costs that can affect the allocated budget and actual time [1]. The field of Software Fault Prediction (SFP) is not limited to software engineering only, it has also contributed to the medical [2-3] and industrial fields [4]. Recently, SFP, a method of designing models to classify software units as faulty [5], has gained considerable attention [6]. The SFP predicts defective software modules/units before the testing phase [7]. Various supervised Machine Learning (ML) models, such as Decision Tree (DT), Neural Networks (NNs), and Support Vector Machines (SVMs) [8-10], as well as unsupervised algorithms, namely K-means++ and QuadTree K-means (QDK) [10], have been used with varying degrees of success. Also, ensemble learning models have been developed and utilized for this purpose [11]. Some studies have applied Principal Component Analysis (PCA) and feature selection techniques to improve the prediction results [12, 13]. Furthermore, a few researchers have turned to Deep Learning (DL) as it produces a higher accuracy rate for SFP [7, 14-17]. The existing studies focus on classifying software units as faulty or not. However, predicting the number of faults is more useful as it helps developers focus more on software units with a large number of faults, thereby reducing the testing effort [7-14]. Software developers aim to find the best solutions to predict the number of faults in early phases. Many models and techniques have been tested. With the emergence of DL models many experiments have proven the ability of DL to predict software faults.

Given the advantages that Convolutional Neural Networks (CNN) have in terms of their ability to predict and adapt to structure, quality, and number of data, this feature is considered important in the field of fault prediction, because the data that we obtain for prediction vary due to the existing differences in programs and features [18-20]. Furthermore, the source code usually consists of millions of lines and hundreds of files [6], which is a huge amount of data that can be better utilized with DL techniques [21]. CNNs along with MLP models have been effectively applied in the area of prediction [7]. In this study, DL models will be developed using CNNs and Multi-Layer

Perceptron (MLP) to predict the number of faults. The model design of [22] was followed, with modifications producing better results. To further improve the prediction results, SMOTEND technique was applied to solve data imbalance, which introduced majority bias and considered noise. The main contribution of the current study lies in the investigation of the effectiveness of the proposed CNN and MLP design to the field of SFP. Additionally, the acquired results were compared with those of well-known ML models to showcase the effectiveness of DL. As far as is known, only the studies [22-34] have considered DL algorithms to predict the number of faults at the class level.

II. RELATED WORK

Initially, SFP was approached as a classification problem, aiming to predict whether software units were faulty. Researchers focused on ML models in earlier studies. Authors in [9] summarized the related articles published between January 1991 and October 2013 and identified the best techniques and measures for SFP models [9]. The results showed that Random Forest (RF) was the best-used model with 80% accuracy. Authors in [8] compared the performances of three of the most popular supervised ML techniques: MLP, Bayesian network, and Naïve Bayes (NB). The findings indicated that NB achieved the highest accuracy (97%).

In addition to supervised ML techniques, unsupervised ML algorithms have been also used. The author in [23] conducted a comparative study on clustering algorithms, specifically K-means and their variants (K-means++, QDK, and Fuzzy C-means (FCM)). NASA datasets with 29 static code attributes were used, with the QDK algorithm exhibiting the best performance. Research continues to improve classification accuracy. Authors in [10] predicted software faults using ML and source code metrics. They also investigated the effect of the feature selection technique on the prediction performance. In a study conducted employing an NASA project, RF achieved the highest accuracy of 93.7%, while correlation-based feature subset selection (CFS) led to a slight increase in performance, reaching 93.84%. Authors in [24] studied the effect of feature-selection techniques utilizing five classification algorithms: MLP, SVM, k-Nearest Neighbors (kNN), NB, and Logistic Regression (LR), with a telecommunications software system [24]. LR exhibited the best performance, whereas the wrapper-based subset selection technique outperformed the other techniques. Authors in [6] performed a new experiment by combining well-known classification algorithms with PCA, which reduced dataset dimensionality. The datasets were taken from Kaggle using the WEKA simulation tool and the experiment achieved an accuracy of 98.70% implementing the SVM model.

Considering improvements in prediction models, in [25], 28 datasets from the PROMISE repository were utilized to investigate the performance of seven ensemble techniques using three classification algorithms. A total of 532 models were built to demonstrate the effectiveness of ensemble techniques in predicting faults. While most existing research has focused on classifying software units as faulty (buggy) or non-faulty (clear), only a few studies have focused on predicting the number of faults [11, 14, 26, 27]. The DTR model aimed to predict the number of faults, including intra- and inter-release predictions, along with several software project datasets [14]. DTR yielded good accuracy, with the intra-release prediction showing better results. Additionally, GP and NNs were used to predict the number of faults employing ten software projects from the PROMISE repository. GP achieved better results for large datasets [27]. The quality of the classification model is highly dependent on the quality of the data [28]. Authors in [5] focused on two data-related problems: class overlap and data imbalance. These problems were solved utilizing neighborhood cleaning and random oversampling. After data processing, four models were trained: kNN, NB, DT, and LR. The Synthetic Minority Oversampling Technique (SMOTE) and an ensemble classifier were used whereas NASA datasets were considered and the proposed approach provided improved results. While ML techniques are widely employed for software unit classification, the research community is still experimenting to find better models, with a few researchers utilizing DL [7, 15, 16, 29]. Authors in [29] used DL algorithms, including ANNs, CNNs, Self-Organizing Maps (SOMs), Learning Vector Quantization-3 (LVQ3), and multipass-LVQ (multiLVQ). CNN achieved the best performance, reaching 99.28%. Additionally, dimensionality reduction techniques have also been applied to improve DL algorithms. Authors in [13] applied PCA and Kernel Principal Component Analysis (KPCA) using DT and ANNs, with the ANN algorithm combined with the KPCA yielded the best results. Authors in [22] designed a CNN to predict the number of software faults, utilizing SMOTEND to overcome the problem of imbalanced data. In the most recent considered studies [30, 31], other DL techniques were implemented to predict software faults, namely Long Short-Term Memory (LSTM), Bidirectional LSTM (BILSTM), and Radial Basis Function Network (RBFN) and to classify the modules into faulty or non-faulty. LSTM and BILSTM gave the better performance with 93.53% and 93.75% accuracy, but RBFN was the fastest [30]. Utilizing DL models based on Recurrent Neural Networks (RNNs), the accuracy reached 95% [31].

III. DATASET

The considered dataset consisted of 12 publicly available software projects obtained from the PROMISE repository, which was used for fault prediction studies. These projects include Apache Ant, Camel, Xerces, Xalan, Ivy, Poi, Log4j, Velocity, Lucene, and Synapse, JEdit, and PROP. Table I summarizes the dataset. In this study, the term class is utilized to refer to each module. The complete details of the data can be found in [32]. Each project consisted of the project name, version number, class name, and 20 object-oriented metrics as a set of features [26], as shown in Table II. In total, 83,113 records and 24 columns, including the target column were obtained. A change in the target column (bug) name is referred to as a fault. This study focused on the class level, and the dataset consisted of the number of faults in each class for each project. Figure 1 shows the total number of classes related to the number of faults. For example, there were 70,000 classes without any faults. The number of software faults is centered around zero, indicating an imbalance in the dataset (the number of classes with no faults is greater than the number of classes having at least one fault).

TABLE I. DATASET

Project	Version	Number of classes	Number of faulty classes	Project	Version	Number of classes	Number of faulty classes	Project	Version	Number of classes	Number of faulty classes
Ant	1.3	125	20	**Lucene**	2.0	195	91	**Jedit**	3.2	272	90
	1.4	178	40		2.2	246	144		4.0	306	75
	1.5	293	32		2.4	340	203		4.1	312	79
	1.6	351	92	**Poi**	1.5	237	141		4.2	367	48
	1.7	745	166		2.0	314	37		4.3	492	11
Camel	1.2	609	217		3.0	442	281		3.2	272	90
	1.4	873	145	**Velocity**	1.4	196	147	**Prop**	1	15134	2439
	1.6	966	189		1.5	214	142		2	17065	2158
Xalan	2.4	724	111		1.6	229	78		3	7949	1048
	2.5	804	388	**Synapse**	1.0	157	16		4	6981	756
Xerces	1.3	453	69		1.1	222	60		5	6649	1178
	1.4	588	437		1.2	256	86		6	631	64
Ivy	1.1	111	63	**Log4j**	1.0	135	34				
	1.4	241	16		1.1	109	37				
	2.0	352	40		1.2	205	189				

TABLE II. OBJECT-ORIENTED SOFTWARE METRICS

Abbreviation	Name	Abbreviation	Name	Abbreviation	Name
WMC	Weighted methods per class	LOC	Lines of code	CE	Efferent couplings
DIT	Depth of inheritance tree	DAM	Data access metric	LCOM3	Lack of cohesion in methods
NOC	Number of children	MOA	Measure of aggregation	IC	Inheritance coupling
CBO	Coupling between object classes	MFA	Measure of functional abstraction	AMC	Average method complexity
RFC	Response for a class	CAM	Cohesion among methods of class	Max_cc	Max McCabe's cyclomatic complexity
LCOM	Lack of cohesion in methods	CBM	Coupling between methods	Avg_cc	Avg McCabe's cyclomatic complexity
CA	Afferent couplings	NPM	Number of public methods	FAULT	Number of faults

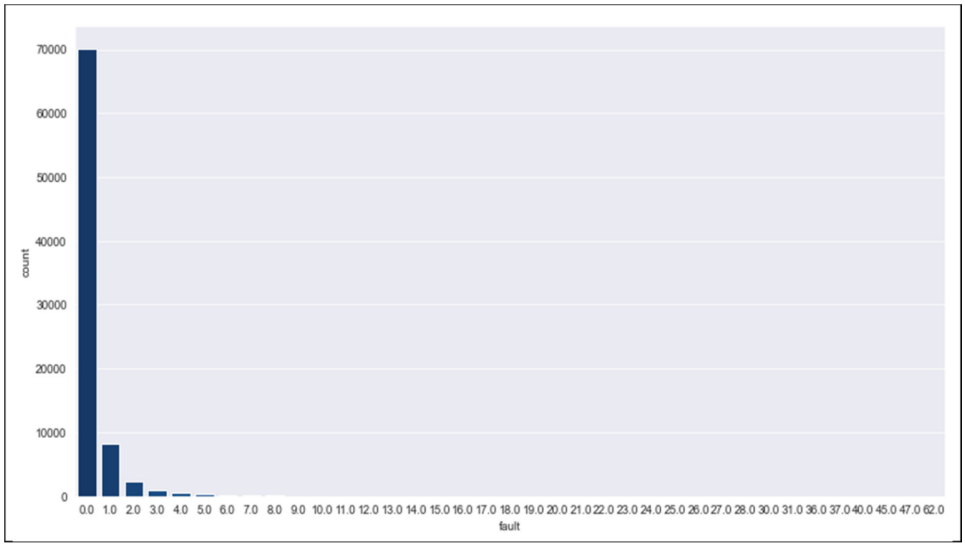


Fig. 1. Number of classes vs number of faults.

A. *Data Analysis*

During the analysis, it was evident that approximately 37,123 duplicate records were present. Consequently, the data were analyzed to determine the underlying relationships between the features and the dependent variable (fault). First, bivariate analysis was applied to identify possible relationships or distinct patterns between the two features. A commonly used technique for this purpose is the correlation matrix, which effectively detects the linear relationship (correlation) between two continuous features. Correlation helps identifying important features with respect to the dependent variable and checks for multiple linear relationships among the features. Figure 2 depicts the only the strongly correlated feature. However, there are only a few strong positive correlations. There was a strong positive correlation between the number of WMC and NPM, and the total number of methods for RFC and LCOM indicates that as the number of the former increased, the number of the

latter also increased within each class. From the first relationship it can be concluded that most of the methods in projects are of a public type. Additionally, there was a strong positive correlation among DIT, IC, and MFA, indicating that most of the methods were inherited. Additionally, RFC exhibited a positive correlation with LOC. There are a few linear relationships between the dependent variable (fault) and the independent variables (features). It was found out that RFC, LOC, and WMC have the strongest linear relationships with the number of faults. These features are important for predicting the number of faults.

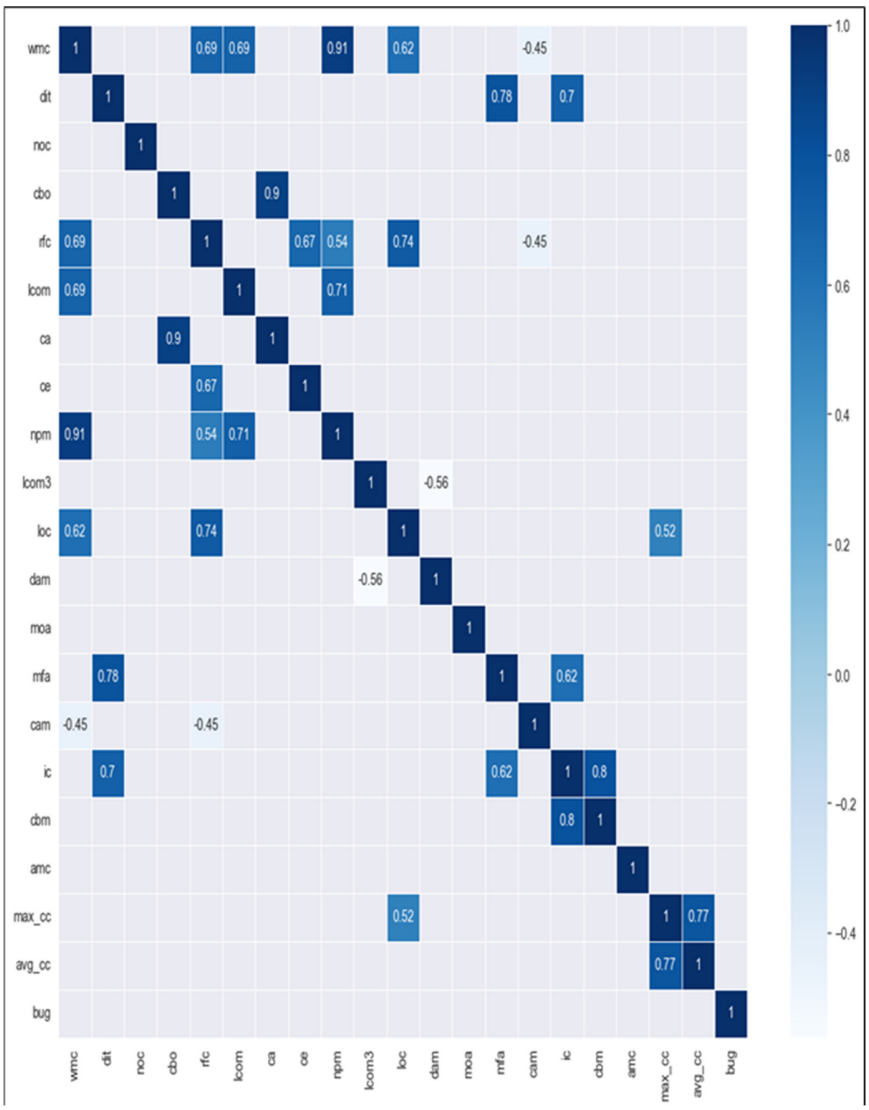


Fig. 2. Correlation matrix, only strongly correlated features.

B. Data Pre-Processing

Data are combined into one file to make them suitable for DL models. In this phase, duplicate records were deleted, resulting in a data size of 45,988. Subsequently, columns that do not affect the target column (fault) were removed, namely the program name (Name), version number (Version), and class name (Name), reducing the total number of columns to 21. The number of faulty records, 11,122, was way less than that of non-faulty records (34,867). Faulty records represented only 24%, while non-faulty records comprised 75% of the total data size. The SMOTEND method was applied to address this problem [11], which increases the number of minority records, specifically the number of faulty records in this case. Thus, the dataset size was 161,086 records after deleting the duplicate and missing records, which led to a 78% increase in classes containing faults. The distribution of the dependent variable (fault) in the original data following the application of the SMOTEND method is displayed in Figure 3. The distribution was improved slightly by increasing the number of faulty records shown, which balanced the data (Figure 4). Subsequently, natural logarithmic transformation and standardization were performed. A natural logarithm transformation was adopted to transform a highly skewed dataset distribution into a less skewed one [33].

A log transformation on the dependent variable (fault) was applied to improve the distribution of the dependent variable further, as portrayed in Figure 5. Standardization was then performed to enhance the feature distribution by eliminating the bias using (1):

$$z = \frac{(x-\mu)}{\sigma} \quad (1)$$

where x expresses a specific element (feature), μ is the mean of the feature, and σ is the standard deviation of the feature.

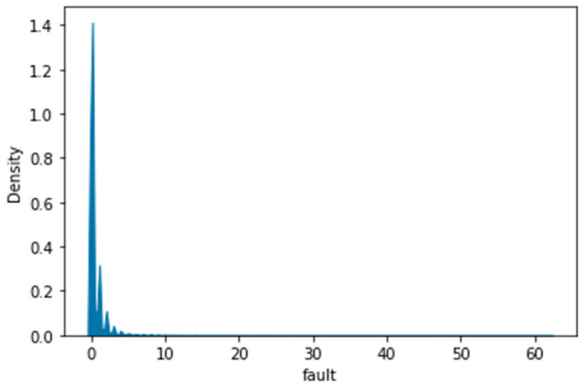


Fig. 3. Distribution of dependent variable, original data.

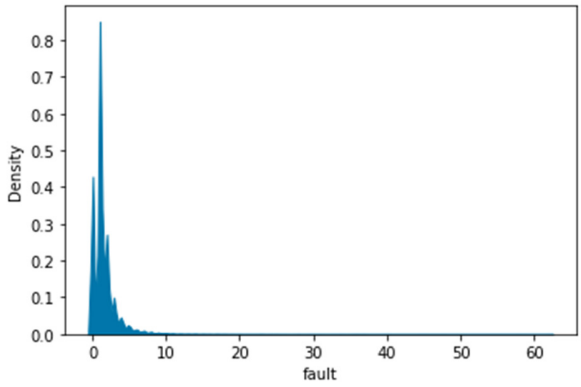


Fig. 4. Distribution of the dependent variable, after applying SMOTEND.

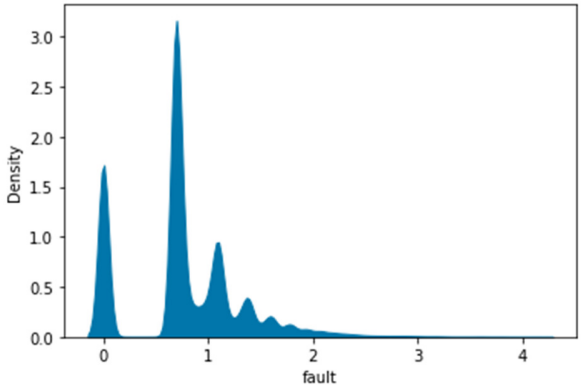


Fig. 5. Distribution of data with a log transformation.

IV. RESEARCH METHODOLOGY

Figure 6 presents the block diagram of the followed process. After collecting and pre-processing the data, the latter were divided into training and testing sets to train the model (70% training and 30% testing). Two DL models, CNN and MLP, were designed to predict faulty units. The results were evaluated using two performance measures, Mean Squared Error (MSE) and Kendall, and the performances of the two models were compared. Finally, the obtained results were compared with the two best ML fault predicting models, which are DTR and SVR.

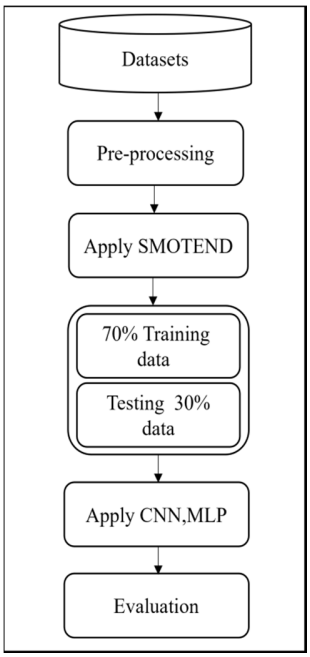


Fig. 6. The overall process.

A. Prediction Model

DL, a subset of ML, is based on ANNs [34]. The structure of an ANN is composed of multiple inputs, outputs, and hidden layers, which makes the learning structure deep. Each layer consists of neurons that process input data and pass the information to the next layer for a specific predictive task. The final output depends on the input data, the choice of activation function, and weight parameters [34]. CNN and MLP are alternative Deep Neural Network (DNN) architectures [34]. To improve prediction accuracy, CNNs were chosen to perform a new experiment based on the architecture developed in [22]. However, MLP has been effectively applied in the area of prediction [7]. Also, it has not been used for predicting the number of software faults.

1) Convolutional Neural Networks

A CNN is a DL technique that has attracted considerable attention for handling high-dimensional data. It engages a mathematical operation called convolution and incorporates a special type of NN consisting of feature extractors that determine weights during the training process [29]. The basic structure of a CNN is composed of three layers: a convolutional layer, a pooling layer, and a fully connected layer. In the

traditional approach, all the features are inserted into one CNN layer and then into the next layer, unlike the way we used.

In this paper, the CNN structure was applied, as described in [22], and subsequent modifications were made to improve the design. The difference was spotted in the division of the features. That is, they were divided according to dimensions, and encapsulation metrics, coupling metrics, abstraction metrics, cohesion metrics, while complexity metrics were considered. There was an effort for the same division to be utilized, but certain groups had a higher number of features than others. For example, the encapsulation metric group contained only one feature, namely the Data Access Metric (DAM), whereas the complexity metric group contained seven features. The resulting model did not learn and did not give satisfactory results.

As a next step, the features were divided randomly into five groups, all groups containing four features equally. Group one contained (WMC, DIT, NOC, CBO), group two (RFC, LCOM, CA, CE), group three (NPM, LCOM3, LOC, DAM), group four (MOA, MFA, CAM, IC), and group five (CBM, AMC, MAX_CC, AVG_CC). Each feature group was fed into a single CNN layer. The output of the CNN input layer becomes the input to a maxpooling layer. The maxpooling layer was added to merge similar features and reduce dimensionality [34]. Subsequently, dropout layers were added to reduce the possibility of overfitting [7]. Finally, flatten layers are added to convert the resulting matrix into an one-dimensional vector. These layers were repeated five times consecutively, which was equal to the number of feature groups, with a different dataset being entered each time. The flattened layers generated five outputs, which were then merged using the merge layer and fed into the first fully connected layer. The final layer is the second fully connected layer, which produced the predicted number of faults. Figure 7 illustrates the design structure of the CNN model.

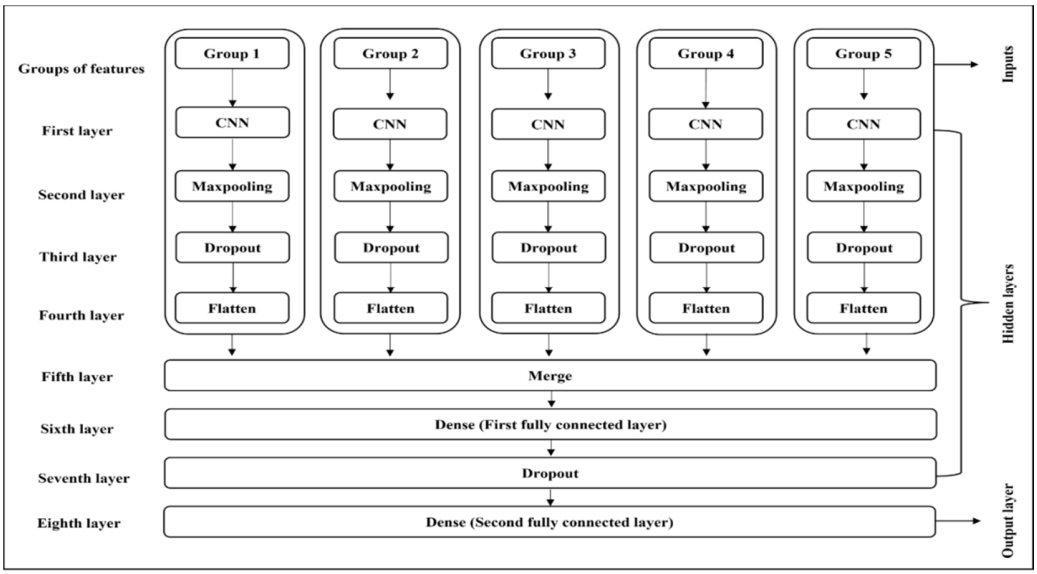


Fig. 7. The structure of the CNN model.

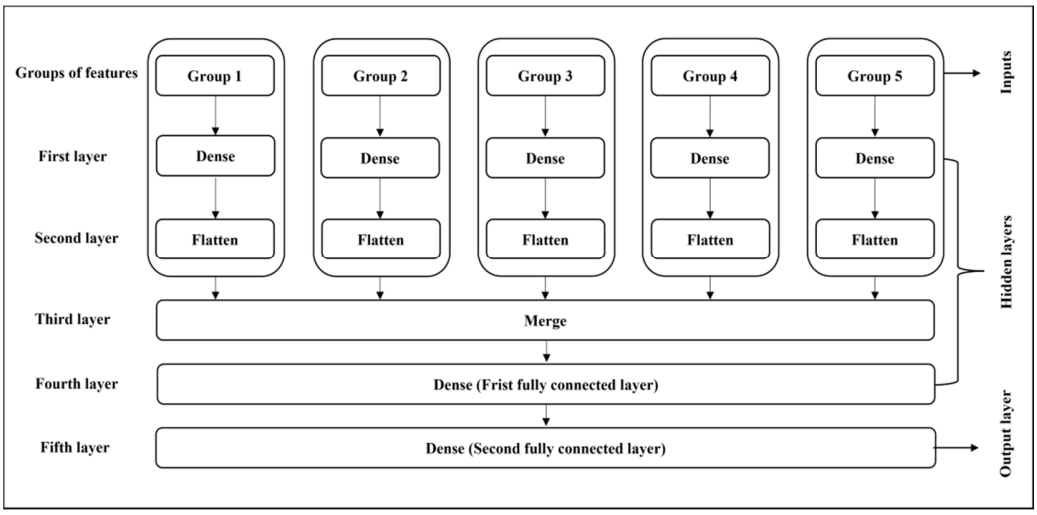


Fig. 8. The structure of the MLP model.

2) Multilayer Perceptron

MLPs are multi-hidden layer NNs composed of a minimum of three layers: an input layer, one or more hidden layers, and an output layer. The input features are weighted and simultaneously fed into the first hidden layer. This layer generates inputs for the next hidden layer, and this process continues until the last layer. The weighted output from the last hidden layer is fed into the output layer, enabling the network to make predictions [7]. In terms of implementation, the deep MLP is the simplest DL model [34]. As a preliminary experiment, this model was built similarly to the previous CNN model. The features were divided into five groups. A dense layer was inserted into each group, followed by a flat layer. Thus, five outputs were obtained. Following that, the merge layer collected the outputs and fed them into the first and second fully connected layers. Figure 8 shows the structure of the MLP model.

V. MODEL EVALUATION

Two different performance metrics were used to evaluate the prediction models: Kendall and MSE. Kendall provides the best performance measure for predicting the number of faults compared to other metrics [11]. On the other hand, the MSE is widely employed and is considered a measure of the quality of a prediction model, with a preference for a smaller value [33]. The Kendall rank correlation coefficient, known as Kendall, is a non-parametric statistic used to test the similarities in data ordering. It has been also used to measure the ordinal association between two measured quantities [11]. In this experiment, the ordinal correlation between the actual and predicted numbers of faults was determined. A higher Kendall coefficient value shows better performance. The Kendall coefficient is given by:

$$\tau = \frac{\#concordant\ pairs - \#discordant\ pairs}{n(n-1)/2} \quad (2)$$

MSE measures the squared difference between the predicted and the actual values [33]. The formula for calculating MSE is given by:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(|y_i - \hat{y})^2 \quad (3)$$

where $\hat{y}$ is the predicted value of y.

VI. RESULTS AND DISCUSSION

A. Experimental Settings

The models were developed in Google Colab using Python 3. An Asus PC with Windows operating system was utilized. The total number of records employed in the experiments was 161,086. Two experiments were conducted. First, the prediction models were evaluated before and after using the SMOTEND method. During model training, the epoch and batch size parameters were modified regularly to achieve the best results. The batch size varied between 10 and 20 and the number of epochs between 10 and 200, with the improvement stopping at 100. The CNN and MLP model parameters were defined as: activation = ReLU and kernel_initializer = glorot_uniform.

B. Experimental Results

The results of Experiments 1 and 2, evaluating both models before and after applying the SMOTEND method, are depicted in Table III. When the data were imbalanced, the MLP performed poorly compared to CNN in both the training and testing phases. However, the results demonstrated further improvement when SMOTEND was used to resolve the imbalance issue, as in the second experiment. In the testing set, the MLP outperformed the CNN, with an enhanced Kendall's value of 0.416 and MSE of 0.195 in the testing phase when utilizing MLP with balanced data, and this proves the effectiveness of MLP in predicting the number of faults. The results were not in accordance with those in [22], where CNN performed well. To ensure the integrity of the model's progression, we need to monitor the validation loss value. This value is adopted to check the performance after each complete epoch and determine if the model requires adjustments. The training and validation losses indicate how effectively the model was trained. The MSE was selected as the loss function due to its compatibility with the data type.

TABLE III. EXPERIMENTS TO TRAIN AND TEST THE IMPACT OF BALANCE AND IMBALANCE DATA WITH CNN AND MLP

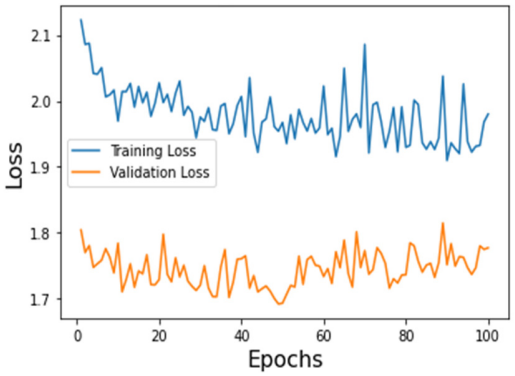
Experiment		**Experiment 1 (without SMOTEND)**		**Experiment 2 (with SMOTEND)**	
MODEL	**METRIC**	Kendall	MSE	Kendall	MSE
CNN	Train	0.19	1.776	0.361	0.222
	Test	0.162	1.316	0.363	0.218
MLP	Train	0.186	1.887	0.444	0.185
	Test	0.183	1.73	0.416	0.195

Figure 9 presents the training and validation losses for both models before processing the unbalanced data. It is noticed that the MLP did not learn well with the current data, thus, the SMOTEND method was applied to solve this issue. Figure 10 shows the optimized training and validation losses for both CNN and MLP. The best performing models are displayed in Figure 10(b), where a clear reduction in loss can be seen. In Figure 10(a), the number of batch size and epochs changed many times and no improvement was noticed.

The difference between CNN and MLP is that CNN contains layers that filter features, unlike MLP, which deals with the data directly without any intervention or filtering. Therefore, it was concluded that the data pre-processing is good enough. MLP performed better, contrary to the conclusions reached in [16-29]. The reason is the type of data used (numeric data) and the non-linearity of the problem.

While DL models employ big data to solve complex problems, they have been effective in addressing SFP problems for better predictions. To verify the effectiveness of the proposed DL models for SFP, two state-of-the-art ML models, SVR and DTR, were applied with the results illustrated in Table IV. These models were utilized due to their accuracy in predicting the number of faults in software modules [33]. The obtained results were compared with those of the implemented DL algorithms. The ML models were applied in the same two experiments using the same datasets and features.

(a)



(b)

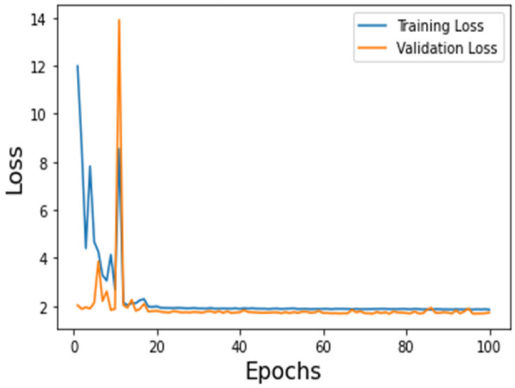
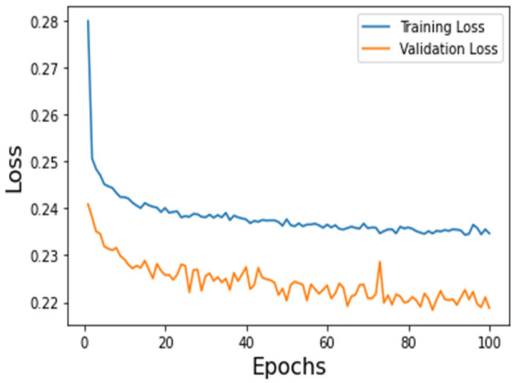


Fig. 9. (a) The CNN and (b) MLP without SMOTEND.

(a)



(b)

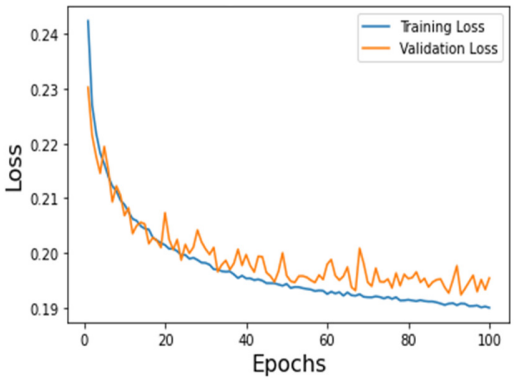


Fig. 10. (a) The CNN and (b) MLP witht SMOTEND.

As shown in Table IV, Experiment 1 demonstrated underfitting, and the model failed to learn with the original data. However, the improvement was noticeable as the data were refined and cleaned. The DTR model provided better results than the SVR model, which required a longer time to run and deliver results. Additionally, the SVR did not present the outcomes of Experiment 1 in all training and test groups. Even though DTR gave better results in Experiment 2, with Kendall's coefficient of 0.486 and MSE of 0.170, the model did not train well.

TABLE IV. PERFORMANCE OF THE ML MODELS IN THE TRAINING AND TEST SETS AND THE SAME EXPERIMENTS

Experiment		**Experiment 1 (without SMOTEND)**		**Experiment 2 (with SMOTEND)**	
MODEL	**METRIC**	Kendall	MSE	Kendall	MSE
DTR	Train	0.307	1.692	0.532	0.146
	Test	0.219	1.929	0.486	0.17
SVR	Train	-	-	0.279	0.255
	Test	-	-	0.276	0.257

Figure 11 depicts the training of the DTR model with balanced data. The red color represents the actual data, while the blue color represents the predicted data. Therefore, it cannot be said that DTR gave better performance than MLP, because even though the MLP results were slightly lower, the model learned well.

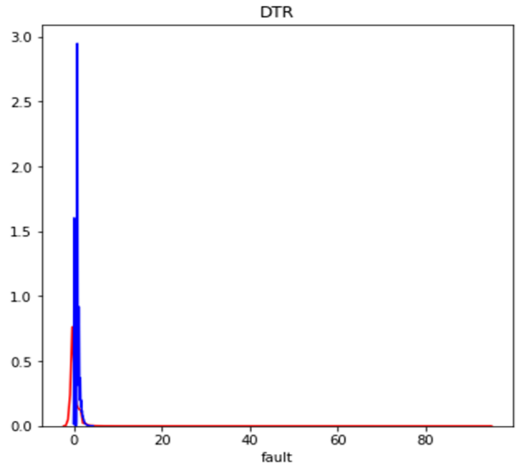


Fig. 11. DTR with SMOTEND.

VII. THREATS TO VALIDITY

This section discusses the external validity of our experiments. External validity relates to the generality of the results. To mitigate this threat, we used 12 publicly available datasets for training and evaluating the proposed model. However, all projects involved in our experiments were written in Java, raising the potential threat of generalizing the results to other programming languages. Furthermore, our chosen datasets posed other threats to external validity, as they consisted of 20 object-oriented metrics. However, other metrics yielded different results.

VIII. CONCLUSION AND FUTURE WORK

SFP has attracted significant research attention due to its ability to decrease the testing cost. While most existing studies focused on the classification of software units as faulty, predicting the number of faults in each software unit is also beneficial. In this study, two deep learning models, CNN and MLP, were designed to predict the number of faults in each software unit. Their performance was evaluated over 12 software project datasets. Furthermore, SMOTEND was applied to solve the data imbalance issue and improve prediction performance, which adversely affects result accuracy.

Based on the findings of this research, the MLP showed better results, with a Kendall value of 0.416 and MSE equal to 0.195, achieving a lower error rate than CNN. In addition, using SMOTEND improved the findings significantly. MLP was better in terms of the results and model performance. The main reason for this is that the problem is non-linear and the data type is numeric, making MLP suitable for facing this issue. Future work plans involve finding methods for obtaining data on fault types and their criticalities to prioritize faults. In addition, software semantics can be utilized to predict faults. Thus, this approach can be extended to techniques that consider code semantics while predicting faulty units.

REFERENCES

[1] A. Kumar and A. Bansal, "Software Fault Proneness Prediction Using Genetic Based Machine Learning Techniques," in *4th International Conference on Internet of Things: Smart Innovation and Usages*, Ghaziabad, India, Apr. 2019, pp. 1–5, https://doi.org/10.1109/IoT-SIU.2019.8777494.

[2] K. Punitha and B. Latha, "Validation of Medical Imaging Software Using Metaheuristic Knowledge Discovery," *Journal of Medical Imaging and Health Informatics*, vol. 6, no. 8, pp. 1966–1971, Dec. 2016, https://doi.org/10.1166/jmihi.2016.1958.

[3] C. Shyamala and S. A. Sahaaya Arul Mary, "Defect Prediction in Medical Software Using Hybrid Genetic Optimized Support Vector Machines," *Journal of Medical Imaging and Health Informatics*, vol. 6, no. 7, pp. 1600–1604, Nov. 2016, https://doi.org/10.1166/jmihi.2016.1857.

[4] J. Hryszko and L. Madeyski, "Cost Effectiveness of Software Defect Prediction in an Industrial Project," *Foundations of Computing and Decision Sciences*, vol. 43, no. 1, pp. 7–35, Mar. 2018.

[5] E. Elahi, A. Ayub, and I. Hussain, "Two staged data preprocessing ensemble model for software fault prediction," in *International Bhurban Conference on Applied Sciences and Technologies*, Islamabad, Pakistan, Jan. 2021, pp. 506–511, https://doi.org/10.1109/IBCAST51254.2021.9393182.

[6] C. L. Prabha and N. Shivakumar, "Software Defect Prediction Using Machine Learning Techniques," in *4th International Conference on Trends in Electronics and Informatics*, Tirunelveli, India, Jun. 2020, pp. 728–733, https://doi.org/10.1109/ICOEI48184.2020.9142909.

[7] O. A. Qasem and M. Akour, "Software Fault Prediction Using Deep Learning Algorithms," *International Journal of Open Source Software and Processes*, vol. 10, no. 4, pp. 1–19, Oct. 2019, https://doi.org/10.4018/IJOSSP.2019100101.

[8] S. Dhankhar, H. Rastogi, and M. Kakkar, "Software fault prediction performance in software engineering," in *2nd International Conference on Computing for Sustainable Global Development*, New Delhi, India, Mar. 2015, pp. 228–232.

[9] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, Feb. 2015, https://doi.org/10.1016/j.asoc.2014.11.023.

[10] G. P. Bhandari and R. Gupta, "Machine learning based software fault prediction utilizing source code metrics," in *3rd International Conference on Computing, Communication and Security*, Kathmandu, Nepal, Oct. 2018, pp. 40–45, https://doi.org/10.1109/CCCS.2018.8586805.

[11] X. Chen, D. Zhang, Y. Zhao, Z. Cui, and C. Ni, "Software defect number prediction: Unsupervised vs supervised methods," *Information and Software Technology*, vol. 106, pp. 161–181, Feb. 2019, https://doi.org/10.1016/j.infsof.2018.10.003.

[12] C. Pan, M. Lu, and B. Xu, "An Empirical Study on Software Defect Prediction Using CodeBERT Model," *Applied Sciences*, vol. 11, no. 11, Jan. 2021, Art. no. 4793, https://doi.org/10.3390/app11114793.

[13] M. Massoudi, N. K. Jain, and P. Bansal, "Software Defect Prediction using Dimensionality Reduction and Deep Learning," in *Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks*, Tirunelveli, India, Feb. 2021, pp. 884–893, https://doi.org/10.1109/ICICV50876.2021.9388622.

[14] S. S. Rathore and S. Kumar, "A Decision Tree Regression based Approach for the Number of Software Faults Prediction," *ACM SIGSOFT Software Engineering Notes*, vol. 41, no. 1, pp. 1–6, Oct. 2016, https://doi.org/10.1145/2853073.2853083.

[15] H. Alsawalqah, H. Faris, I. Aljarah, L. Alnemer, and N. Alhindawi, "Hybrid SMOTE-Ensemble Approach for Software Defect Prediction," in *6th Computer Science On-line Conference*, Prague, Czech Republic, Apr. 2017, pp. 355–366, https://doi.org/10.1007/978-3-319-57141-6_39.

[16] K. Wongpheng and P. Visutsak, "Software Defect Prediction using Convolutional Neural Network," in *35th International Technical Conference on Circuits/Systems, Computers and Communications*, Nagoya, Japan, Jul. 2020, pp. 240–243.

[17] A. Hasanpour, P. Farzi, A. Tehrani, and R. Akbari, "Software Defect Prediction Based On Deep Learning Models: Performance Study." arXiv, Apr. 02, 2020, https://doi.org/10.48550/arXiv.2004.02589.

[18] T. Liu, S. Fang, Y. Zhao, P. Wang, and J. Zhang, "Implementation of Training Convolutional Neural Networks." arXiv, Jun. 03, 2015, https://doi.org/10.48550/arXiv.1506.01195.

[19] L. B. Salah and F. Fourati, "Systems Modeling Using Deep Elman Neural Network," *Engineering, Technology & Applied Science Research*, vol. 9, no. 2, pp. 3881–3886, Apr. 2019, https://doi.org/10.48084/etasr.2455.

[20] S. Sahel, M. Alsahafi, M. Alghamdi, and T. Alsubait, "Logo Detection Using Deep Learning with Pretrained CNN Models," *Engineering, Technology & Applied Science Research*, vol. 11, no. 1, pp. 6724–6729, Feb. 2021, https://doi.org/10.48084/etasr.3919.

[21] E. E. Miandoab and F. S. Gharehchopogh, "A Novel Hybrid Algorithm for Software Cost Estimation Based on Cuckoo Optimization and K-Nearest Neighbors Algorithms," *Engineering, Technology & Applied Science Research*, vol. 6, no. 3, pp. 1018–1022, Jun. 2016, https://doi.org/10.48084/etasr.701.

[22] L. Qiao, G. Li, D. Yu, and H. Liu, "Deep Feature Learning to Quantitative Prediction of Software Defects," in *45th Annual Computers, Software, and Applications Conference*, Madrid, Spain, Jul. 2021, pp. 1401–1402, https://doi.org/10.1109/COMPSAC51774.2021.00204.

[23] R. Jothi, "A Comparative Study of Unsupervised Learning Algorithms for Software Fault Prediction," in *Second International Conference on Intelligent Computing and Control Systems*, Madurai, India, Jun. 2018, pp. 741–745, https://doi.org/10.1109/ICCONS.2018.8663154.

[24] H. Wang and T. M. Khoshgoftaar, "A Study on Software Metric Selection for Software Fault Prediction," in *18th IEEE International Conference On Machine Learning And Applications*, Boca Raton, FL, USA, Dec. 2019, pp. 1045–1050, https://doi.org/10.1109/ICMLA.2019.00176.

[25] S. S. Rathore and S. Kumar, "An empirical study of ensemble techniques for software fault prediction," *Applied Intelligence*, vol. 51, no. 6, pp. 3615–3644, Jun. 2021, https://doi.org/10.1007/s10489-020-01935-6.

[26] S. S. Rathore and S. Kumar, "An empirical study of some software fault prediction techniques for the number of faults prediction," *Soft Computing*, vol. 21, no. 24, pp. 7417–7434, Dec. 2017, https://doi.org/10.1007/s00500-016-2284-x.

[27] S. S. Rathore and S. Kuamr, "Comparative analysis of neural network and genetic programming for number of software faults prediction," in *National Conference on Recent Advances in Electronics & Computer*

Engineering, Roorkee, India, Feb. 2015, pp. 328–332, https://doi.org/10.1109/RAECE.2015.7510216.

[28] A. Agrawal and T. Menzies, "Is 'better data' better than 'better data miners'? on the benefits of tuning SMOTE for defect prediction," in *40th International Conference on Software Engineering*, Gothenburg, Sweden, Jun. 2018, pp. 1050–1061, https://doi.org/10.1145/3180155.3180197.

[29] G. P. Bhandari and R. Gupta, "Measuring the Fault Predictability of Software using Deep Learning Techniques with Software Metrics," in *5th IEEE Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering*, Gorakhpur, India, Nov. 2018, pp. 1–6, https://doi.org/10.1109/UPCON.2018.8597154.

[30] I. Batool and T. A. Khan, "Software fault prediction using deep learning techniques," *Software Quality Journal*, vol. 31, no. 4, pp. 1241–1280, Dec. 2023, https://doi.org/10.1007/s11219-023-09642-4.

[31] E. Borandag, "Software Fault Prediction Using an RNN-Based Deep Learning Approach and Ensemble Machine Learning Techniques," *Applied Sciences*, vol. 13, no. 3, Jan. 2023, Art. no. 1639, https://doi.org/10.3390/app13031639.

[32] M. Jureczko, "Significance of Different Software Metrics in Defect Prediction," *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 86–95, 2011.

[33] L. Qiao, X. Li, Q. Umer, and P. Guo, "Deep learning based software defect prediction," *Neurocomputing*, vol. 385, pp. 100–110, Apr. 2020, https://doi.org/10.1016/j.neucom.2019.11.067.

J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *Journal of Big Data*, vol. 6, no. 1, Mar. 2019, Art. no. 27, https://doi.org/10.1186/s40537-019-0192-5.