ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

A Machine Learning-Based Framework for Software Defect Prediction Using Structural Software Metrics: An Empirical Analysis

Isaiah Ifeanyi Nweze[1]; Paul Maduabuchi Agu[2]; Ezekiel Nwibo Gabriel[3]; Charles Ugwute[4]; Chukwuka Abraham Nwovu[5]; Boniface Mbalaba Ofoke[6]

[1]Department of Computing University of Greater Manchester Bolton, United Kingdom
[2]Department of Robotics, Autonomous Systems and Telecommunications Engineering, University of Greater Manchester, Bolton, United Kingdom
[3]Department of Applied Computer Science University of Greater Manchester Bolton, United Kingdom
[4]Department of Computing University of Greater Manchester Bolton, United Kingdom
[5]Department of Computing University of Greater Manchester Bolton, United Kingdom
[6]Department of Cloud and Network Security University of Greater Manchester Bolton, United Kingdom

[1]Orcid ID: 0009-0002-7054-783X; [3]Orcid ID: 0009-0004-4656-2946

Publication Date: 2026/05/26

Abstract: Software defects are a critical challenge in modern software engineering systems due to the growing complexity of the structure and interdependence of objects in object-oriented system designs, which negatively affect the maintenance costs and reliability of the system. This paper introduces a machine learning-based system that predicts software defects using structural software metrics. A structured dataset of 145 software modules of 94 numerical features was analysed using six major metrics: Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Lack of Cohesion of Methods (LCOM), Number of Children (NOC), Response for Class (RFC) and Weighted Methods per Class (WMC). The statistical analysis indicated substantial variation and skewness in complexity-related measures, particularly CBO and RFC, suggesting the presence of structural outliers. Correlation analysis revealed a significant association among coupling, cohesion, and response measures, indicating that defect proneness is not affected by single factors but rather by interacting structural properties. The dataset had a moderate class imbalance, with most modules being non-defective. To compare the predictive performance, three supervised machine learning models (Decision Tree, Random Forest and Logistic Regression) were trained using a stratified 70:30 train-test split. The Random Forest model achieved the highest overall performance with an accuracy of 72.73%, precision of 75%, recall of 50%, and an F1-score of 0.60, reflecting a good balance between classification accuracy and generalisation. Logistic Regression was more precise but had less recall, whereas the Decision Tree model was less accurate but more interpretable. In general, the results show that statistical analysis and machine learning provide an efficient approach to early defect detection. The paper highlights the importance of structural complexity measures in identifying defect-sensitive modules. It advocates using an ensemble learning algorithm to enhance software quality assurance and overall system reliability.

Keywords: *Software Defect Prediction, Machine Learning, Random Forest, Decision Tree, Logistic Regression, Software Metrics, Classification.*

How to Cite: Isaiah Ifeanyi Nweze; Paul Maduabuchi Agu; Ezekiel Nwibo Gabriel; Charles Ugwute; Chukwuka Abraham Nwovu; Boniface Mbalaba Ofoke (2026) A Machine Learning-Based Framework for Software Defect Prediction Using Structural Software Metrics: An Empirical Analysis. *International Journal of Innovative Science and Research Technology*, 11(5), 1642-1652. https://doi.org/10.38124/ijisrt/26may670

I. INTRODUCTION

The prediction of software defects has become an indispensable field of study in software engineering due to the growing complexity, scope, and urgency of modern software systems [1], [2]. As software applications are increasingly deployed in areas such as healthcare, finance, transportation, and telecommunications, reliability and defect reduction have become core concerns for keeping systems intact and maintaining user confidence. Software defects, also known as bugs, are faults or errors in code that can lead to unexpected behaviour, system failures, security vulnerabilities, and high

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

operational costs. Early detection and correction of such defects are thus key to high-quality software systems and to reduced maintenance costs in the post-deployment phase [3].

In a large-scale software system, defects are not usually accidental; they are more likely to reflect the software's underlying structural and design properties [4]. Modern software development, dominated by object-oriented programming, introduces sophisticated interdependencies among classes and modules through concepts such as inheritance, coupling, and polymorphism. Although these characteristics contribute to modularity and reuse, they can lead to greater defect propagation when the features are not managed properly. Some of the most well-established metrics for revealing software complexity and the propensity to defect are Coupling Between Objects (CBO), Lack of Cohesion of Methods (LCOM), Depth of Inheritance Tree (DIT), and Response for Class (RFC) [5], [6]. For example, tight coupling and low cohesion are commonly associated with more interdependent components that are harder to test, maintain, and change, increasing the likelihood of defects.

Manual code inspection and static testing are traditional methods commonly used in the software development process. These approaches, however, tend to be labour-intensive and time-consuming, and they fail to scale to large codebases. Also, they address defects that have already been introduced into the system rather than anticipating them. Such a reactive process will result in higher development costs and longer project schedules, especially in a complex system where bugs can propagate across several modules [7].

To address these drawbacks, machine learning has become an effective tool for predictive software quality assurance [8]–[12]. Machine learning models can identify hidden patterns and relationships that conventional analytical approaches cannot detect, leveraging historical software metrics and defect data. These models facilitate the categorisation of software modules as defective or non-defective, allowing developers to focus on more important testing tasks and use resources more effectively. Methods such as Decision Trees, Random Forests, and Logistic Regression have become widely used in defect prediction research because they can handle structured data and capture complex feature interactions [13]–[17].

Software defect prediction using machine learning involves several critical steps, including data preprocessing, feature selection, model training, and performance evaluation. Data preprocessing makes the dataset clean, consistent, and analysable, whereas feature selection can be used to identify the most useful measures that affect the occurrence of defects [6]. The model training process is based on learning patterns from historical data, and predictive performance is assessed using accuracy, precision, recall, and F1-score. Among those, ensemble methods like Random Forest have been of particular interest because of their strength and ability to minimise overfitting. In contrast, Decision Trees are highly interpretable, which makes them useful for studying decision-making paths.

Regardless of the benefits of machine learning methods, there are several issues with software defect prediction. The existence of an unbalanced dataset, in which non-defective modules far outnumber defective ones, is a primary problem, as it leads to biased model predictions [18]–[21]. The second difficulty is that the relationships among software metrics are nonlinear and complex, thereby limiting the usefulness of simple models such as Logistic Regression [22]–[24]. In addition, the distribution of software metrics, such as skewness and outliers, might affect model performance and generalisation.

The current research is based on a data-oriented approach to analysing software defect prediction using machine learning. The research will be conducted by examining a systematic dataset of software metrics, detecting trends related to modules at risk of defects, and evaluating the performance of various classification models. The paper combines statistical techniques, exploratory data visualisation, and supervised machine learning to develop a predictive model for detecting defects at their early stages [1]. The approach is not limited to existing reactive testing mechanisms; it enables proactive testing of high-risk modules based on their structural properties.

The relevance of this study lies in its potential to enhance quality assurance in software development through smart, fact-based decision-making. The study also enhances the reliability, maintainability, and overall quality of the software system by identifying primary predictors of defect occurrence and of model performance. Moreover, the results can be useful to software engineers and project managers, helping them develop more effective testing strategies and resource allocation practices.

In summary, software defect prediction is a particularly important intersection of software engineering and data science, where state-of-the-art analytical methods and tools can be applied to address longstanding issues in software quality control. Using statistical analysis in combination with machine learning models, the work offers a unified approach to the behaviour of defects in complex software systems and prediction.

II. RESEARCH METHODOLOGY

This study adopted a quantitative research approach. It is a data-driven methodology that relies on a structured secondary dataset of software metrics comprising 145 software modules and 94 numerical features. These features are object-oriented design characteristics, such as coupling, cohesion, inheritance depth, and class complexity. The target variable is a binary indicator of whether a module is defective. The dataset is representative of software systems in which defects arise from structural properties rather than by chance. The dataset is available at;

https://www.kaggle.com/datasets/iribaboci/software-defect-prediction. It consists of object-oriented software metrics, and complexity-related and size-related metrics (Halstead metrics, number of operators and operands, lines of

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

code (LOC)) that were also included to represent computational complexity and structural attributes of software modules.

The adopted research methodology provides an extensive analytical framework for assessing software defect prediction using machine learning methods. The complexity of software systems and the multifactorial nature of defect occurrence indicated that the methodological approach was specifically designed to incorporate statistical analysis, exploratory data modelling, and supervised machine learning into a single computational workflow [1]. The methodology was also developed to assess historical defect trends and create a predictive model to detect defect-prone modules in software systems. The Python programming language, with a Jupyter Notebook environment, was used to implement all analytical procedures, thereby guaranteeing the reproducibility, transparency, and scalability of the results.

The methodology began with data acquisition and preprocessing. Previously, the data was in the Attribute-Relation File Format (ARFF), which was imported into the Python environment and converted to a tabular format using the Pandas library. This step involved converting the categorical class labels into numerical ones (1: defective, 0: non-defective) to support machine learning modelling. The data cleaning procedures were used to address inconsistencies, including missing data, standardise data types, and check numerical integrity. To avoid extreme distortion of model performance caused by missing values, median imputation was used to preserve distributional properties and reduce bias.

The next step after preprocessing was the exploratory data analysis phase, aimed at gaining insight into the statistical characteristics and distribution patterns of the software metrics. Mean, median, standard deviation, and range were calculated as descriptive statistics to establish baseline variability across features. Histograms and boxplots were used as visualisation methods to point out skewness, outliers, and extreme values in the dataset. These analyses indicated that several complexity-related metrics were highly skewed, implying that structurally complex modules might have a disproportionate contribution to defect occurrence. To analyse relationships among software metrics, a correlation analysis was conducted to assess multicollinearity [14]. The Pearson correlation coefficients were computed and visualised as a correlation heatmap, enabling identification of highly correlated features, including the coupling and response metrics. Knowledge of these relations played a vital role in interpreting model behaviour and in preventing predictive performance from being artificially enhanced by redundancy. A stratified sampling approach was then used to partition the dataset into a training and testing group to preserve the initial class distribution.

A 70:30 ratio was used, yielding 101 and 44 samples for training and testing, respectively. The stratification was especially significant because of the moderate imbalance in the classes, with respect to the number of defects and non-defects, that was reasonable in the dataset and hence in both subsets [18], [19], [25]. The modelling stage involved applying three supervised machine learning models: Decision Tree, Random Forest, and Logistic Regression. The Decision Tree classifier was chosen because it is currently interpretable and can model nonlinear relationships using hierarchical decision rules. The ensemble learning method known as Random Forest, which enhances predictive accuracy and minimises overfitting, was employed. Logistic Regression was included as a baseline linear model to compare model performance under the linear hypothesis.

The Scikit-learn library was used to train the model, with default and optimised parameters used where necessary. Logistic Regression was applied with feature scaling based on standardisation to address convergence issues and stabilise optimisation. All the models were trained on the training set and tested on the testing set to measure generalisation. The machine learning models were optimised using grid search and cross-validation to improve model performance and generalisation.

The evaluation of the models was based on various performance measures, such as accuracy, precision, recall, and F1-score. The metrics are used to provide a comprehensive evaluation of classification performance, especially in the presence of class imbalance [6], [26]. Accuracy measures overall correctness, while precision and recall measure the model's ability to detect faulty modules. The F1-score provides a balanced measure of accuracy and recall, making it especially useful for defect prediction.

In addition to numerical analysis, a confusion matrix analysis was conducted to visualise the classification results and identify patterns of misclassification. The analysis provided insights into false positives and false negatives, which are essential for understanding the practical implications of the defect prediction models. Moreover, the discriminative ability of the models was evaluated using Receiver Operating Characteristic (ROC) curves and Area Under the Curve (AUC) values; a larger AUC indicates better classification performance [4].

Feature importance was analysed using the Random Forest model to identify the most important predictors of defect occurrence. The analysis showed that metrics related to coupling and cohesion informed model decisions, supporting the theoretical explanation of software complexity as a major source of defects [6].

The modelling process was also validated through a critical assessment of model behaviour and the consistency of model performance, thereby ensuring robustness and reliability. Ensemble methods and control of model complexity helped mitigate overfitting [13]. The entire analysis pipeline was to be reproducible, with all preprocessing, modelling, and evaluation steps recorded and executed in a uniform computational environment.

Ethical issues were minimal, as the research was based on publicly available software metrics data without any personal or sensitive information. Data integrity and

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

methodological transparency were, however, strictly adhered to ensure the validity and reproducibility of the findings.

In general, the research methodology combines statistical rigour, exploratory data analysis, and machine learning within a single framework for software defect prediction. Through the integration of these techniques, the research goes beyond descriptive analysis to offer a forecasting and practical model to aid proactive software quality management. The methodology shows how current-day data-driven approaches can transform traditional software testing into an intelligent, predictive approach to defect prevention and enhanced software reliability.

III. DATA ANALYSIS AND INTERPRETATION

The statistical distribution of software measures provides a broad baseline understanding of the structural variability in the data and how it can be used to predict defect proneness. The descriptive statistics shown in Table I indicate that several software metrics are highly variable, especially those related to system complexity and module interactions. An example is Coupling Between Objects (CBO), which has a mean value of 32.45 with a large standard deviation (45.21). This implies that most modules have comparatively low coupling, but a few modules have very high dependencies. Likewise, Response for Class (RFC) has a mean of 75.32 and a standard deviation of 88.17, indicating a wide range in the complexity of method invocation among classes. CBO (210) and RFC (250) have the highest values that are extreme outliers in the dataset.

Table 1 Descriptive Statistics of Key Software Metrics

Metric	Mean	Std Dev	Min	Q1	Median	Q3	Max
CBO	32.45	45.21	0	5	12	45	210
RFC	75.32	88.17	3	20	40	90	250
WMC	15.67	20.44	1	3	8	20	120
LCOM	0.42	0.28	0	0.15	0.40	0.70	1.0

The large interquartile ranges also serve as an additional indicator that both simple and highly complex modules are represented in the dataset, and suggest that the defect risk is skewed. In software engineering terms, these observations support the well-known theoretical assumptions. High coupling increases interdependency between classes, thereby amplifying the effects of code changes. Low cohesion, as indicated by higher LCOM values, means that the structure of classes is poorly organised and may have unrelated responsibilities. The two attributes are widely recognised as significant factors in defect formation [5].

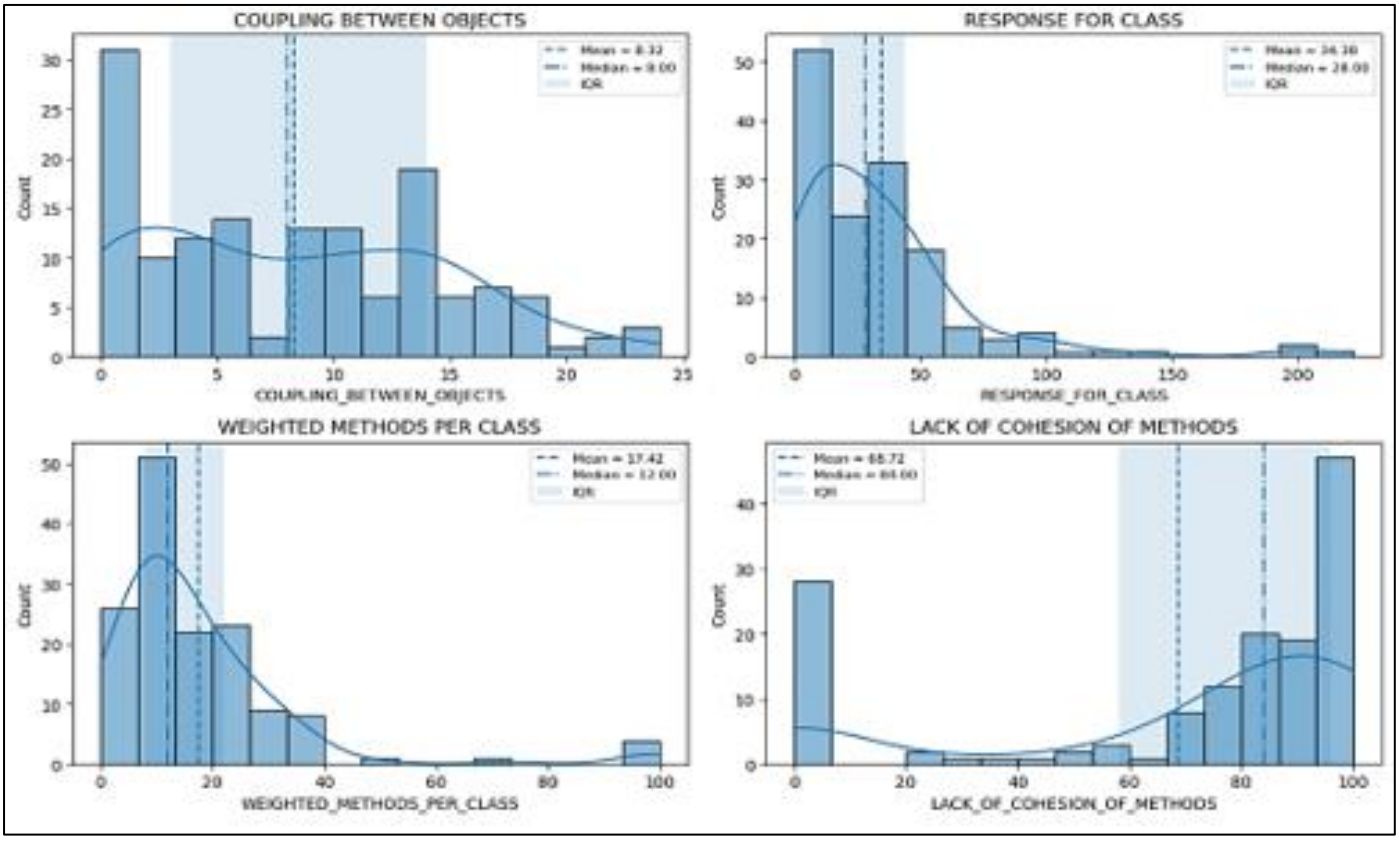

Fig 1 Statistical Distribution of Key Software Metrics

Fig. 1 presents a more detailed statistical description of the major software measures, including Coupling Between Objects (CBO), Response for Class (RFC), Weighted Methods per Class (WMC), and Lack of Cohesion of Methods (LCOM). The multi-panel display allows simultaneous analysis of central tendency, dispersion and distributional shape of key structural attributes of the software system.

The distributions of CBO, RFC, and WMC are heavily right-skewed, and most modules have moderate complexity,

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

though some have extremely high metric values. Indicatively, RFC has a mean value of 75.32, indicating the presence of high-value outliers. These outliers suggest modules with high method interactions and more complex responses.

WMC shows a mean of 15.67 and a median of 8, indicating moderate complexity with higher-value outliers. These modules are usually linked to the increased cognitive burden and vulnerability.

LCOM exhibits moderate variability, with a mean of 0.42, indicating varying cohesion levels across modules. This implies that a high percentage of modules are poorly cohesive, resulting in fragmented class design and reduced modular integrity.

The shaded interquartile ranges (IQRs) in all subplots indicate the spread of the central 50% of observations, and it can be affirmed that defect-prone properties are not evenly distributed but are concentrated in specific parts of the data.

In general, the statistical analysis of the distribution provides substantial empirical support for the conclusion that structural imbalance, characterised by high coupling, complexity, and low cohesion, is common in the dataset and likely plays an important role in defect occurrence.

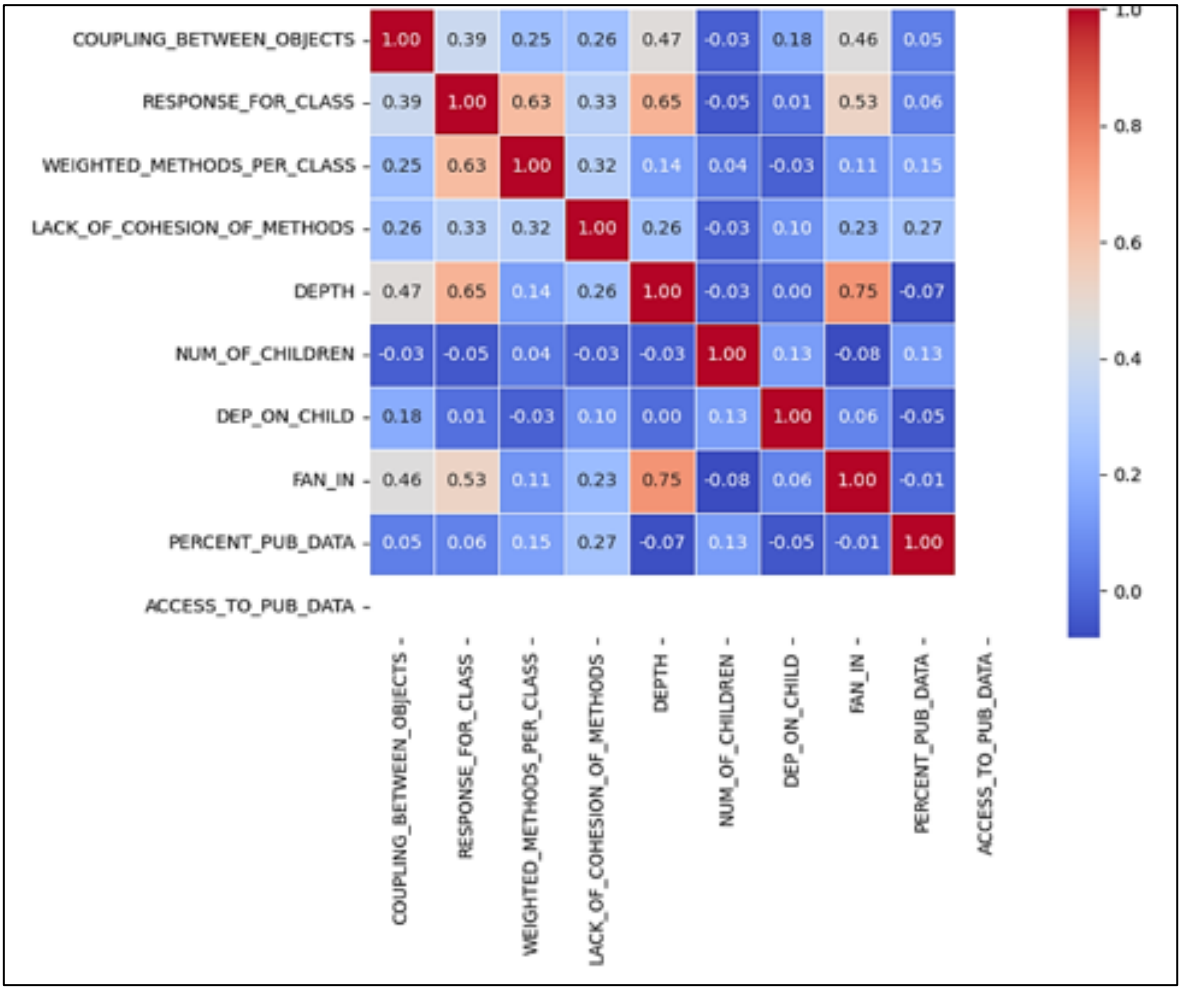

Fig 2 Correlation Matrix of Selected Software Metrics

Fig. 2 illustrates the correlations among the selected software measures using a correlation matrix, providing insight into how structural attributes interact to form defects. As observed in the heatmap, some positive correlations are strong, notably between RFC and WMC (0.63) and between RFC and DEPTH (0.65), indicating that more complex modules in terms of methods tend to have deeper inheritance structures.

There is a particularly close correlation between DEPTH and FANIN (0.75), and it is possible to assert that deeply nested classes are also seriously used by other modules. This structural property enhances the system's interdependence and may increase the spread of defects among components.

The intermediate relationships between CBO and RFC (0.39) and between CBO and FANIN (0.46) further suggest that the central role of coupling is to define the system's complexity. These results suggest that highly coupled modules are largely characterised by a high degree of interaction and dependency, making them more prone to defects.

On the other hand, some metrics, such as NUMOFCHILDREN and DEPONCHILD, show weak or near-zero correlations with other variables, indicating that not

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

all structural attributes play an equal role in predicting defects [14], [4].

The general correlation pattern indicates that software defects result from the joint effect of multiple interacting metrics, rather than from the actions of single factors. This underscores the need to apply ensemble learning methods capable of capturing complex nonlinear relationships among features.

Table 2 Performance Comparison of Machine Learning Models

Model	Accuracy	Precision	Recall	F1-Score
Decision Tree	0.6136	0.5333	0.4444	0.4848
Random Forest	0.7273	0.7500	0.5000	0.6000
Logistic Regression	0.7273	0.8750	0.3889	0.5385

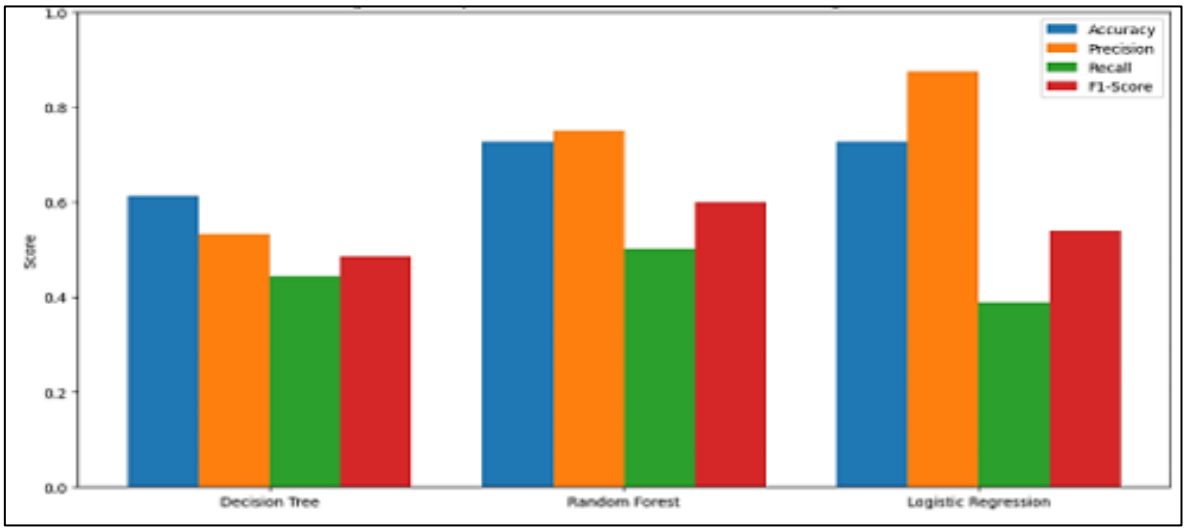

Fig 3 Comparative Performance of Machine Learning Models

Fig. 3 demonstrates a comparative analysis of three classification models, which include Decision Tree, Random Forest, and Logistic Regression, based on four performance measures, which are accuracy, precision, recall, and F1-score. The overall performance of the Random Forest model is better, with an accuracy of around 0.73, a precision of 75%, and a high F1 Score compared to the other tested models. This indicates that the model strikes the right balance between detecting defective modules and minimising false forecasts.

The best precision (0.88) is observed with Logistic Regression, indicating the model's high reliability in predicting defective modules. Its relatively low recall (0.39) indicates that it cannot identify a large portion of real defects, reducing its usefulness in practice for applications where defect detection is urgent. The Decision Tree model demonstrates relatively poor results across all measures but offers a useful degree of interpretability, allowing one to see the decision rules clearly. The findings indicate that the ensemble learning technique, especially Random Forest, proves quite beneficial for managing intricate, high-dimensional data. This makes it predictive, as it can combine several decision trees, thereby capturing complex interactions among features [13], [8].

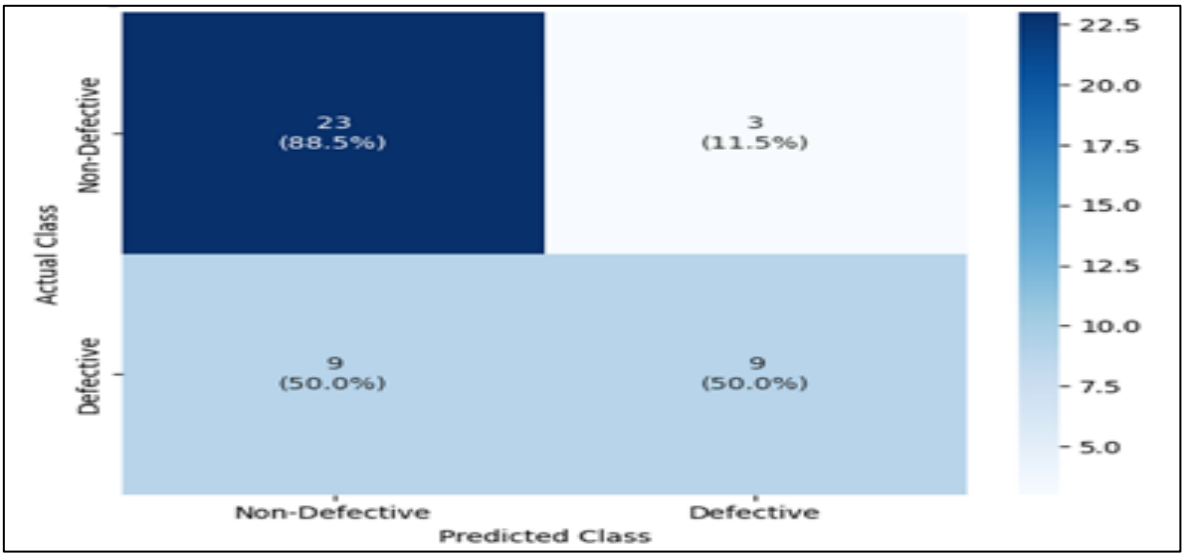

Fig 4 Confusion Matrix of Random Forest Predictions

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

Fig. 4 presents a detailed classification result of the Random Forest model. The matrix indicates that 23 non-defective modules (88.5%) were correctly classified, with only 3 (11.5%) misclassified as defective. For faulty modules, the model is correct in 9 cases (50%) and wrong in 9 cases. This indicates that the model is sensitive to non-defective modules, but its sensitivity to defective modules is moderate. False negatives are particularly significant because they represent defective modules that go undetected. In practical software systems, these errors may cause system failures, increased maintenance costs, and reduced reliability. Despite this shortcoming, the model has reasonable precision and recall, which is why it can be deployed in practice and has potential for optimisation.

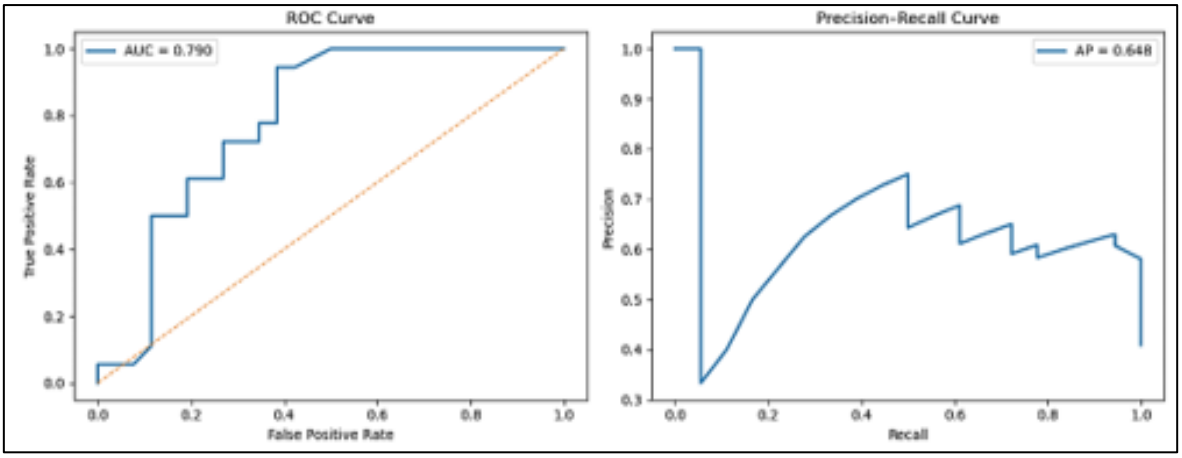

Fig 5 Classification Discrimination Performance of Random Forest

Fig. 5 shows the ROC and Precision-Recall curves for the Random Forest model, providing a detailed analysis of classification performance. The Area Under the Curve (AUC) of the ROC curve is approximately 0.79, indicating good discriminatory ability between defective and non-defective modules. The curve is quite high relative to the diagonal baseline, suggesting that the model performs better than random classification. Another implication of the Precision-Recall curve is that the model is effective at addressing class imbalance. With an average precision (AP) of around 0.65, the model shows plausible performance for maintaining precision while increasing recall. Combining these curves provides a complementary understanding of the model's performance, and it can be concluded that the Random Forest model effectively reproduces the underlying patterns in the dataset [7].

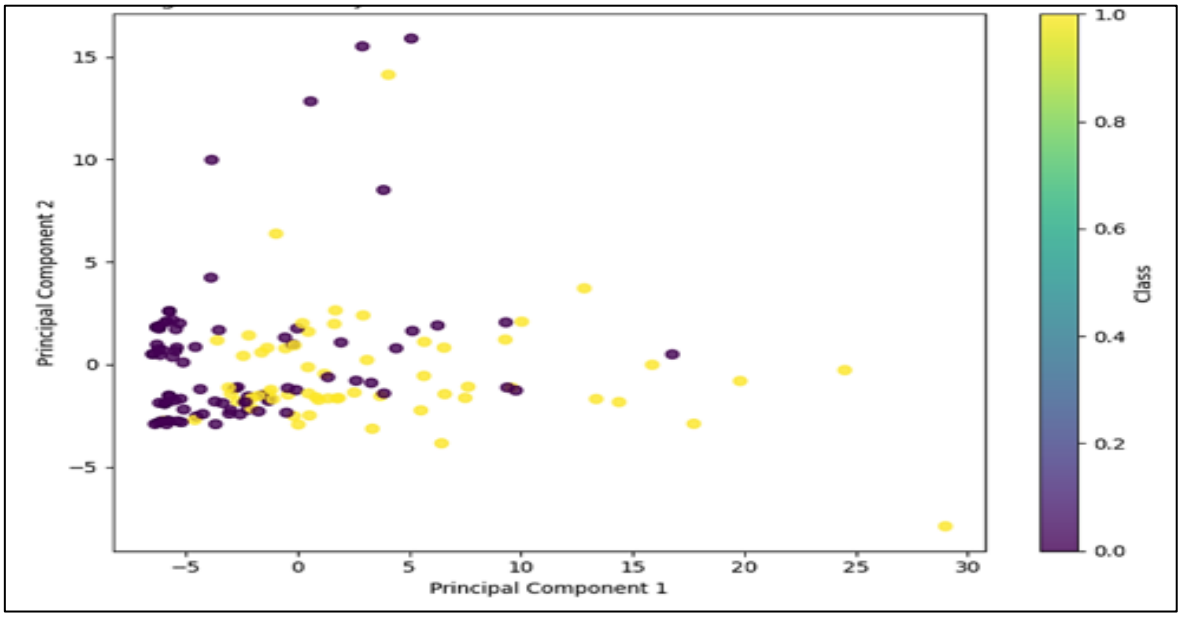

Fig 6 PCA Projection of Defective and Non-Defective Modules

Fig. 6 shows a two-dimensional projection of the data using Principal Component Analysis (PCA), which allows visualising the ability to separate the classes. A scatter plot shows semi-clustering between defective and non-defective modules, indicating meaningful structural differences among the selected modules. The intersection between the classes indicates that there is always a complicated nature of defect prediction and that it cannot be purely linearly separable. This supports the essence of nonlinear models like Random Forest to learn the underlying trends.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

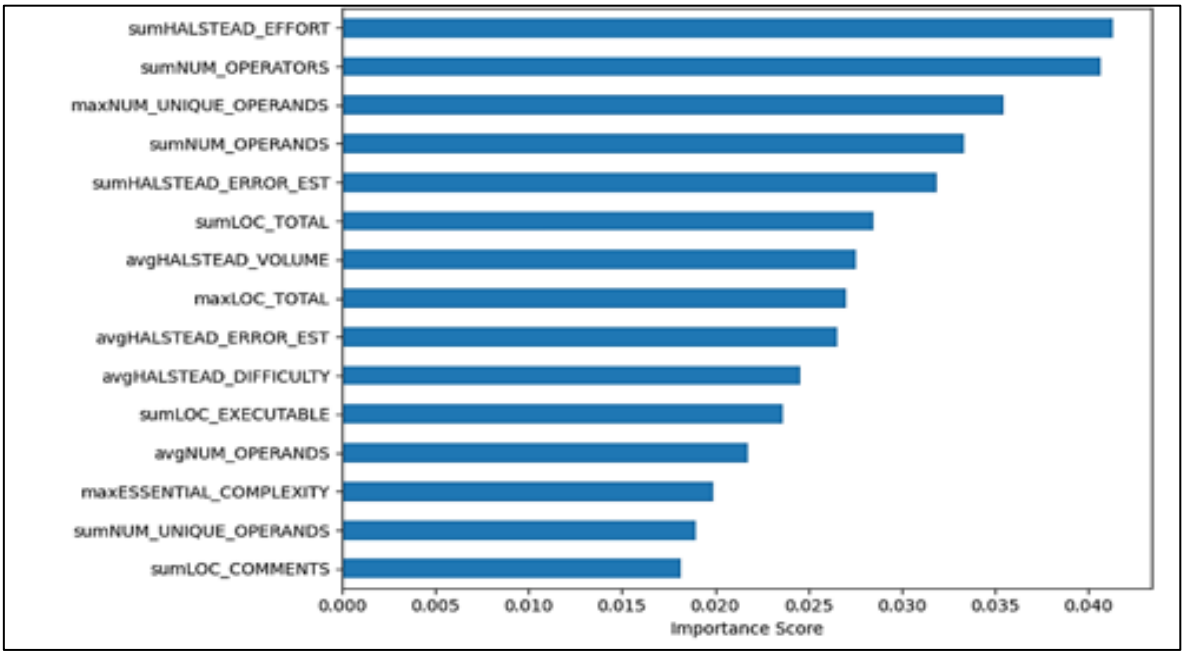

Fig 7 Top Predictors of Software Defect Occurrence

Fig. 7 illustrates the relative importance of features as predictors of software defects using the Random Forest model. The findings show that Halstead complexity measures, especially Halstead effort, the number of operators, and the number of unique operands, are among the strongest predictors of defect occurrence. The leading role of Halstead's effort implies that computational complexity is a crucial factor in the process of defect proneness. Modules with increased computational intensity are more complex to maintain, test, and comprehend, and thus have a greater potential for defects. Also, the structural and logical complexity of the code, as measured by metrics such as the number of operators and operands, can increase vulnerability to errors.

Moreover, measures of size, including Lines of Code (LOC), total LOC, and executable LOC, are highly significant. This implies that larger modules are more likely to have defects due to implementation complexity and lower maintainability. The prominence of Halstead error-estimation metrics further supports the relationship between computational complexity and defect occurrence, indicating that mathematically derived complexity measures can be useful predictors of software quality risks. In general, the results show that computational complexity and code size are major factors in determining the occurrence of defects. In practical terms, the results indicate that developers ought to pay attention to minimising code complexity, reducing extraneous operations, and ensuring manageable module sizes to enhance software quality and minimise the risk of defects [6], [27].

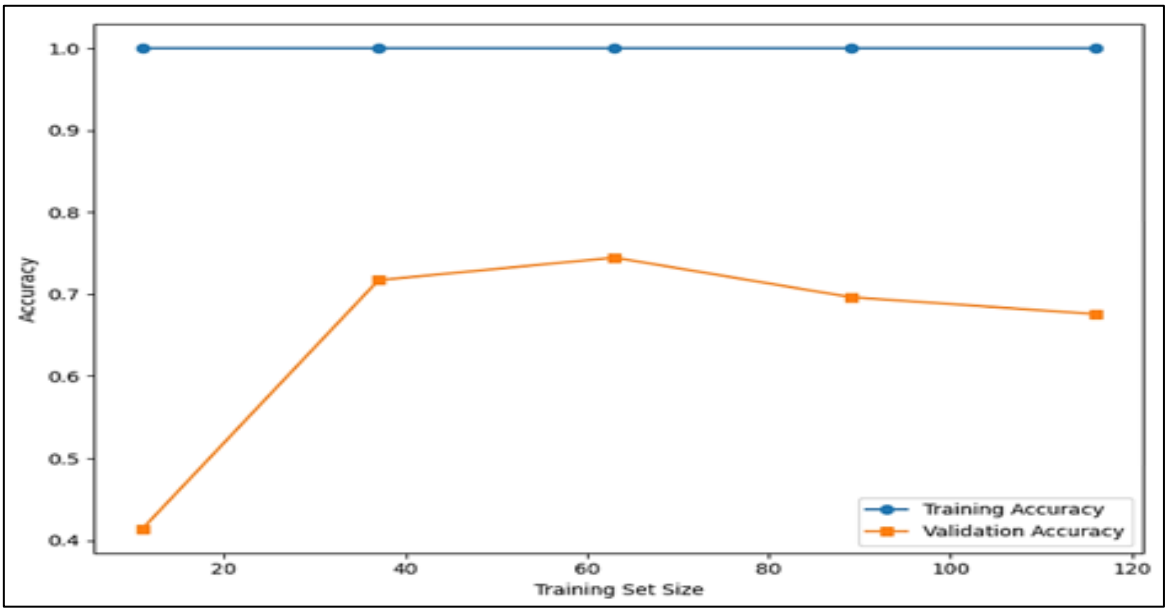

Fig 8 Learning Curve of Random Forest Model

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

Fig. 8 shows the learning curve of the Random Forest model, illustrating the relationship between training set size and model performance. The training accuracy is always high (around 1.0), whereas the validation accuracy reaches a steady state of about 0.70. The difference between the training and validation curves indicates the model's tendency to overfit the training data, which leads to poorer generalisation to unseen data. The comparatively consistent validation results indicate that the model has satisfactory predictive power. This indicates that potential solutions to this issue include balancing model complexity and generalisation, especially in high-dimensional data.

IV. DISCUSSION OF FINDINGS

The results of this research provide important insights into the structure of factors that affect the occurrence of software defects and the usefulness of machine learning models for predicting defect-prone modules. The combination of statistical analysis, correlation evaluation, and predictive modelling provides an in-depth insight into the role of software complexity and interdependencies in system reliability issues.

The statistical analysis of the distribution revealed substantial variation in the main software metrics, including CBO, RFC, and WMC. A high degree of skewness and extreme outliers suggest that defect-prone characteristics are concentrated in a small number of extremely complex modules. This is in line with the cut-and-dried theories of software engineering that hold that highly coupled, intricate modules are more challenging to maintain and thus more prone to defects [5], [4].

The correlation analysis also showed that the software metrics are not independent and are rather strongly correlated with each other. The relationships among measures such as RFC, WMC, and DEPTH, as observed, show the interdependence of software design attributes. These results indicate that the presence of a defect results from the interplay of several structural elements rather than a single metric, supporting the need for multivariate modelling techniques [14].

The comparative analysis of machine learning models revealed that the Random Forest classifier outperformed the Decision Tree and Logistic Regression models [13], [4]. Its high performance stems from its use of an ensemble method, which enables it to capture nonlinear relationships and interactions among features. Although Logistic Regression was very precise, it had low recall, indicating it is not effective at detecting defective modules and thus is not well-suited for scenarios where defect detection is critical.

The confusion matrix analysis showed that although the Random Forest model is effective at detecting non-defective modules, it has moderate sensitivity in detecting defective ones. False negatives are also an indicator of a main limitation, and an undetected defect may have a huge impact on real-world software systems [19], [20]. It can be inferred that additional streamlining of the model, including hyperparameter optimisation or cost-sensitive training, can be required to enhance recall.

The ROC and Precision-Recall results confirmed the strength of the Random Forest model, with an AUC of approximately 0.79 and a mean average precision of 0.65. These findings indicate strong classification performance, and the model performs well on imbalanced datasets.

The PCA figure further helped to understand that the defective and non-defective modules could be separated. Although some clustering is evident, the spread between classes indicates that defect prediction is a complex task and cannot be based on simple linear boundaries. This is another reason why ensemble and nonlinear models are used.

The analysis of the feature importance identified the supremacy of Halstead complexity measures and code size metrics in predicting defect occurrence. The findings emphasise the need to control the computational complexity and continue with modular design in software development.

Lastly, the learning curve analysis showed a discrepancy between training and validation performance, indicating some degree of overfitting. Even though the model retains a reasonable level of generalisation, this highlights the need to strike a balance between model complexity and generalisation.

In general, the findings reveal that software defect prediction is a complex issue that involves statistical analysis, feature engineering, and sophisticated machine learning methods. The results align with the available literature that prioritises the importance of structural complexity and coupling in the occurrence of defects [6], [4]. The findings support the need to consider multiple interacting software measures rather than single variables when predicting defects.

V. CONCLUSION

This paper examined how machine learning can be used to predict software defects based on software metric data. A thorough analysis of the problem, using statistical characterisation, correlation analysis, and a predictive model, shows that structural complexity and interdependent design attributes significantly affect software defects. The findings show that measures such as coupling, response complexity, and cohesion are significant in determining defect proneness. Skewed distributions and extreme values indicate that defects are concentrated in highly complex modules, and specific quality assurance measures should be developed. The best-performing model was the Random Forest, which achieved the best overall results, indicating it is well-suited to capturing intricate relationships in the data. The ROC and Precision-Recall analysis results, together with the model performance, confirm the model's appropriateness for real-world defect prediction tasks.

Although these are encouraging findings, the study is also characterised by limitations, including moderate recall of defective modules and overfitting. These problems suggest that predictive models need to be further optimised and

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

refined. Finally, statistical analysis and machine learning can be integrated into a powerful framework for predicting software defects. By identifying high-risk modules early in the development process, organisations can enhance software quality, minimise maintenance costs, and improve system reliability. The Random Forest model performed best, with an accuracy of 72.73% and an AUC of approximately 0.79, indicating strong predictive ability. The proposed framework can be incorporated into the software development process to assist in detecting defects early and enhancing overall software quality assurance.

RECOMMENDATIONS

Based on the findings of this research, the following recommendations are offered to enhance software defect prediction and overall software quality.

- Software development teams should prioritise the focus on testing and code review of modules that have high degrees of coupling and complexity.
- To facilitate earlier detection of defect-prone modules, organisations ought to incorporate predictive models, especially ensemble methods like the Random Forest, in their development processes.
- Future work should explore techniques such as hyperparameter tuning and class balancing methods (e.g., SMOTE) to improve recall and reduce false negatives.
- There should be more features added, such as process metrics and developer activity data, to enhance predictive accuracy.
- Models must also be tested over many datasets and software projects to enhance generalizability.
- The development practice to be used by developers must reduce the risk of coupling and enhance cohesion, which will reduce defect risk.

LIMITATIONS OF THE STUDY

This research is informative on software defect prediction, but several limitations should be noted.

First, the dataset used in this study is based on a single software project, which may limit the generalizability of the results to other applications or development domains. The structural characteristics of different software systems may also vary, affecting defect prediction performance.

Second, the analysis is based mainly on code metrics by design, such as coupling, cohesion, and complexity. Although the metrics are informative, they fail to capture dynamic or process-relevant aspects, such as developer behaviour, code churn, or version history, which can also be important causes of defects.

Third, despite the Random Forest model's high performance, there are some false negatives, meaning some faulty modules go undetected. This restriction reveals the need to optimise the model further and use more advanced methods.

Lastly, the dataset has moderate class imbalance, which can affect the predictive power of some models. In addition to evaluation metrics like Precision-Recall curves, other methods, such as resampling or cost-sensitive learning, may be employed to further improve results.

FUTURE WORK

This study can be extended in several ways by future research.

First, the application of process measures, such as code churn, developer activity, and code commit history, might yield better predictive performance because they capture dynamic aspects of the software development process.

Second, more sophisticated machine learning models, such as deep learning and hybrid ensemble models, could be considered to enhance classification accuracy and robustness. Also, explainable AI methods can enhance model interpretability and provide deeper insights into the decision-making process.

Third, cross-project validation must be performed to determine how many other software systems and domains the proposed approach can be applied to. This would make the model more practical for real-world applications.

Lastly, defect prediction models must be integrated into continuous integration and deployment (CI/CD) pipelines to enable real-time monitoring and defect control.

REFERENCES

[1]. S. Stradowski and L. Madeyski, "Machine learning in software defect prediction: A business-driven systematic mapping study," Inf. Softw. Technol., vol. 155, p. 107128, 2023.
[2]. N. Grattan, D. A. da Costa, and N. Stanger, "The need for more informative defect prediction: A systematic literature review," Inf. Softw. Technol., vol. 171, p. 107456, 2024.
[3]. A. Alsaeedi and M. Z. Khan, "Software defect prediction using supervised machine learning and ensemble techniques: A comparative study," J. Softw. Eng. Appl., vol. 12, no. 5, pp. 85–100, 2019.
[4]. C. Zhou, P. He, C. Zeng, and J. Ma, "Software defect prediction with semantic and structural information of code based on graph neural networks," Inf. Softw. Technol., vol. 152, p. 107057, 2022.
[5]. A. O. Balogun, S. Basri, S. J. Abdulkadir, and A. S. Hashim, "Performance analysis of feature selection methods in software defect prediction: A search method approach," Appl—Sci., vol. 9, no. 13, p. 2764, 2019.
[6]. M. Ali, T. Mazhar, A. Al-Rasheed, T. Shahzad, Y. Y. Ghadi, and M. A. Khan, "Enhancing software defect prediction: A framework with improved feature selection and ensemble machine learning," PeerJ Comput—Sci., vol. 10, p. e1860, 2024.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26may670

[7]. S. K. Pandey and A. K. Tripathi, "An empirical study toward dealing with noise and class imbalance issues in software defect prediction," Soft Comput., vol. 25, no. 21, pp. 13465–13492, 2021.
[8]. G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," J. Syst. Softw., vol. 195, p. 111537, 2023.
[9]. R. van Dinter, C. Catal, G. Giray, and B. Tekinerdogan, "Just-in-time defect prediction for mobile applications: Using shallow or deep learning?" Softw. Qual. J., vol. 31, no. 4, pp. 1281–1302, 2023.
[10]. Z. M. Zain, S. Sakri, and N. H. A. Ismail, "Application of deep learning in software defect prediction: Systematic literature review and meta-analysis," Inf. Softw. Technol., vol. 158, p. 107175, 2023.
[11]. M. M. Jouybari, A. Tajary, M. Fateh, and V. Abolghasemi, "A novel deep neural network structure for software fault prediction," PeerJ Comput—Sci., vol. 10, p. e2270, 2024.
[12]. C. M. Liapis, A. Karanikola, and S. Kotsiantis, "Data-efficient software defect prediction: A comparative analysis of active learning-enhanced models and voting ensembles," Inf. Sci., vol. 676, p. 120786, 2024.
[13]. H. Alsawalqah, N. Hijazi, M. Eshtay, H. Faris, A. A. Radaideh, I. Aljarah, and Y. Alshamaileh, "Software defect prediction using heterogeneous ensemble classification based on segmented patterns," Appl: Sci., vol. 10, no. 5, p. 1745, 2020.
[14]. C. Liu, D. Yang, X. Xia, M. Yan, and X. Zhang, "A two-phase transfer learning model for cross-project defect prediction," Inf. Softw. Technol., vol. 107, pp. 125–136, 2019.
[15]. A. Alazba and H. Aljamaan, "Software defect prediction using stacking generalisation of optimised tree-based ensembles," Appl. Sci., vol. 12, no. 9, p. 4577, 2022.
[16]. A. N. Babatunde, R. O. Ogundokun, L. B. Adeoye, and S. Misra, "Software defect prediction using DAGging meta-learner-based classifiers," Mathematics, vol. 11, no. 12, p. 2714, 2023.
[17]. A. Daza, "Software defect prediction based on a multiclassifier with hyperparameters: Future work," Results Eng., vol. 25, p. 104123, 2025.
[18]. K. K. Bejjanki, J. Gyani, and N. Gugulothu, "Class imbalance reduction (CIR): A novel approach to software defect prediction in the presence of class imbalance," Symmetry, vol. 12, no. 3, p. 407, 2020.
[19]. S. Feng, J. Keung, X. Yu, Y. Xiao, and M. Zhang, "Investigation on the stability of SMOTE-based oversampling techniques in software defect prediction," Inf. Softw. Technol., vol. 139, p. 106662, 2021.
[20]. S. Goyal, "Handling class-imbalance with KNN (neighbourhood) under-sampling for software defect prediction," Artif. Intell. Rev., vol. 55, no. 3, pp. 2023–2064, 2022.
[21]. S. R. Goyal, "A systematic review on AI-based class imbalance handling in software defect prediction," Results Eng., p. 106578, 2025.
[22]. Z. Xu et al., "LDFR: Learning deep feature representation for software defect prediction," J. Syst. Softw., vol. 158, p. 110402, 2019.
[23]. R. Malhotra and S. Priya, "DHG-BiGRU: Dual-attention based hierarchical gated recurrent unit model for software defect prediction," Inf. Softw. Technol., vol. 178, p. 107600, 2025.
[24]. M. Nashaat and J. Miller, "Refining software defect prediction through attentive neural models for code understanding," J. Syst. Softw., vol. 220, p. 112266, 2025.
[25]. Y. Tang, Y. Zhou, C. Yang, Y. Du, and M. S. Yang, "An instance gravity oversampling method for software defect prediction," Inf. Softw. Technol., vol. 179, p. 107657, 2025.
[26]. B. Arasteh, K. Arasteh, A. Ghaffari, and R. Ghanbarzadeh, "A new binary chaos-based metaheuristic algorithm for software defect prediction," Cluster Comput., vol. 27, no. 7, pp. 10093–10123, 2024.
[27]. F. Yang, G. Zeng, F. Zhong, P. Xiao, W. Zheng, and F. Qiu, "CfExplainer: Explainable just-in-time defect prediction based on counterfactuals," J. Syst. Softw., vol. 218, p. 112182, 2024.