Annals of Emerging Technologies in Computing (AETiC)
Vol. 8, No. 1, 2024

Research Article

Explainable Software Defects Classification Using SMOTE and Machine Learning

Agboeze Jude and Jia Uddin*

Woosong University, Daejeon, South Korea
202280097@live.wsu.ac.kr; jia.uddin@wsu.ac.kr
*Correspondence: jia.uddin@wsu.ac.kr

Received: 5th July 2023; Accepted: 26th December 2023; Published: 1st January 2024

Abstract: Software defect prediction is a critical task in software engineering that aims to identify and mitigate potential defects in software systems. In recent years, numerous techniques and approaches have been developed to improve the accuracy and efficiency of the defect prediction model. In this research paper, we proposed a comprehensive approach that addresses class imbalance by utilizing stratified splitting, explainable AI techniques, and a hybrid machine learning algorithm. To mitigate the impact of class imbalance, we employed stratified splitting during the training and evaluation phases. This method ensures that the class distribution is maintained in both the training and testing sets, enabling the model to learn from and generalize to the minority class examples effectively. Furthermore, we leveraged explainable AI methods, Lime and Shap, to enhance interpretability in the machine learning models. To improve prediction accuracy, we propose a hybrid machine learning algorithm that combines the strength of multiple models. This hybridization allows us to exploit the strength of each model, resulting in improved overall performance. The experiment is evaluated using the NASA-MD datasets. The result revealed that handling the class imbalanced data using stratify splitting approach achieves a better overall performance than the SMOTE approach in SDD (Software Defect Detection).

Keywords: ***Classification; Detection; Explainable AI; Machine learning; Software Defect***

1. Introduction

Software defect prediction plays a vital role in the software development life cycle, aiming to identify and mitigate potential defects early in the process. In recent years, the increasing complexity of software systems and the demand for higher reliability have led researchers and practitioners to explore the application of machine learning techniques to enhance defect prediction accuracy. Software defects can be expensive and time-consuming to repair, causing substantial delays in software development initiatives. Researchers and practitioners have been exploring different methods for predicting and avoiding software defects recently [1]. Machine learning algorithms have gained popularity in software defect prediction. Techniques such as decision trees and random forests have been applied to classify software modules as defective or non-defective based on historical data. These algorithms learn from past defect data and extract relevant features to make predictions [2]. In recent years, research efforts have been focused on enhancing the accuracy and applicability of defect prediction models [3]. Feature selection techniques have been introduced to identify the most informative and relevant software metrics, reducing the dimensionality of prediction models and improving their interpretability [4]. Data pre-processing techniques, such as

Agboeze Jude and Jia Uddin, "Explainable Software Defects Classification Using SMOTE and Machine Learning", *Annals of Emerging Technologies in Computing (AETiC)*, Print ISSN: 2516-0281, Online ISSN: 2516-029X, pp. 35-49, Vol. 8, No. 1, 1st January 2024, Published by International Association for Educators and Researchers (IAER), DOI: 10.33166/AETiC.2024.01.00, Available: http://aetic.theiaer.org/archive/v8/v8n1/p4.html.

handling class imbalance and addressing the missing data have been utilized to ensure the reliability of the prediction process.

1.1. Literature Review

In Menzies *et al.* proposed a data mining approach for predicting defects using static code attributes. Their study showed promising results in accurately identifying defect-prone modules. However, the study focused on small-scale systems, limiting its generalization to larger and more complex software projects [5]. We explored the application of support vector machines (SVM) for defect prediction in open-source software. Their findings demonstrated the effectiveness of SVM in accurately classifying defective and non-defective modules [6]. However, the study did not consider the impact of software metrics selection, which could potentially affect the prediction performance. In [7], Nagappan *et al.* investigated the relationship between code churn and software defects. The study did not consider other relevant software metrics, limiting the comprehensiveness of the prediction model. Using evolution algorithms, in [8], Turhan *et al.* developed a prediction model to estimate the likelihood of defects in software modules. While their study provided insights into the application of evolutionary algorithms for defect prediction, the model's performance was highly dependent on the chosen evolutionary algorithm parameters. In [9], Yang *et al*. proposed an ensemble approach for software defect prediction combining multiple classifiers to improve prediction accuracy. Although their approach achieved better results compared to individual classifiers, the study did not thoroughly analyze the impact of different ensemble configurations. Inspired by the concept of transfer learning, in [10], Tantithamthavorn *et al.* proposed a defect prediction model that leverages knowledge from related projects to improve prediction performance. However, the study did not thoroughly investigate the transferability in diverse software projects. In [11], introduced a hybrid model that combines learning and rule-based approaches for defect prediction. While their study showcased the benefits of combining different techniques, the model's performance heavily relied on the quality and effectiveness of the predefined rules. In [12], Umer *et al.* proposed a deep-learning-based approach for defect prediction using code change history and static code attributes. The study demonstrated the potential of deep learning techniques in capturing complex patterns. However, the model's performance may be affected by the availability and quality of historical data. In [13], Amir *et al.* explored the effectiveness of software metrics extracted from both source code and issue-tracking systems for defect prediction. The findings revealed that integrating metrics from multiple sources improved the prediction and accuracy. However, the study did not investigate the impact of different weighting schemes for combining the metrics. In [14], proposed a hybrid feature selection approach based on genetic algorithms and principal component analysis for software defect prediction. While their approach showed promise in reducing the dimensionality of the feature space, the study did not evaluate the impact of different parameter settings on the performance of the feature selection algorithm. Leveraging ensemble learning, in [15], Tarunim *et al.* developed a prediction model that combines multiple machine learning algorithms for defect prediction. The study demonstrated improved performance compared to individual algorithms. However, the model's performance may vary depending on the choice and configuration of the ensemble methods. In [16], Hale *et al.* investigated the use of time series analysis techniques for defect prediction, considering the temporal nature of software metrics. While their study provided insights into the potential benefits of time series analysis, the applicability of the approach may be limited to projects with sufficient historical data.

In [17], Mehta *et al.* compared the performance of different machine learning models for software defect prediction under different data imbalance conditions. They used four machine learning models: logistic regression, decision tree, random forest, and support vector machine. They also use three data imbalance conditions: 1:1, 1:10, and 1:100. Their results show that the models are more sensitive to the class imbalance problem when the number of defect-prone instances is small. For example, the accuracy of the

logistic regression model decreases from 80% to 60% when the data imbalance condition changes from 1:1 to 1:100.

Table 1. Contributions and limitations of different studies in the literature

Ref. Contributions	Limitations
[20] a data mining approach for predicting defects using static code attributes was explored; they showed promising results in accurately identifying defect-prone modules.	This study was limited to only small-scale systems. It was not exposed to larger and more complex software.
[21] The author introduced support vector machine (SVM) for defect prediction purposes in an open-source software perform a great effect in accurately classifying defective and non-defective modules.	However, the study did not consider the impact of software metrics selection, which could potentially affect the prediction performance
[22] In this state-of-the-art an investigation was conducted to check relationship between code churn and software defects.	There was lack of software metrics, thereby limiting the comprehension of the predictive model.
[23] proposed an ensemble approach for software defect prediction combining multiple classifiers to improve prediction accuracy, approach achieved better results compared to individual classifiers.	The study did not thoroughly analyze the impact of different ensemble configurations.
[24] By concept of transfer learning, proposed a defect prediction model that leverages knowledge from related projects to improve prediction performance.	The study did not thoroughly investigate the transferability in diverse software projects.
[25] In this study, a hybrid model was explored to combine learning and rule-based approaches for defect prediction. While their study showcased the benefits of combining different techniques.	The model's limitations, the performance was heavily relied on the quality and effectiveness of the predefined rules.
[26] proposed a deep-learning-based approach for defect prediction using code change history and static code attributes	The model's performance may be affected as a result the kind of quality of historical data.
[27] The author explores software metrics extracted from both source code and issue-tracking systems for defect prediction. The findings revealed that integrating metrics from multiple sources improved the prediction and accuracy.	The study did not investigate the impact of different weighting schemes for combining the metrics.
[30] Through a comprehensive analysis with non-transformation, time series forecasting method and conventional SGRMs, it has been shown that linear regression with Box-Cox (L_Box- Cox_T) could work well to predict the software fault in short time prediction.	The limitation encountered from the study is SGRM's can only be efficient when there is an increase in the input testing days for software fault prediction.
[31] The authors applied explainable AI techniques to analyze the machine learning models.	The limitations of the paper are they did not evaluate the results with explainable AI extracted features.

In [18], Sanchita *et al*. discussed the different techniques that have been proposed to address the class imbalance problem, including sampling, cost-sensitive learning, and ensemble learning. Sampling techniques involve changing the distribution of the data to make it more balanced. Cost-sensitive learning techniques assign different costs to misclassifying different types of instances. Ensemble learning techniques combine the predictions of multiple models to improve the overall performance. in [19], Alsaeedi *et al.* investigated the use of oversampling to address the class imbalance problem in software defect prediction. The paper utilised deep learning and two imbalance data condition 1:1 and 1:10. The

result shows that oversampling can improve the performance of the deep learning model for software defect prediction. For example, the accuracy of the model increases from 70% to 80% when the data imbalanced conditions change from 1:1 to 1:10. The authors also find that oversampling is more effective than under-sampling for improving the performance of the deep learning model.

In this paper, we focused on addressing the class imbalance in machine learning in software defect predication tasks thereby employing stratified splitting, explainable AI techniques, and hybrid machine learning algorithms, we advantageously attenuate the impart of class imbalance, thereby enhanced interpretability, and improved prediction accuracy. The stratified splitting during training and evaluation ensures that the class allocation is maintained in both the training and testing sets, enabling the model to learn from generalize to the minority class precedent successful. To enhance interpretability in the machine learning models in software defect prediction, we employ explainable AI methods like Lime and Shap. In which lime focused on local interpretability by inducing an explanation for individual predictions, allowing us to understand the reasoning behind the model's decisions., explaining individual predictions, while Shap provide both local and global interpretability and attributed the contribution of each feature, providing productive insights into the value and impart of different features on the model predictions. The hybrid allows to explore the strength of each model, thereby resulting in the overall performance and various and relationships in the data, enhanced predictive power.

1.2. Contributions

In this paper, we focused on the contributions of the proposed method are summarized as follows:

1. It employed a stratified splitting method to address class imbalance during the training and evaluation phase of a machine learning model. This technique ensured that the class distribution was maintained in both the training and testing sets, mitigating the impact of class imbalance on model performance.
2. Leveraging explainable AI techniques (Lime and Shap) to provide interpretability to the machine learning model. Lime allowed for local interpretability by generating explanations for individual predictions, while Shap utilized game theory concepts to attribute the contribution of each feature towards the model's output, enhancing transparency and trust in the decision-making process.
3. Five machine learning models were applied to the software fault dataset to evaluate the performance. Details results of the machine learning models are reported in this draft along with Shap and Lime explainable AI.

The remaining is organized as follows. Section II discusses related work briefly. Section III discusses the methodology. Section IV discusses the result and analysis. Section V discusses the conclusion.

2. Methodology

The first step in the methodology involves collecting the relevant data which is the foundational step in defect prediction. To identify software defects, we need historical data that includes information about past defects, code changes, and other relevant factors. Once the data is collected, the second step is to preprocess the data which involves removing noise from the data, such as missing values, duplicates, standardizing the format, etc. The need for data preprocessing is essential for cleaning and preparing the data for analysis to enhance the reliability and integrity of the dataset, reducing noise and inconsistencies that can impact the model performance [28]. Feature selection technique comes as the step to identify the most relevant features suitable for SDD. Selecting informative features can lead to a more interpretable model, reducing complexity, shorter training times, and enhancing the model's prediction power. The next step is to develop a suitable model using the final preprocessed data for SDD. After the model development, the final step is to evaluate and validate the model performance with metrics such as

precision, recall, f1 score, accuracy, and ROC curve. The evaluation step also incorporates explainable AI techniques such as Lime and SHAP to provide interpretability and transparency to the model's prediction output [29]. The importance of model evaluation and validation lies in ensuring that the developed model is reliable, effective, and capable of accurately detecting defective software.

Figure 1 demonstrates the experiment design that aims to investigate the comparative analysis of the different ways of modeling the class imbalance dataset fitted to several machine learning algorithms to identify the method that will yield the best performance in SDD.

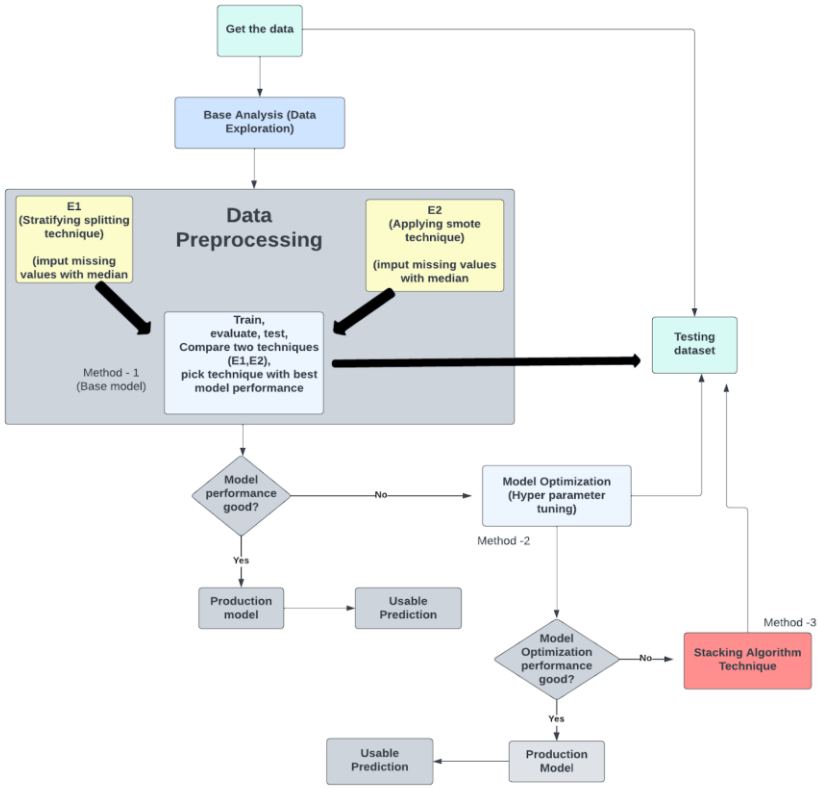

Figure 1. Experiment Design

The five ML algorithms (XG, RF, GNB, LGBM, LR) were trained on each of the data models E1 and E2 as the base performance model in a train-test-split of 90:10, where 90% of the data was used for training and 10% was used for evaluating the model performance. E1 represents the experiment that treated the class imbalance by applying stratify splitting method and E2 represents the experiment that treated the imbalance dataset by applying SMOTE method to oversample the minority class. The two best-performing model algorithms in each data model were further optimized through hyperparameter tuning using GridSearch CV to improve the model performance and compare across the different data models. A third method was also introduced to further improve the model's overall performance. Table 1 summarises the three methods and the different algorithms used in each step for better analysis.

Table 2. Algorithms used in each method

Method 1	**Base Models**
E1	XG, RF, LGBM, LR, GNB
E1	XG, RF, LGBM, LR, GNB
Method 2	Hyperparameter tuning(GridSearch)
E1	XG, RF
E2	XG, RF
Method 3	Hybrid Algorithm
E1	XG, RF, LGBM, LR, DT
E2	XG, RF, LGBM, LR, DT

2.1. Dataset Description

The data for this study was downloaded from Figshare [20], which is a zipped file containing two file directories D` and D``. In each directory contains 13 files namely: CM1, JM1, KC1, KC3, KC4, MC1, MC2, MW1, PC1, PC3, and PC4 with all provided in arff format. I also observed that KC4 has no record in both directories and JM1 in the record had the dependent feature "Defective" entered in as "Label". As part of my preprocessing step, I had the KC4 file removed and renamed the dependent feature in the JM1 record to the appropriate value 'Defective". The dataset contains a total of 61147 records combined, with 40 features (39 numeric features and one categorical feature). The description of the dataset is provided in Figure 2.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 61147 entries, 0 to 17376
Data columns (total 40 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   LOC_BLANK                       58817 non-null  float64
 1   BRANCH_COUNT                    61147 non-null  int64
 2   CALL_PAIRS                      40493 non-null  float64
 3   LOC_CODE_AND_COMMENT            61147 non-null  int64
 4   LOC_COMMENTS                    61147 non-null  int64
 5   CONDITION_COUNT                 40493 non-null  float64
 6   CYCLOMATIC_COMPLEXITY           61147 non-null  int64
 7   CYCLOMATIC_DENSITY              40493 non-null  float64
 8   DECISION_COUNT                  40493 non-null  float64
 9   DECISION_DENSITY                10516 non-null  float64
 10  DESIGN_COMPLEXITY               61147 non-null  int64
 11  DESIGN_DENSITY                  40493 non-null  float64
 12  EDGE_COUNT                      40493 non-null  float64
 13  ESSENTIAL_COMPLEXITY            61147 non-null  int64
 14  ESSENTIAL_DENSITY               40493 non-null  float64
 15  LOC_EXECUTABLE                  61147 non-null  int64
 16  PARAMETER_COUNT                 40493 non-null  float64
 17  HALSTEAD_CONTENT                61147 non-null  float64
 18  HALSTEAD_DIFFICULTY             61147 non-null  float64
 19  HALSTEAD_EFFORT                 61147 non-null  float64
 20  HALSTEAD_ERROR_EST              61147 non-null  float64
 21  HALSTEAD_LENGTH                 61147 non-null  int64
 22  HALSTEAD_LEVEL                  61147 non-null  float64
 23  HALSTEAD_PROG_TIME              61147 non-null  float64
 24  HALSTEAD_VOLUME                 61147 non-null  float64
 25  MAINTENANCE_SEVERITY            40493 non-null  float64
 26  MODIFIED_CONDITION_COUNT        40493 non-null  float64
 27  MULTIPLE_CONDITION_COUNT        40493 non-null  float64
 28  NODE_COUNT                      40493 non-null  float64
 29  NORMALIZED_CYLOMATIC_COMPLEXITY 40493 non-null  float64
 30  NUM_OPERANDS                    61147 non-null  int64
 31  NUM_OPERATORS                   61147 non-null  int64
 32  NUM_UNIQUE_OPERANDS             61147 non-null  int64
 33  NUM_UNIQUE_OPERATORS            61147 non-null  int64
 34  NUMBER_OF_LINES                 40493 non-null  float64
 35  PERCENT_COMMENTS                40493 non-null  float64
 36  LOC_TOTAL                       61147 non-null  int64
 37  Defective                       61147 non-null  object
 38  GLOBAL_DATA_COMPLEXITY          30623 non-null  float64
 39  GLOBAL_DATA_DENSITY             30623 non-null  float64
dtypes: float64(26), int64(13), object(1)
```

Figure 2. Dataset features and their datatype

2.2. Feature Selection and Engineering

In this section, the following methods were adapted for feature selection and engineering:

2.2.1. label Encoding

The data was transformed by converting the categorical feature to a numeric feature before developing the model. This is a standard practice to always convert all categorical features to numeric since machine learning algorithms cannot abstract labels but rather numeric values.

2.2.2. Feature Scaling

To normalize the features, the MinMax scaler was utilized. The choice to use the MinMax scaler here over the Standard scaler was because the dataset was heavily skewed i.e. they are not normally distributed (not perfectly Gaussian)

2.2.3. Correlation

This is a way to determine the correlated features between the dependent and independent features. In this case, these features were identified using a heatmap.

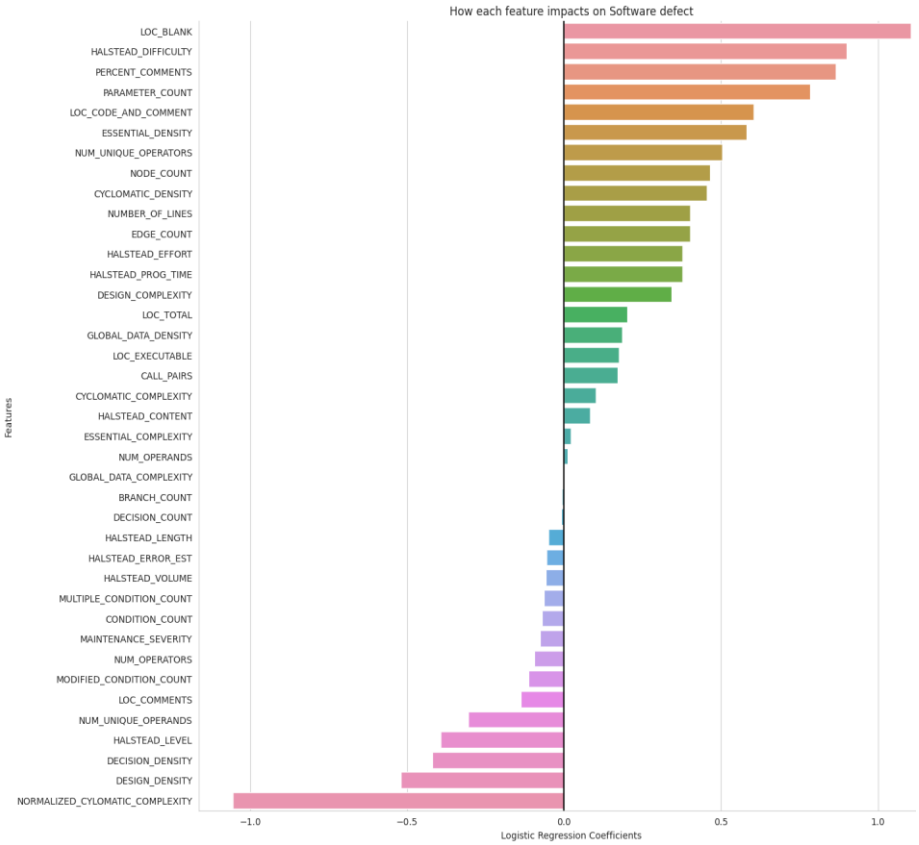

Figure 3. Dataset features and their level of impact on SDP

2.2.4. Feature Selection with Logistic Regression Technique

This method utilized interpretability fitted to logistic regression to sort the features according to how they impact positively or negatively on the dependent variable in ascending order as seen in Figure 3. This is to understand the features the model is basing its predictions on, especially since a logistic regression classifier is easy to decode using the coefficients it has assigned, in this case, to each feature. This enabled the selection of only the significant features that have a high tendency to contribute to the model's performance, unlike "GLOBAL_DATA_COMPLEXITY", "BRANC_COUNT", and "DECISION_COUNT" features which were dropped during model training as they had very little or no impact.

2.3. Machine Learning Algorithms

For this study, seven (7) machine learning algorithms were developed for analysis and comparison, which are presented in the following sub-sections.

2.3.1. Random Forest

Random Forest is a popular machine-learning algorithm used for both regression and classification tasks. It is an ensemble learning method that combines multiple decision trees to make a prediction. In the random forest model, each tree is constructed by randomly selecting a subset of the features and a subset of the training data. They are mostly known for their high accuracy, ability to handle large datasets and resistance to overfitting.

2.3.2. Extreme Gradient Boosting Algorithm

Extreme Gradient Boosting is a popular machine learning algorithm that belongs to the family of gradient boosting methods. It is designed to boost the performance of tree-based models by iteratively learning from the errors of previous models. XGBoost is a powerful algorithm that can be used for a wide range of applications and provides robust and accurate predictions with feature importance measures. Its ability to handle missing values and its regularized approach makes it a popular choice for many machine-learning tasks.

2.3.3. Light Gradient Boosting Machine (LightGBM)

LightGBM is a popular machine learning algorithm that belongs to the family of gradient boosting methods. It is designed to boost the performance of tree-based models by improving the efficiency and accuracy of existing algorithms, such as XGBoost performance of tree-based models by improving the efficiency and accuracy of existing algorithms, such as XGBoost. LightGBM is a powerful algorithm that can be used for a wide range of applications and provides robust and accurate predictions with high efficiency. Its leaf-wise tree growth and histogram-based algorithm make it a highly efficient algorithm, while its support for categorical features and early stopping make it a flexible and powerful tool for many machine-learning tasks. Leaf-wise tree growth and histogram-based algorithm make it a highly efficient algorithm, while its support for categorical features and early stopping make it a flexible and powerful tool for many machine-learning tasks.

2.3.4. Gaussian Naive Bayes (GNB)

Gaussian Naive Bayes is a simple and widely used probabilistic algorithm for classification tasks. It is based on the Bayes theorem and the assumption that the features are conditionally independent given the class. It is a simple yet effective algorithm for classification tasks, especially when the number of features is large compared to the number of samples. Its probabilistic nature, fast training, and easy implementation make it a popular choice for many machine-learning applications. However, it may not perform well when the independence assumption does not hold or when the data has a non-Gaussian distribution.

2.3.5. Logistic Regression

This is a commonly used algorithm for binary classification tasks, which involves predicting one of two possible outcomes. It uses a logistic function to model the probability of a binary outcome as a function of one or more predictor variables. Logistic Regression is a simple yet effective algorithm for binary classification tasks, especially when the relationship between the input features and the outcome is linear or can be approximated by a linear function. Its probabilistic nature, fast training, and easy interpretation make it a popular choice for many machine-learning applications. However, it may not perform well when the relationship between the input features and the outcomes is non-linear or when the input features are highly correlated.

2.3.6. Decision Tree

A Decision Tree is a popular algorithm for classification and regression tasks. It uses a tree-like model to represent decisions and their possible consequences. Each internal node of the tree represents a test on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label or a numerical value. Decision Tree is a versatile and powerful algorithm for both classification and regression tasks, especially when the underlying relationship between the input features and the output is non-linear or complex. Its non-parametric nature, easy interpretation, and feature selection capability make it a popular choice for many machine learning applications. However, it may not perform well when the data is noisy or when the tree structure becomes too complex. These concepts and calculations are essential for understanding and building decision tree classifiers. The following few equations provide the foundation for the construction and evaluation of decision trees.

2.3.7. Stacking Algorithm

Stacking is an ensemble learning technique that combines multiple base models to improve the predictive performance of the model. It involves training several base models on the training set and then using their predictions as input to a meta-model that combines them to make the final prediction. This is a robust technique that can handle outliers and missing data by using the predictions of the base models. Stacking is a powerful and flexible technique for improving the predictive performance of machine learning models, especially when dealing with complex and noisy data. Its ability to combine different types of models and leverage their strengths makes it a popular choice for many machine-learning applications. However, it may require more computational resources and training time compared to single models and may be prone to overfitting if not carefully designed and validated.

3. Experimental Result and Analysis

For this study, the following metrics were used to evaluate the performance of the models: accuracy (5), recall (6), precision (7), AUC which shows the tradeoff between TP rate and FP rate, and f-measure (8) [22]. The metric values were all computed using the statistical values of True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN).

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}} \quad (1)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (2)$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (3)$$

$$\text{Fmeasure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

XAI Shap and Lime were applied to each algorithm to analyze the feature impact in the model output and model interpretability respectively. In Figure 5 and Figure 6, it is observed that random forest and XGboost used all the features for prediction. This explained why they both came out as the best-performing base models. It can be observed in Figures 5(a)-(c), that the red and blue colours occupy half of the horizontal rectangles for each class. This means each feature has an equal impact on the classification of defective software. Also, the upper features indicate features with the most predicting power while the lower features do not contribute as much.

Table 3. Showing the base model results evaluated on E1 data model

Name	Precision	Accuracy	F1_score	Recall	Auc	Train	Test
RF	0.97	0.98	0.94	0.91	0.95	0.10	0.99
LR	0.51	0.89	0.10	0.06	0.52	0.89	0.89
LGBM	0.81	0.92	0.35	0.22	0.61	0.93	0.92
XGB	0.89	0.93	0.54	0.38	0.69	0.95	0.93
NB	0.37	0.88	0.26	0.21	0.58	0.88	0.88

Table 4. Showing base model results evaluated on E2 data model

Name	Precision	Accuracy	F1_score	Recall	Auc	Train	Test
RF	0.904	0.98	0.91	0.91	0.95	0.10	0.98
LR	0.24	0.71	0.36	0.80	0.75	0.74	0.71
LGBM	0.39	0.86	0.48	0.63	0.76	0.90	0.86
XGB	0.50	0.90	0.57	0.65	0.79	0.95	0.90
NB	0.37	0.88	0.26	0.21	0.58	0.58	0.88

Table 5. Stack algorithm results on E1 and E2 data model

Data-model	Accuracy	Auc	F1_score	Recall	Train	Test
E1	0.990	0.985	0.953	0.979	0.991	0.990
E2	0.981	0.948	0.909	0.905	0.993	0.982

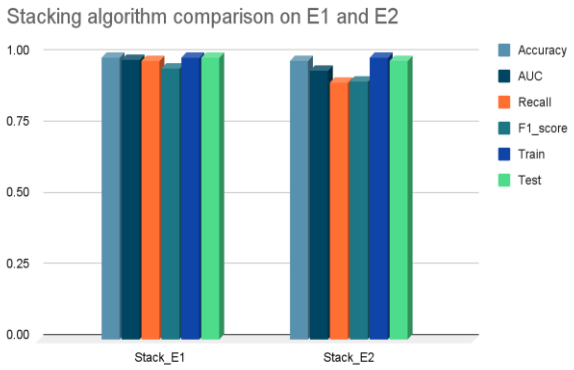

Figure 4. Hybrid algorithm model results in comparison of E1 and E2 data model

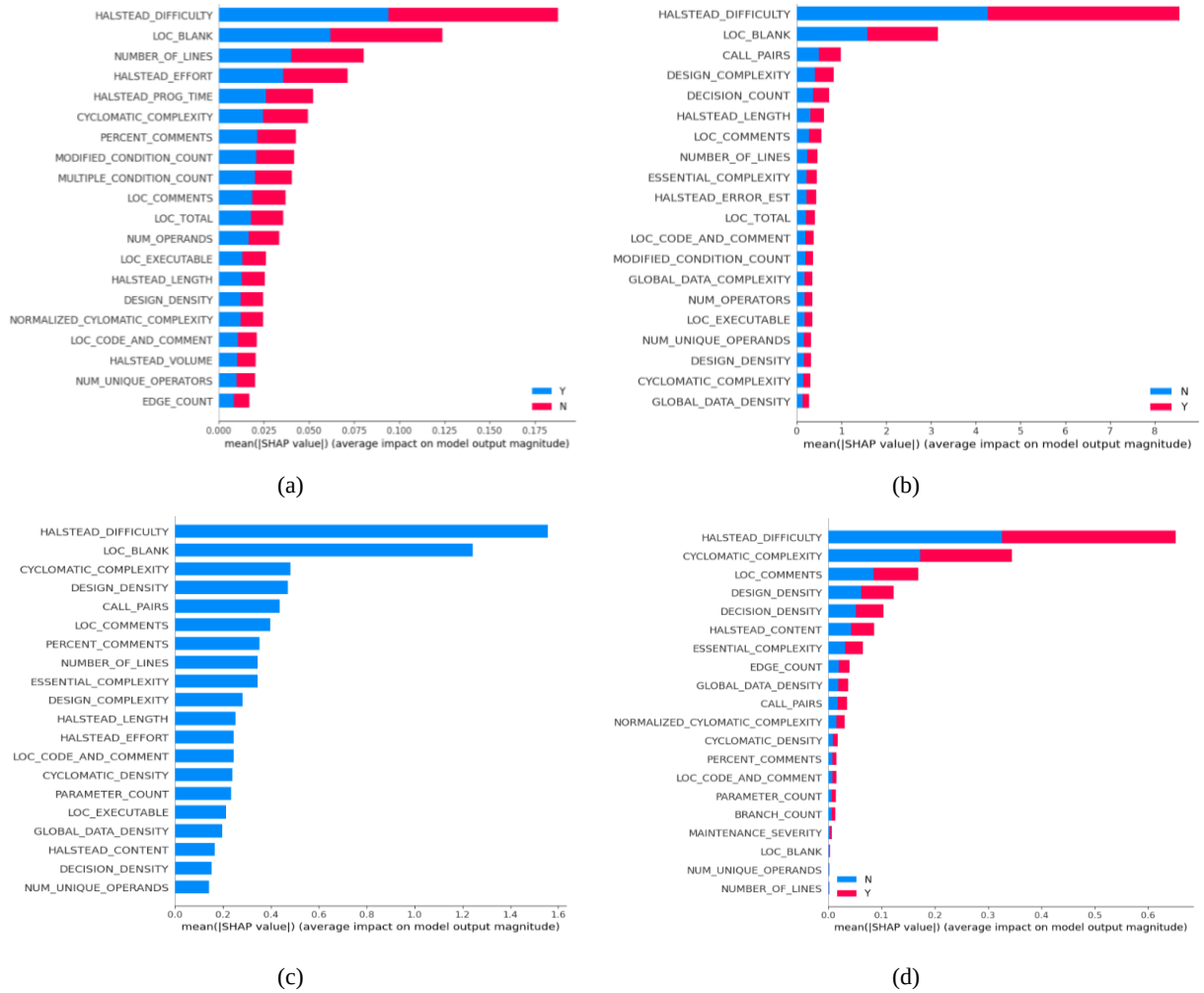

Figure 5. SHAP summary prediction performance showing for (a) Random Forest, (b) LightGbm, (c) XGboost, (d) Decision Tree

Figure 6 shows a lime explanation for each of the models for the top six features. Random Forest model predicted defective software with 59% confidence based on features like Normalized_cyclomatic_complexity, multiple_condition_count, and decision density. While XGboost predicted non-defective with 96% confidence based on features like design_density, parameter_count, global_bank_density, and loc_bank. These five algorithms were fitted on E1 and E2 as the base model, it

was observed that RandomForest and XGBoost came out with the best accuracy, where RF had an AUC score of 95% and XG with 69% using the E1 approach as seen in Table 2.

Figure 6. Showing prediction assessment using XAI-lime for the individual models

On the E2 data model with the same five algorithms fitted, XG and RF still came out as the best two models with AUC of 79% and 95% respectively. Given the result on E1 and E2 for the base model. It clearly showed that RF was performing better than XG on both approaches, but when their confusion metrics were compared, surprisingly XG on E1 was doing better in terms of classifying defective software as True Negative (TN). XG on E1 classified a total of 570 as True Negative while RF on E1 classified 568 as True Negative. Then for E2, both XG and RF classified 569 as True Negative. Optimizing the two best-performing models (XG and RF) using GridSearchCV did not show any significant improvement in the model's performance. The overall best model in this study came out as the Stacking Algorithm Classifier, as seen in Table 4, which had five different models with different strengths combined as one single model [23]. All the evaluation metrics used for validating the performance like accuracy, f1 score, recall, and AUC all had above 90% on both E1 and E2. A closer look at the comparison with their confusion matrices showed that the stacking classifier was performing better with the E1 approach than with E2. The confusion matrix for the Stacking algorithm model compared when tested for E1 and E2 revealed that the stacking algorithm on E1 was able to classify 5468+575 as the correct prediction and 23+49 as the incorrect prediction. In contrast, the algorithm on E2 classified 5439+564 as correct prediction and 53+59 as incorrect prediction. This, therefore, shows that the stack algorithm on E1 was able to classify a total of 575 as defective software compared to all other models, placing it as the overall best-performing model. Also, each of the models developed in this study was tested for any chances of over-fitting as this is a common challenge

encountered during model training or optimization. There was no sign of over-fitting throughout the stage of training these models as revealed in all the result tables given by the difference between the training and test scores.

4. Conclusion

In this paper, we have presented a comprehensive approach to address the class imbalance in machine learning tasks. By utilizing stratified splitting, explainable AI techniques (Lime and Shape), and a hybrid machine learning algorithm, we successfully mitigated the impact of class imbalance, enhanced interpretability, and improved prediction accuracy. The implementation of stratified splitting ensured that class distribution was maintained, enabling the model to effectively learn from minority class examples, thereby reducing bias towards the majority class. Incorporating XAI such as Lime and Shap, provided interpretability to machine learning models. Lime offered local interpretability by generating explanations for individual predictions, allowing us to understand the reasoning behind the model's decisions. Shap on the other hand attributed the contribution of each feature, offering valuable insights into the importance and impact of different features on the model prediction. The proposed hybrid algorithm allowed us to capture diverse patterns and relationships in the data, leading to enhanced predictive capabilities.

Acknowledgement

This research is funded by Woosong University Academic Research 2024.

References

[1] Kiran Maharana, Surajit Mondal and Bhushankumar Nemade, "A review: Data pre-processing and data augmentation techniques", in *Global Transitions Proceedings*, Vol. 3, No. 1, pp. 91-99, June 2022, ISSN: 2666-285X, Published by Elsevier B.V., DOI: 10.1016/j.gltp.2022.04.020, Available: https://www.sciencedirect.com/science/article/pii/S2666285X22000565.

[2] Anuradha Chug and Shafali Dhall, "Software defect prediction using supervised learning algorithm and unsupervised learning algorithm", In *Proceedings of the 4th International Conference Confluence 2013: The Next Generation Information Technology Summit*, Noida, India, 26-27 September 2013, ISBN:978-1-84919-846-2, Published by IEEE Xplore, DOI: 10.1049/cp.2013.2313, Available: https://ieeexplore.ieee.org/document/6832328.

[3] Zeyu Wang, Jian Liu, Yuanxin Zhang, Hongping Yuan, Ruixue Zhang *et al.*, "Practical issues in implementing machine-learning models for building energy efficiency: Moving beyond obstacles", *Renewable and Sustainable Energy Reviews*, ISSN: 1364-0321, pp. 110929, Vol. 143, June 2021, Published by Elsevier BV, DOI: 10.1016/j.rser.2021.110929, Available: http://www.sciencedirect.com/science/article/pii/S1364032121002227.

[4] Romi S. Wahono and Nanna Suryana, "Combining particle swarm optimization-based feature selection and bagging technique for software defect prediction", *International Journal of Software Engineering and Its Applications*, ISSN: 1738-9984, Vol. 7, No. 5, pp. 153-166, 2013, DOI: 10.14257/ijseia.2013.7.5.16, Available: https://digital-library.theiet.org/content/conferences/10.1049/cp.2013.2293.

[5] Tim Menzies, Jeremy Greenwald and Art Frank, "Data mining static code attributes to learn defect predictors", *IEEE transactions on Software Engineering*, ISSN: 0098-5589, Vol. 33, No. 1, pp. 2-13, 2006, DOI: 10.1109/TSE.2007.256941, Available: https://ieeexplore.ieee.org/abstract/document/4027145.

[6] Karim O. Elish and Mahmoud O. Elish, "Predicting defect-prone software modules using support vector machines", *Journal of Systems and Software*, Vol. 81, No. 5, pp. 649-660, 2008, DOI: 10.1016/j.jss.2007.07.040, Available: https://www.sciencedirect.com/science/article/abs/pii/S016412120700235X.

[7] Nachiappan Nagappan, Brendan Murphy and Victor Basili, "The influence of organizational structure on software quality: an empirical case study", In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 10-18 May 2008, pp. 521-530, Published by ACM Digital Library, DOI: 10.1145/1368088.1368160, Available: https://dl.acm.org/doi/10.1145/1368088.1368160.

[8] Burak Turhan, Tim Menzies, Ayşe B. Bener and Justin Di Stefano, "On the relative value of cross-company and within-company data for defect prediction", *Empirical Software Engineering*, No. 14, pp. 540-578, January 2009, DOI: 10.1007/s10664-008-9103-7, Available: https://link.springer.com/article/10.1007/s10664-008-9103-7.

[9] Fei Wu, Xiao-Y. Jing, Shiguang Shan, Wangmeng Zuo and Jing-Y. Yang, "Multiset feature learning for highly imbalanced data classification", In *Proceedings of the AAAI conference on artificial intelligence*, Washington DC, USA, 4-9 February 2017, Vol. 31, No. 1, San Francisco, California USA, DOI: 10.1609/aaai.v31i1.10739, https://ojs.aaai.org/index.php/AAAI/article/view/10739.

[10] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan and Kenichi Matsumoto, "The Impact of Automated Parameter Optimization on Defect Prediction Models", *IEEE Transactions on Software Engineering*, Print ISSN: 0098-5589, Vol. 45, No. 7, pp. 683-711, July 2019, DOI: 10.1109/TSE.2018.2794977, Available: https://ieeexplore.ieee.org/abstract/document/8263202.

[11] Diana-L. Miholca, "An Improved Approach to Software Defect Prediction using a Hybrid Machine Learning Model", In *2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (SYNASC), Timisoara, Romania, 20-23 September 2018, pp. 443-448, ISBN:978-1-7281-0626-7, DOI: 10.1109/SYNASC.2018.00074, Available: https://ieeexplore.ieee.org/document/8750697.

[12] Lei Qiao, Xuesong Li, Qasim Umer and Ping Guo, "Deep learning-based software defect prediction", *Neurocomputing*, Vol. 385, pp. 100-110, April 2020, ISSN: 0925-2312, Elsevier, DOI: 10.1016/j.neucom.2019.11.067, Avialable: https://www.sciencedirect.com/science/article/abs/pii/S0925231219316698.

[13] Amir Elmishali and Meir Kalech, "Issues-Driven features for software fault prediction", *Information and Software Technology*, Vol. 155, March 2023, ISSN: 0950-5849, DOI: 10.1016/j.infsof.2022.107102, Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584922002117.

[14] Lina Jia, "A hybrid feature selection method for software defect prediction," In *IOP Conference Series: Materials Science and Engineering*, Vol. 394, pp. 032035, August 2018, IOP Publishing, ISSN: 1757-899X, DOI: 10.1088/1757-899X/394/3/032035, Available: https://iopscience.iop.org/article/10.1088/1757-899X/394/3/032035.

[15] Sharma Tarunim, Aman Jatain, Shalini Bhaskar and Kavita Pabreja, "Ensemble Machine Learning Paradigms in Software Defect Prediction," In *Procedia Computer Science*, Vol. 218, pp. 199-209, 2023, ISSN: 1877-0509, DOI: 10.1016/j.procs.2023.01.002, Available: https://www.sciencedirect.com/science/article/pii/S1877050923000029.

[16] Uzma Raja, David P. Hale and Joanne E. Hale, "Modeling software evolution defects: a time series approach", *Software Maintenance and Evolution: Research and Practice*, Vol. 21, No. 1, pp. 49-71, December 2008, DOI: 10.1002/smr.398, Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.398.

[17] Sweta Mehta and Patnaik K. Sridhar, "Improved prediction of software defects using ensemble machine learning techniques", *Neural Computing and Application*, Vol. 33, pp. 10551-10562, March 2021, Print ISSN: 0941-0643, DOI: 10.1007/s00521-021-05811-3, Available: https://link.springer.com/article/10.1007/s00521-021-05811-3.

[18] Pandey Sanchita and Kuldeep Kumar, "Software Fault Prediction for Imbalanced Data: A Survey on Recent Developments", In *Procedia Computer Science*, ISSN: 1877-0509, Vol. 218, pp. 1815-1824, January 2023, DOI: 10.1016/j.procs.2023.01.159, Available: https://www.sciencedirect.com/science/article/pii/S187705092300159X.

[19] Alsaeedi Abdullah and Mohammad Z. Khan, "Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study", *Journal of Software Engineering and Applications*, ISSN Online: 1945-3124, Vol. 12, No. 5, pp. 85-100, 2019, Published by SCIRP, DOI: 10.4236/jss.2015.37034, Available: https://www.scirp.org/journal/paperinformation?paperid=92522.

[20] Daniel Rodriguez, Roberto Ruiz, Jose C. Riquelme and Rachel Harrison, "A study of subgroup discovery approaches for defect prediction", *Information and Software Technology*, ISSN: 0950-5849, Vol. 55, No. 10, pp. 1810-1822, October 2013, Published by Elservier, DOI: 10.1016/j.infsof.2023.05.002, Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584913001018.

[21] Thanh T. Khuat and My H. Le, "Evaluation of Sampling-Based Ensembles of Classifiers on Imbalanced Data for Software Defect Prediction Problems", *SN Computer Science*, No. 1, pp. 108, March 2020, Published by springernature, DOI: 10.1007/s42979-020-0119, Available: https://link.springer.com/article/10.1007/s42979-020-0119-4.

[22] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu and Baowen Xu, "Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction", In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM, 2017), Toronto, ON, Canada, pp. 11-19, ISBN:978-1-5090-4040-7, Published by IEEE, DOI: 10.1109/ESEM.2017.8, Available: https://ieeexplore.ieee.org/document/8169980.

[23] Issam H. Laradji, Mohammad Alshayeb and Lahouari Ghouti., "Software defect prediction using ensemble learning on selected features", *Information and Software Technology*, ISSN: 0950-5849, Vol. 58, pp. 388-402, February 2015, Published by Elsevier, DOI: 10.1016/j.infsof.2014.07.005, Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584914001591.

[24] Yang Liu, Yan Kang, Chaoping Xing, Tianjian Chen and Qiang Yang, "A Secure Federated Transfer Learning Framework", *IEEE Intelligent Systems*, Print ISSN: 1541-1672, Vol. 35, No. 4, pp. 70-82, Published by IEEE, DOI: 10.1109/MIS.2020.2988525, Available: https://ieeexplore.ieee.org/document/9076003.

[25] Abdullah A. Mamun, Md Sohel, Naeem Mohammad, Md Samiul H. Sunny, Debopriya R. Dipta *et al.*, "A Comprehensive Review of the Load Forecasting Techniques Using Single and Hybrid Predictive Models", *IEEE Access*, ISSN: 2169-3536, pp. 134911-134939, July 2020, Published by IEEE, DOI: 10.1109/ACCESS.2020.3010702, Available: https://ieeexplore.ieee.org/document/9144528.

[26] Amirabbas Majd, Mojtaba V. Asl, Alireza Khalilian, Pooria P.-Tehrani and Hassan Haghighi, "SLDeep: Statement-level software defect prediction using deep-learning model on static code features", *Expert Systems with Applications*, ISSN: 0957-4174, Vol. 147, June 2020, Published by Elsevier B.V., DOI: 10.1016/j.eswa.2019.113156, Available: https://www.sciencedirect.com/science/article/abs/pii/S0957417419308735.

[27] Maram Assi, Safwat Hassan, Stefanos Georgiou and Ying Zou, "Predicting the Change Impact of Resolving Defects by Leveraging the Topics of Issue Reports in Open Source Software Systems", *Software Engineering and Methodology*, ISSN: 1049-331X, Vol. 32, No. 6, pp. 1-34, September 2023, Published by ACM, DOI: 10.1145/3593802, Available: https://dl.acm.org/doi/abs/10.1145/3593802.

[28] Kiran Maharana, Surajit Mondal and Bhushankumar Nemade, "A review: Data pre-processing and data augmentation techniques", In *Global Transitions Proceedings*, ISSN: 2666-285X, Vol. 1, No. 3, pp. 91-99, 2022, DOI: 10.1016/j.gltp.2022.04.020, Available: https://www.sciencedirect.com/science/article/pii/S2666285X22000565.

[29] Rudresh Dwivedi, Devam Dave, Het Naik, Smiti Singhal, Rana Omer *et al.* "Explainable AI (XAI): Core Ideas, Techniques, and Solutions", *ACM Journals*, ISSN: 0360-0300, Vol. 55, No. 9, pp. 1-33, Januray 2023, Published by CSUR, DOI: 10.1145/3561048, Available: https://dl.acm.org/doi/10.1145/3561048.

[30] Momotaz Begum, Jahid H. Rony, Md R. Islam and Jia Uddin, "Long-Term Software Fault Prediction Model with Linear Regression and Data Transformation", *Journal of Informatiom Systems and Telecommunication*, ISSN: 2322-1437, Vol. 11, No. 3, pp. 222-231, July-September 2023, Published by JIST, DOI: 10.61186/jist.36585.11.43.222, Available: http://jist.ir/Article/36585/FullText.

[31] Momotaz Begum, Mehedi H. Shuvo, Imran Ashraf, Abdullah A. Mamun, Jia Uddin *et al.*, "Software Defects Identification: Results Using Machine Learning and Explainable Artificial Intelligence Techniques", *IEEE Access*, Vol. 11, pp. 132750-132765, 2023, DOI: 10.1109/ACCESS.2023.3329051, Available: https://ieeexplore.ieee.org/abstract/document/10304128.