Integrating LightGBM and XGBoost for Software Defect Classification Problem

Gregorius Airlangga*

Engineering Faculty, Information System Study Program, Atma Jaya Catholic University of Indonesia, Jakarta, Indonesia
Email: gregorius.airlangga@atmajaya.ac.id
Correspondence Author Email: gregorius.airlangga@atmajaya.ac.id

Abstract—Software defect classification is a crucial process in quality assurance, pivotal for the development of reliable software systems. This paper presents an innovative approach that synergizes traditional software complexity metrics with advanced machine learning algorithms, namely Light Gradient Boosting Machine (LightGBM) and Extreme Gradient Boosting (XGBoost), to enhance the accuracy and efficiency of software defect classification. Leveraging a dataset characterized by McCabe’s and Halstead’s metrics, this study embarks on meticulous data preprocessing, feature engineering, and hyperparameter optimization to train and evaluate the proposed models. The LightGBM and XGBoost models are fine-tuned through the Optuna framework, aiming to maximize the ROC-AUC score as a measure of classification performance. The results indicate that both models perform robustly, with XGBoost demonstrating a slight superiority in predictive capability. The integration of machine learning with traditional complexity metrics not only enhances the defect classification process but also provides deeper insights into the factors influencing software quality. The findings suggest that such hybrid approaches can significantly contribute to the predictive analytics tools available to software engineers and quality assurance professionals. This research contributes to the field by offering a comprehensive methodological framework and empirical evidence for the effectiveness of combining machine learning algorithms with traditional software complexity metrics in software defect classification.

Keywords: Software Defect; Classification; LightGBM; XGBoost; Machine Learning

1. INTRODUCTION

In the quest for quality and reliability in software engineering, the classification and management of software defects remain pivotal [1]–[4]. Traditional approaches, grounded in the use of software complexity metrics like McCabe’s Cyclomatic Complexity and Halstead’s Software Science Metrics, have provided foundational insights into the inherent complexities of software systems [5]–[7]. Yet, the rapid evolution in the field of machine learning, particularly with the advent of algorithms like Light Gradient Boosting Machine (LightGBM) and Extreme Gradient Boosting (XGBoost), offers a promising horizon for revolutionizing software defect classification [8]–[10]. This urgency is underscored by the increasing complexity and criticality of modern software systems, where even minor defects can lead to significant financial loss, security breaches, and loss of human life in critical applications. Our paper presents an integrated model that synergizes these conventional metrics with advanced machine learning techniques, aiming to significantly enhance the accuracy and efficiency of software defect classification, thereby addressing these critical challenges. The journey of software defect classification has witnessed a paradigm shift from simplistic, metric-based models to intricate, data-driven approaches [11]–[13]. In the initial stages, the focus was predominantly on structural complexity metrics. McCabe’s Cyclomatic Complexity, introduced in the 1970s, brought a novel perspective by quantifying the decision points in a program, thereby providing a measurable insight into its complexity. This metric has been widely linked to various aspects of software quality, especially in predicting potential hotspots for defects [14]–[16]. Halstead’s Software Science Metrics further enriched this domain by introducing a set of metrics that reflect the operational and size characteristics of software.

These metrics, encompassing aspects like volume, effort, and difficulty, have been extensively used to assess the cognitive load involved in software comprehension and maintenance, thus acting as indirect indicators of software quality [17]–[19]. However, the emergence of machine learning has introduced a transformative approach to defect classification. Algorithms like LightGBM and XGBoost, known for their efficiency and efficacy, have demonstrated exceptional capabilities in classification tasks across diverse sectors [20]–[22]. Their application in software defect classification is relatively recent but holds immense potential. The ability of these algorithms to process large datasets, handle a variety of feature types, and their superior performance in terms of accuracy and speed positions them as ideal candidates for enhancing traditional defect classification methods. The literature survey critically examines the evolution of software defect classification, tracing the journey from traditional software complexity metrics to the incorporation of machine learning techniques such as LightGBM and XGBoost [23]. Our research draws significantly from that work, since we provide foundational framework for understanding how these advanced techniques could solve software defect problem.

As far as we know, since founded the McCabe’s Cyclomatic Complexity, this metric quantified software complexity based on control flow within programs [24]. It became a standard tool for predicting error-proneness and assessing software maintainability. Furthermore, Halstead proposed a set of metrics that focused on the size and operational characteristics of software. These metrics, including Effort, Time Estimator, and Volume, offered a new perspective on understanding software complexity and its implications on software quality [25]. The early
21st centuries witnessed a gradual shift towards data-driven approaches in software defect classification. This transition was marked by the introduction of statistical models and early machine learning techniques [26]. Studies during this period began to explore the use of classification algorithms, like decision trees and neural networks, to enhance the prediction accuracy beyond what was achievable with traditional metrics alone [27]. Recent years have seen the advent of more sophisticated machine learning techniques in software defect classification. The integration of machine learning algorithms with traditional software complexity metrics represents a significant advancement in the field. The current state-of-the-art in software defect classification has been significantly influenced by the integration of machine learning techniques with traditional complexity metrics. Recent studies have focused on the use of ensemble methods, like random forests and gradient boosting machines, which combine multiple algorithms to improve prediction accuracy [28]. These methods have been found to outperform traditional single-algorithm approaches in terms of both precision and recall. Studies have reaffirmed the relevance of McCabe's and Halstead's metrics in the modern context of software defect prediction [29]. These metrics are often used as baseline indicators of software complexity and maintainability, serving as crucial inputs for more advanced analytical models. A number of recent papers have demonstrated the effectiveness of LightGBM and XGBoost in various classification tasks, including software defect prediction. Their ability to handle large datasets, feature importance analysis, and efficient computation makes them ideal for this application. Comparative analyses between traditional metrics-based approaches and machine learning-based models have shown a significant improvement in defect classification when using the latter [30].

These studies provide a foundation for the use of machine learning techniques in software quality assurance. Our research contributes to the existing body of knowledge in several significant ways. Firstly, it involves implementing machine learning algorithms, specifically LightGBM and XGBoost, for defect classification. Additionally, it includes an in-depth comparative analysis of these algorithms, namely LightGBM and XGBoost, in the context of software defect prediction. This analysis utilizes a range of complexity metrics to offer a comprehensive evaluation. Furthermore, our work explores the interpretability of these machine learning models in correlation with established software engineering principles. This aspect is vital for understanding how these models operate within the parameters of software engineering. Lastly, the research examines feature importance as identified by these models, providing valuable insights into the key predictors of software defects. This examination helps in understanding the factors that most significantly contribute to software vulnerabilities, thereby aiding in their early identification and rectification. The remaining structure of the paper is as follows, in section 2, we will delve into the dataset specifications, the experimental environment, the training and testing methodology, the metrics used for evaluating the performance of the models, and the explanation about algorithms: XGBoost and LightGBM. Unlike previous studies, which primarily focused on either the application of traditional software complexity metrics or the exploration of machine learning techniques in isolation, our research integrates these two approaches. We aim to assess how the combination of advanced machine learning algorithms with established complexity metrics can enhance the accuracy and efficiency of software defect classification, an aspect not extensively explored in prior works. In section 3 we explain about results and detailed Analysis, we present a thorough analysis of the experimental results, including a comparison of the performance of LightGBM and XGBoost. This comparison is crucial, as it highlights the differences in efficiency and effectiveness between these two leading machine learning algorithms in the context of software defect classification. Additionally, this section explores how these results align with the theoretical frameworks of software engineering, thereby bridging the gap between empirical results and established theories. The purpose of this comprehensive analysis is not only to contribute to the existing body of knowledge but also to provide practical insights for software engineers and developers in choosing appropriate methodologies for defect classification. This section also explores how these results align with the theoretical frameworks of software engineering. In the last section, we conclude the paper by summarizing the main findings and contributions, highlighting the significance of combining machine learning algorithms with traditional complexity metrics in the realm of software defect classification.

2. RESEARCH METHODOLOGY

This research methodology commences with the 'Data Collection and Processing' phase, where foundational approaches from McCabe's and Halstead's methods are employed for evaluating software complexity, including the calculation of Cyclomatic Complexity. This phase is crucial for ensuring data integrity, involving preprocessing steps like handling missing values, converting categorical data, and removing anomalies and duplicates. The process seamlessly transitions into 'Feature Engineering', a phase that capitalizes on existing McCabe and Halstead metrics while innovating with new features such as Complexity (C), ComplexityEffort (CE), and LocToLen (LTL). This phase is not just about feature creation but also about meticulous selection and transformation to refine the dataset for optimal modeling performance. Upon completing feature engineering, the methodology advances to 'Model Development and Training'. Here, the dataset is divided into training and testing sets, and advanced models like LightGBM and XGBoost are intensively trained, with a keen focus on hyperparameter optimization using the Optuna framework, all aimed at maximizing the ROC-AUC score. This stage is critical for fine-tuning the models to achieve high accuracy and reliability. The culmination of this
methodological journey is the 'Model Evaluation and Comparison' stage, where the performance of the models is rigorously evaluated based on the ROC-AUC score. A comparative analysis between integration of LightGBM and XGBoost and each model performance itself are conducted to ascertain their efficacy in defect classification. Additionally, a voting classifier is implemented, combining the strengths of both models, and evaluated for its predictive power. This stage also includes a feature importance analysis to identify key predictors and concludes with a thorough interpretation of the findings, contextualizing them within the realm of software engineering. This comprehensive approach ensures a robust and detailed understanding of software complexities, paving the way for effective defect prediction and enhanced software quality.

**Figure 1. Research Methodology Diagram**

### 2.1 Data Collection and Processing

McCabe’s and Halstead’s methods are foundational approaches in the field of software engineering, specifically utilized for evaluating the complexity of computer programs. These methods offer distinct perspectives and tools.
for understanding various aspects of a program's structure and operations. Cyclomatic Complexity, conceptualized by Thomas J. McCabe, is a pivotal metric in this domain. It essentially quantifies the complexity of a program by examining its control flow. The core idea is that a higher Cyclomatic Complexity suggests a program with a greater number of decision points, thereby implying increased complexity. In practical terms, programs exhibiting high values of this metric are often more challenging to understand, test, and maintain. This is because the intricate decision paths can lead to difficulties in predicting program behavior and ensuring reliability.

On the other hand, Maurice Halstead's Software Science Metrics offer a different approach. Halstead's metrics delve into evaluating the size and complexity of a program through a series of quantifiable measures. These metrics include various aspects such as the total number of operators and operands, the volume, which is indicative of the mental effort needed for programming, and the program's length. A crucial aspect of Halstead's metrics is the interpretation of values for volume, difficulty, and effort. Elevated values in these areas typically suggest greater complexity. Additionally, a higher intelligence value in Halstead's metrics indicates a more efficient program design, where greater functionality is achieved with lesser effort. In the realm of software complexity metrics, several key features are commonly analyzed. For instance, 'loc' or Lines of Code, is a straightforward measure representing the total number of lines in a module’s code. The 'v(g)' or Cyclomatic Complexity metric assesses the number of linearly independent paths within the program's control flow, providing insight into the complexity of the code. Alongside, 'ev(g)' and 'iv(g)' represent the essential and design complexities of a program, respectively, offering a nuanced view of the underlying structural complexity.

The Halstead metrics further extend this analysis with 'n', representing the total number of operators and operands, 'V' for volume, 'I' for program length, 'D' for difficulty, 'I' for intelligence, 'E' for effort, and 'T' for the estimated time required for development. These metrics collectively offer a comprehensive understanding of the program's complexity, effort, and efficiency. Additional features like 'LOC' or 'L(0)Code', 'LOC' or 'L(0)Comment', and 'LOB' or 'L(0)Blank' count the lines of code, comments, and blank lines, respectively, offering insights into the coding style and documentation. Unique operators and operands ('uniqOp' and 'uniqOpnd') are counted to understand the diversity of the code's elements, while 'totalOp' and 'totalOpnd' give a total count of these elements. Lastly, the 'branchCount' is a measure of the branching in the flow graph of the program, and 'defects' is a binary variable indicating the presence or absence of reported defects in the module. This comprehensive set of metrics, spanning from McCabe's Cyclomatic Complexity to Halstead's detailed measures, provides a multidimensional view of software complexity, essential for understanding, maintaining, and improving software systems. The dataset utilized in this study encapsulates a range of software complexity metrics, including those established by McCabe and Halstead, and goes through a comprehensive preprocessing phase to ensure data quality and consistency. Initially, any missing values in the dataset, which are represented as '?', are replaced with NaN to maintain consistency across the dataset. This is followed by the conversion of categorical data into a numerical format, a crucial step that facilitates more effective analysis and modeling. The dataset is then meticulously scrutinized for any anomalies or inconsistencies that might skew the results or affect the modeling process. Finally, to maintain the integrity and reliability of the data, any duplicate entries identified in the dataset are removed. These preprocessing steps are fundamental in preparing the dataset for the subsequent stages of the study, ensuring that the data is clean, consistent, and ready for effective analysis.

2.2 Feature Engineering

Feature engineering is a pivotal step in preparing the dataset for modeling in our study. This intricate process includes several key activities. First, we utilize the existing features in the dataset, which comprise a blend of McCabe's and Halstead's metrics, among others. These established metrics form the foundation of our feature set. Next, we engage in engineering new features to enrich the dataset further. This involves crafting novel features based on specific formulas, such as Complexity (C), ComplexEffort and LocToLen as presented in the equation 1,2 and 3.

\[
\text{Complexity (C)} = v(g) \ast ev(g) \ast iv(g)
\]

(1)

\[
\text{ComplexEffort (CE)} = C \ast e
\]

(2)

\[
\text{LocToLen (LTL)} = \frac{loc}{(1+0.0001)}
\]

(3)

Finally, we undertake a rigorous feature selection and transformation process. This includes evaluating features for their correlation and applying necessary transformations to reduce multicollinearity. Features that exhibit high correlation are subject to detailed scrutiny, and some may be dropped to optimize the performance of the model. This step is crucial in ensuring that only the most relevant and impactful features are included in the final modeling process, thereby enhancing the accuracy and efficiency of the predictive models.

2.3 Model development and Training

In this study, LightGBM and XGBoost are the chosen models for their efficiency and effectiveness in handling complex datasets. The development and training of these models follow a structured process. Initially, the dataset is carefully divided into two subsets: the training set and the testing set. This division is crucial for both training
the models effectively and evaluating their performance accurately. Once the data splitting is completed, the focus shifts to the model training phase. Both LightGBM and XGBoost models are trained extensively on the training dataset, with a key emphasis on fine-tuning their hyperparameters to achieve optimal performance. For this purpose, the Optuna framework is employed, which is renowned for its efficiency in hyperparameter optimization. The primary objective of this optimization is to maximize the ROC-AUC score, a critical measure of model performance. The ROC-AUC score as presented in equation 4 is calculated as the sum of the products of the True Positive Rate (TPR) and the False Positive Rate (FPR) at various threshold settings. This metric effectively captures the balance between the model's sensitivity (TPR) and its specificity (FPR), providing a robust evaluation of its overall predictive power. This meticulous training process ensures that the models are not only well-tuned but also ready to deliver high-accuracy predictions on the testing set.

\[
\text{ROC} - \text{AUC} = \sum \text{TPR} \times \text{FPR} 
\]

where TPR (True Positive Rate) and FPR (False Positive Rate) are calculated for various threshold settings.

2.4 Model evaluation and comparison

The evaluation of models is based on the ROC-AUC score and confusion matrices. A comparative analysis of LightGBM and XGBoost is conducted to assess their performance in defect classification. Feature importance analysis is also performed to identify key predictors of defects. A voting classifier is implemented to combine the predictions from LightGBM and XGBoost. The classifier uses a 'soft' voting mechanism and is evaluated using the ROC-AUC score and confusion matrices. The findings are interpreted in the context of software engineering, highlighting the relationship between software complexity metrics and defect likelihood. The study discusses the implications of these findings for defect prediction in software systems.

2.5 Light GBM Method

Light Gradient Boosting Machine (LightGBM) represents a significant advancement in the field of machine learning, particularly in the realm of gradient boosting frameworks [31]. This method is renowned for its efficiency and effectiveness, especially when dealing with large datasets. Its foundation lies in the principle of sequentially building models, specifically decision trees, each aimed at correcting the errors of its predecessors. The culmination of this process is an ensemble model that boasts improved accuracy and robustness. At the heart of LightGBM's methodology are decision trees, which serve as the base learners. These trees are constructed through a greedy algorithm, focusing on maximizing information gain at each split. This approach is instrumental in the model's ability to learn complex patterns in data.

There are two innovative techniques that contribute significantly to LightGBM's efficiency: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). GOSS enhances the learning process by focusing on instances with larger gradients, thus prioritizing more significant errors. EFB, on the other hand, efficiently handles sparse data by bundling exclusive features, thereby reducing the feature space without losing valuable information. The mathematical essence of LightGBM is encapsulated in its optimization of a specific loss function. Consider a dataset comprising N instances, each with M features. Let \( F \) represent the ensemble model and \( y_i \) the corresponding label for each instance \( i \). The loss function \( L \), a crucial component, is defined to quantify the discrepancy between the predicted values \( F(x_i) \) and the actual target values \( y_i \). This is mathematically represented as presented in equation 5.

\[
L(F) = \sum_{i=1}^{N} F(x_i, y_i) 
\]

During the training phase, each iteration involves introducing a new tree \( h_t \) to the ensemble, aimed at minimizing the overall loss. This process can be described by the following equation 6.

\[
F_t(x) = F_{t-1}(x) + \alpha \times h_t(x) 
\]

Here, \( \alpha \) denotes the learning rate, a parameter that controls the contribution of each new tree. The optimization of the ensemble model, guided by gradient descent, focuses on the negative gradient of the loss function. The objective is to find the best tree \( h_t \) in each iteration, a process described by equation 7.

\[
h_t = \text{arg min} \sum_{i=1}^{N} g_i \times h_{xi} + 0.5 \times h_{2xi} \times h_{xi} 
\]

In this expression, \( g_i \) represents the gradient of the loss function with respect to \( F_{t-1}(x) \). To Handling Overfitting and Categorical Features, LightGBM incorporates regularization to mitigate overfitting, a common challenge in machine learning models. Additionally, it employs strategies like early stopping, setting a maximum depth for trees, and data subsampling. Moreover, LightGBM adeptly handles categorical features, converting them into numerical values through methods such as one-hot encoding. LightGBM stands as a testament to the advancements in machine learning, offering a blend of efficiency, accuracy, and flexibility. Its ability to process large datasets with remarkable speed, without compromising on the model's performance, makes it a valuable tool in the arsenal of data scientists and researchers.
2.6. XGBoost Method

XGBoost is a part of the gradient boosting framework, where new models are added sequentially to correct the errors made by previous models [32]. The final prediction model is an ensemble of these individual models, typically decision trees. The core of XGBoost is an objective function that needs to be minimized, which is a combination of a loss function and a regularization term as presented in equation 8.

\[
\text{Obj}(\Theta) = \text{L}(\Theta) + \Omega(\Theta)
\]  

(8)

L is the training loss function, and \( \Omega \) represents the regularization term. Furthermore, the loss function L measures the difference between the predicted values and the actual target values. For a set of \( n \) predictions and corresponding targets, it is given in equation 9.

\[
\text{L}(\Theta) = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2
\]  

(9)

where \( y_i \) is the actual value, and \( \hat{y}_i \) is the predicted value. Then, the regularization term \( \Omega \) helps to smooth the final learned weights to avoid overfitting. It is defined as \( \Omega(\Theta) = \gamma T + 1/2 \lambda \sum_{j=1}^{T} w_j^2 \), where \( T \) is the number of leaves in the tree, \( w_j \) is the score on each leaf, \( \gamma \) and \( \lambda \) are regularization parameters. In XGBoost, the prediction model is an ensemble of \( K \) additive functions (trees) such as \( \hat{y}_i = \sum_{k=1}^{K} f_{k}(\Theta_k) \), where \( F \) is the space of regression trees. XGBoost adds one tree at a time. Each tree is trained to minimize the objective function, using the gradient descent method. Shrinkage (learning rate) scales the contribution of each new tree to prevent overfitting. XGBoost can automatically handle missing values. When a split on a feature is made, the algorithm learns the direction to assign missing values based on reduction in the loss function. XGBoost includes a built-in cross-validation method at each iteration, which eliminates the need for a separate cross-validation loop.

2.7. Voting-based Classification

In a voting-based classifier, multiple machine learning models are used to make predictions for the same dataset. The final output is determined based on the 'votes' from all models. There are two main types of voting: hard voting and soft voting. Hard voting uses the mode of all predictions, while soft voting averages the probability estimates from each model. LightGBM and XGBoost, both being gradient boosting frameworks, are ideal candidates for a voting classifier due to their individual strengths in handling large datasets and diverse feature spaces. The combination leverages LightGBM's efficiency with categorical data and XGBoost's robust handling of missing values [33]–[35].

Consider a classification problem with \( N \) instances and \( M \) features. Let \( y_i \) be the actual label of the \( i \)th instance. LightGBM and XGBoost models generate predictions \( \hat{y}_{i,\text{LightGBM}} \) and \( \hat{y}_{i,\text{XGB}} \), respectively. For hard voting, the final prediction \( \hat{y}_i \) is the mode of predictions from both models. For soft voting, if \( p_{i,\text{LightGBM}} \) and \( p_{i,\text{XGB}} \) are the predicted probabilities, the final prediction is based on the average probability: \( \hat{y}_i = (p_{i,\text{LightGBM}} + p_{i,\text{XGB}}) / 2 \). To improve accuracy, the ensemble approach often results in higher accuracy than individual models, especially in cases where models' errors are uncorrelated. To reduce overfitting, it can be solved by combining predictions, the ensemble method can reduce the risk of overfitting compared to a single model. The relative weights of LightGBM and XGBoost in soft voting can be adjusted based on validation performance. Hyperparameter tuning of each model is crucial for optimal performance.

3. RESULTS AND DISCUSSION

The combination of ROC-AUC scores, confusion matrices, feature importance rankings, and correlation heatmaps gives a comprehensive view of the model's performance and the characteristics of the data. The XGBoost model seems to perform slightly better than the combined voting classifier. The feature importance rankings and boxplots highlight the importance of traditional complexity metrics in predicting software defects, while the correlation heatmaps suggest that some feature reduction could potentially improve the model's performance by removing redundancy.
The image in figure 2 explains a correlation matrix, an essential tool in statistical analysis that quantifies the linear relationship between multiple variables in a dataset. In this matrix, each cell represents the correlation coefficient between two variables, ranging from -1 to 1. The color gradient, from dark blue to dark red, visually encodes the strength and direction of the correlation; dark blue signifies a strong negative correlation, while red indicates no correlation, and dark red shows a strong positive correlation. The diagonal, naturally comprised of 1s, indicates the perfect correlation of variables with themselves. The symmetry across this diagonal is due to the reciprocity of correlation coefficients; the correlation of variable A to B is identical to B to A. Observations of high correlation coefficients, either positive or negative, outside the diagonal can indicate potential multicollinearity, a condition where variables are highly interdependent. In predictive modeling, this condition can distort the true relationship with the outcome variable and degrade the model’s performance.

For instance, the variable loc (likely representing 'lines of code') exhibits a high positive correlation with v(g) (possibly a complexity metric such as McCabe's complexity), suggesting that programs with more lines of code tend to have higher complexity. Similarly, total_Op and total_Opnd, perhaps denoting 'total operators' and 'total operands' within the code, show a strong positive correlation. This implies that code segments with a higher count of operators also contain more operands, indicating a complexity that may affect the code's maintainability and potential defect rates. In predictive models, particularly those prone to overfitting such as linear regression, the presence of highly correlated predictors can be problematic. These models rely on the assumption that features are independent of one another. When features are highly correlated, it can lead to unstable estimates of the model parameters, making the model sensitive to minor changes in the model or data. Hence, the correlation matrix is a vital component of the feature selection process. By identifying pairs or sets of variables that are highly correlated, data scientists can make informed decisions about which variables to include or exclude, combine, or transform to improve model performance. Moreover, the correlation matrix aids in uncovering redundant information. Variables that are highly correlated may not be necessary for building a predictive model as they provide similar information about the variance in the data. Including both could lead to inefficiency in the model and an increased computational burden without a commensurate gain in accuracy or insight.

![Feature Importance](https://ejurnal.stmik-budidarma.ac.id/index.php/mib)

**Figure 3. Feature Importance**

The provided image in figure 3 illustrates a horizontal bar chart, commonly used to represent the feature importance derived from a machine learning model. Feature importance is a technique that assigns a score to input features based on how useful they are at predicting a target variable. In this graph, each bar corresponds to a feature (variable) from a dataset, and the length of the bar indicates the importance of that feature in the predictive model. The longer the bar, the more significant the feature is. At the top of the chart, we see features with names like i, loc, d, which could correspond to metrics or characteristics in a software defect prediction model. The i feature has the highest importance score, followed very closely by loc. This suggests that these two features are the strongest predictors within the model. For instance, i could be an identifier for an individual module or component within a software system, and its high importance could imply that certain modules are more prone to defects. Similarly, loc might stand for “lines of code,” and its prominence in the chart suggests that the number of lines in a code module is a strong indicator of the likelihood of defects, possibly due to the complexity or size of the codebase.

The feature d and n also have significant importance scores, though not as high as i and loc. These might represent other complexity metrics or attributes of the software that influence its quality or propensity for defects. For example, d might stand for “dependency count” and n for “number of nodes” in a dependency graph. The complexity of these relationships could be critical in understanding the robustness of the software. Further down, we see a feature named ComplexEffort, which also has a considerable importance score, implying that the effort required to develop or maintain a component is a key factor in predicting defects. LocToLen, which might measure
the ratio of lines of code to module length or size, also appears to be a relevant metric. Its significant score indicates that the relative size of a code unit, not just the absolute amount of code, has a substantial impact on defect prediction.

IOBlank, possibly referring to the count of blank or non-executable lines such as input/output definitions, and uniq_Op, which could denote the count of unique operations, both show moderate importance. These might reflect coding style or complexity measures that correlate with defect likelihood. The features complexity, v(g), IOComment, iv(g), l, ev(g), and locCodeAndComment show lower levels of importance. This doesn't mean they are irrelevant, but rather that their predictive power is less compared to the top-ranking features. complexity might measure overall software complexity, while v(g) could refer to a specific complexity metric like McCabe's cyclomatic complexity. IOComment might be the number of comments related to input/output operations, and iv(g) and ev(g) could represent internal and external visibility of components, respectively. The fact that l and locCodeAndComment (which may measure lines of code including comments) are ranked lower suggests that while they provide some information, other features better capture the underlying patterns related to software defects.

A confusion matrix lays out the number of correct and incorrect predictions of a model against the actual values. For a binary classification task, such as predicting software defects, the matrix consists of four components: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). TPs and TNs represent correct predictions of defects and non-defects, respectively, while FPs and FNs represent errors where non-defects are mistaken for defects and vice versa. In Figure 4, we observe a substantial number of TNs (8074), indicating that the model proficiently identifies the absence of defects. However, there are 487 instances of FPs, which, while relatively low, indicate cases where the model incorrectly predicted defects, potentially leading to unnecessary reviews or corrective actions. On the other hand, the model missed 1534 actual defects (FNs), which is critical as these represent defects that would remain undetected. The 973 TPs suggest that the model is relatively effective but not foolproof in identifying true defects.

Comparatively, Figure 5 shows a slight improvement in identifying TNs (8079) and a minor decrease in FPs (482), indicating a marginal enhancement in distinguishing non-defective cases. Nevertheless, this is countered by a slight increase in FNs (1540), meaning more actual defects have been overlooked by the model. Concurrently, there is a slight decrease in TPs (967), indicating fewer correct defect predictions compared to the first matrix. The contrast between the two matrices might seem minimal, but it’s significant in the context of model
performance and the implications of prediction errors. While both models are good at identifying non-defective cases, even a small number of FNs can be critical if defects lead to severe consequences. Therefore, these matrices provide essential insights into the model's reliability and can guide further tuning to balance the trade-off between identifying defects and avoiding false alarms. When deciding which model to deploy, one must consider the cost and risk associated with FP and FN, which vary depending on the specific application domain. For instance, in safety-critical software systems, reducing FNs may be prioritized over reducing FPs to ensure no defects go undetected, despite the increased cost of reviewing false alarms. The matrices serve as a foundation for refining models to align with these priorities, ensuring the chosen model offers the best balance for its intended application.

In this analysis, several key points emerge, highlighting the importance of feature selection and correlation in our models. Firstly, as shown in figures 2 and 3, the feature importance plots from both the LightGBM and XGBoost classifiers offer valuable insights into which features exert the most influence on the model's predictions. Commonly important features identified include 'loc' (lines of code), 'v(g)' (cyclomatic complexity), and 'I' (program length), all of which are well-established software metrics. Interestingly, while the feature 'i' (Halstead's Intelligence) stands out as the most significant feature in the LightGBM model, 'loc' holds more importance in the XGBoost model. Additionally, the heatmaps provide a visual representation of the correlation between different features. It's observed that features with high correlation might carry redundant information, which could be considered for removal to streamline the model. This is evident in the heatmaps presented in figure 1, where certain features, notably 'loc' and 'I' (program length), show strong positive correlations with several other metrics. This suggests that these features are central to the model's predictions and play a key role in their accuracy. Moreover, the correlation heatmaps also shed light on the relationship of features with the target variable 'defects'. Some features exhibit less correlation with 'defects', indicating that they might have less predictive power within the model. This information is crucial for refining the models and focusing on the most impactful features.

<table>
<thead>
<tr>
<th>Methods</th>
<th>ROC-AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LightGBM &amp; XGBoost (Voting Based)</td>
<td>0.8002</td>
</tr>
<tr>
<td>XGBoost</td>
<td>0.8006</td>
</tr>
</tbody>
</table>

The voting classifier, which combines the LightGBM and XGBoost models using a soft voting mechanism, achieved a ROC-AUC score of 0.8002. This score is a measure of the model's ability to distinguish between the classes (defective and non-defective modules). The closer the ROC-AUC score is to 1, the better the model is at making this distinction. A score of 0.8002 suggests that the voting classifier performs well, though there's room for improvement. On the other hand, XGBoost model, optimized using the best parameters from the hyperparameter tuning process, attained a slightly higher ROC-AUC score of 0.8006 compared to the voting classifier. This indicates that XGBoost, on its own, is very effective for this classification task. The confusion matrices for both classifiers provide insights into the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). For the voting classifier: Out of the total predictions, 8074 were TN, 973 were TP, 487 were FP, and 1534 were FN. In addition, for the XGBoost classifier, out of the total predictions, 8079 were TN, 967 were TP, 482 were FP, and 1540 were FN. Both classifiers have a similar number of correct predictions (TN + TP) and incorrect predictions (FP + FN). However, the XGBoost classifier has a slightly higher number of TN and a lower number of TP compared to the voting classifier.

4. CONCLUSION

This research embarked on an integrative journey, combining traditional software complexity metrics with advanced machine learning techniques to enhance software defect classification. Through meticulous data preprocessing, feature engineering, and hyperparameter optimization, two powerful classifiers, LightGBM and XGBoost, were trained and evaluated. Our findings indicate that both LightGBM and XGBoost classifiers perform robustly, with XGBoost showing a slight edge in predictive performance as indicated by the ROC-AUC scores. The confusion matrices and feature importance analyses further underscore the significance of combining traditional metrics with machine learning algorithms, revealing how each feature contributes to the classification task. The implications of this research are twofold. Firstly, it provides a clear demonstration of the efficacy of machine learning approaches in the realm of software defect classification. Secondly, it highlights the continued relevance of traditional software complexity metrics in informing these models. This dual approach enables a more nuanced understanding of software defects, potentially leading to better-informed quality assurance processes in software engineering. While the results are promising, further research is recommended to explore the integration of additional complexity metrics, investigate the impact of dataset size and diversity, and to test the generalizability of the models across different types of software systems. Future studies could also experiment with other machine learning algorithms and ensemble methods to further improve classification accuracy. In conclusion, this study contributes to the ongoing discourse on software quality assurance, providing evidence that machine learning
models, when appropriately integrated with domain-specific metrics, can offer substantial benefits in predicting and managing software defects.

REFERENCES


information,” in 2022 5th International Conference on Data Science and Information Technology (DSIT), 2022, pp. 1–5.
