International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 9, Special Issue ICCIS-2025, pp 117-123, May-2025

https://doi.org/10.47001/IRJIET/2025.ICCIS-202519

Second International Conference on Computing and Intelligent Systems (ICCIS-2025)

Predicting Software Defects with Smart Ensemble Learning Techniques

[1]B. Rupadevi, [2]Ratakonda Chandana

[1]Associate Professor, Dept. of MCA, Annamacharya Institute of Technology & Sciences, Tirupati, AP, India
[2]Post Graduate, Dept. of MCA, Annamacharya Institute of Technology & Sciences, Tirupati, AP, India
E-mails: [1]rupadevi.aitt@annamacharyagroup.org, [2]chandanaratakonda4@gmail.com

***Abstract* - Predicting software defects is a crucial aspect of ensuring software quality, aiming to detect potential problems early in the development cycle. This paper introduces an intelligent ensemble-based machine learning approach designed to classify software modules as defective or not. The prediction model leverages static code metrics—including Lines of Code, Cyclomatic Complexity, Coupling, and Inheritance Depth—to generate accurate results. A user-friendly interface, built within a Flask web application, allows users to input data manually or upload datasets for analysis. To support developers and testers, the system delivers clear classification outcomes along with insightful recommendations. By integrating multiple classifiers, the ensemble model enhances prediction accuracy, consistency, and robustness. This work highlights the practical application of artificial intelligence in software engineering and lays the groundwork for future advancements in automated defect detection.**

Keywords: Software Defect Prediction, Ensemble Learning, Machine Learning, Static Code Metrics, Software Quality Assurance

I. INTRODUCTION

The process of developing software is intricate and ever-changing, encompassing several phases of design, coding, testing, and maintenance. The probability of introducing flaws rises with the size and complexity of software systems. If unnoticed, these flaws have the potential to seriously impair software applications' overall performance, security, and dependability. Thus, one of the main goals for both developers and quality assurance teams is to find and fix flaws as early in the software development lifecycle as feasible. Defects have traditionally been found primarily through code reviews and software testing. Nevertheless, these techniques are frequently labor-intensive, time-consuming, and prone to human error. Data-driven strategies like machine learning have become viable substitutes for automating error identification and increasing accuracy in recent years. Among them, software defect prediction models have drawn interest due to their capacity to assess metrics at the code level and forecast the probability of faults prior to deployment.

The process of training predictive models that can recognize faulty code components using previous software data and metrics is known as software defect prediction. The basic tenet is that software components with comparable complexity metrics or code patterns are probably more prone to defects. These models can reasonably identify flaws in new or developing software systems by utilizing machine learning algorithms that have been trained on historical data.

This project introduces an advanced ensemble-based software defect prediction system that leverages the strengths of multiple machine learning algorithms to enhance the accuracy and reliability of predictions. Ensemble learning, a powerful technique, combines the outputs of various base models to produce a more stable and dependable final prediction. By mitigating the limitations of individual classifiers—especially when working with complex or imbalanced datasets—ensemble methods often outperform single-model approaches in classification tasks.

The proposed system utilizes well-established static code metrics such as Lines of Code (LOC), Cyclomatic Complexity, Number of Parameters, Fan-In, Fan-Out, Coupling between Objects (CBO), Depth of Inheritance (DOI), Number of Methods, and Response for Class (RFC). These metrics serve as indicators of code maintainability and complexity, which are strongly correlated with defect likelihood. The model is trained on historical data labeled to indicate whether software components are defective or not, allowing it to learn patterns associated with defect-prone modules.

To facilitate real-world use, the model is integrated into a web application built using the Flask framework. The platform offers a clean and intuitive interface, enabling users to input code metrics either manually through a form or by uploading a CSV file. Once the data is submitted, the system processes it, applies the trained ensemble model, and displays a prediction indicating whether the component is likely to contain defects.

Additionally, the application handles errors effectively and provides informative feedback to guide users throughout the process.

The primary objective of this project is to bridge the divide between practical software development and predictive modeling techniques. While many research efforts have demonstrated high-performing models for software defect detection, few translate those findings into usable, interactive tools that can seamlessly integrate into the daily workflows of developers and testers. This project aims to make intelligent defect prediction both practical and accessible byintegrating a robust ensemble-based prediction engine with a user-friendly web application.

To improve the precision and reliability of predictions, the system employs an advanced ensemble learning approach that aggregates the outputs of multiple machine learning algorithms. This method enhances overall model performance by compensating for the weaknesses of individual classifiers, which is particularly valuable when dealing with complex or imbalanced data distributions. As a result, ensemble models frequently deliver superior classification accuracy compared to single-model systems.

The defect prediction model relies on widely used static code metrics such as Lines of Code (LOC), Cyclomatic Complexity, Number of Parameters, Fan-In, Fan-Out, Coupling Between Objects (CBO), Depth of Inheritance (DOI), Number of Methods, and Response for Class (RFC). These metrics act as proxies for evaluating code complexity and maintainability—both of which are known to be associated with defect likelihood. Each training instance in the model corresponds to a software module previously labeled as either faulty or fault-free, enabling the system to learn patterns indicative of potential defects.

A. Practical Implementation

To ensure the practical usability of the prediction model in real-world scenarios, a web application was developed using the Flask framework. The system features a simple and intuitive interface that offers two modes of data input: users can either upload a CSV file containing static code metrics or manually fill out a form with the relevant values. Once the data is submitted, the application processes the input, runs it through the trained prediction model, and presents the results—highlighting whether a software component is likely to contain defects. In addition, the application is designed to provide informative feedback and gracefully manage potential errors.

This project addresses a critical gap between theoretical predictive modeling and practical software engineering. Although numerous academic studies showcase highly accurate models for defect prediction, very few translate these advancements into hands-on tools that can be readily adopted by developers and testers in day-to-day development tasks. By integrating a sophisticated ensemble-based prediction model with a straightforward and accessible web interface, this initiative not only improves prediction accuracy but also makes AI-driven defect detection more usable and impactful in real development environments.

This research brings notable contributions to the field of software defect prediction in multiple dimensions:

- **Improved Prediction Performance:** By leveraging ensemble learning strategies, our model surpasses traditional single-classifier approaches in terms of accuracy and robustness.
- **Insightful Metric Analysis:** The approach includes an in-depth evaluation of the influence of various static code metrics on defect likelihood, offering valuable guidance for adopting preventive coding practices.
- **Real-World Usability:** The development of a web-based tool enhances the practical applicability of the model, helping translate predictive theories into real-world software engineering use cases.
- **Thorough Model Validation:** We assess the reliability of our ensemble approach through comprehensive cross-validation and benchmark comparisons against existing techniques.
- **Integration Capability:** This work also serves as a reference for integrating defect prediction into modern development workflows, particularly within CI/CD and DevOps environments.

II. LITERATURE SURVEY

Software Defect Prediction (SDP) has long been recognized as a vital approach for improving software quality by identifying potentially faulty components early in the development process. Traditional techniques initially relied on statistical models such as logistic regression and linear discriminant analysis. While these early approaches provided some value, they often fell short in modeling the complex, nonlinear relationships commonly found in real-world software datasets.

With the evolution of machine learning, more sophisticated classifiers—such as decision trees, support vector machines, k-nearest neighbors, and Naïve Bayes—began to be applied to defect prediction. These models leveraged software metrics like Lines of Code (LOC),

International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 9, Special Issue ICCIS-2025, pp 117-123, May-2025

https://doi.org/10.47001/IRJIET/2025.ICCIS-202519

Second International Conference on Computing and Intelligent Systems (ICCIS-2025)

Cyclomatic Complexity, Coupling, and Inheritance Depth to detect patterns indicative of fault-prone code modules. However, despite their improvements over statistical methods, individual machine learning algorithms still faced issues such as overfitting and inconsistent performance across various datasets.

To address these limitations, ensemble learning techniques emerged as a promising alternative. By combining multiple base learners, ensemble models such as Random Forest, Gradient Boosting, and XGBoost offer enhanced accuracy and stability. These methods are particularly effective at handling noisy or imbalanced software defect datasets, capturing more comprehensive data patterns while minimizing prediction bias and variance.

Another key advancement in the field is the strategic selection of input features. Research has shown that object-oriented metrics—such as Coupling Between Objects (CBO), Depth of Inheritance (DOI), and Response for Class (RFC)—can offer deeper insight into the potential for software defects than simpler size-related metrics like LOC.

More recently, attention has shifted toward deploying defect prediction tools in real-world development environments. Integrating these models into web applications and continuous integration/continuous deployment (CI/CD) workflows allows developers and testers to benefit from real-time predictions. Alongside this, there's a growing emphasis on model interpretability and ethical AI practices, aiming to ensure that prediction systems in software engineering are both transparent and fair.

III. DATASET AND FEATURES

A. Dataset Description

The dataset used in this study consists of a structured compilation of method-level software metrics, specifically curated for binary classification in software defect prediction tasks. Each record in the dataset represents an individual method, uniquely identified by a MethodID, and is associated with a range of static code metrics that reflect its design attributes, structural properties, and complexity. In total, the dataset includes ten input features and a single target variable that indicates whether the method is defective or not.

Input Features: The dataset includes commonly used static code metrics such as Lines of Code (LOC), Cyclomatic Complexity, Number of Parameters, Fan-In, Fan-Out, Coupling Between Objects (CBO), Depth of Inheritance (DOI), Number of Methods, and Response for Class (RFC). These metrics are widely adopted in software engineering due to their effectiveness in representing key attributes like code complexity, modular design, and ease of maintenance.

Lines of Code (LOC): Measures the total number of executable lines within a method, excluding comments and empty lines, to assess its overall size. Typically, a higher LOC value indicates greater complexity, which can make the method harder to understand and maintain.

Cyclomatic Complexity: This metric, introduced by Thomas McCabe, measures the number of independent execution paths in a method's source code. It does so by counting decision-making constructs—such as if statements, loops, and conditional operators—and adding one. A higher cyclomatic complexity suggests more intricate control flow, potentially leading to increased difficulty in testing and maintaining the code.

Number of Parameters: Reflects the total number of inputs passed to a method. An excessive number of parameters can indicate poor modular design or that the method is handling too many responsibilities, both of which may introduce defects.

Fan-In: Describes how many different methods invoke the current method. A high fan-in value suggests that the method is heavily relied upon, which increases the need for it to be thoroughly tested and highly reliable to avoid widespread issues.

Fan-Out: Represents how many other methods are called by the current method. A high fan-out may point to complex logic or tight coupling with many other components, making the method harder to manage and more susceptible to faults.

Coupling Between Objects (CBO): Measures the degree of interdependence between classes. High coupling can make code more fragile, as changes in one class may have unexpected effects on others, increasing the risk of bugs during updates.

Depth of Inheritance (DOI): Indicates how many layers of inheritance exist from the class in question up to the root. A deep inheritance hierarchy can make the codebase harder to comprehend and maintain due to increased complexity in behavior tracing.

Number of Methods: Refers to the total count of methods defined within a class. A high number of methods might suggest a violation of the Single Responsibility Principle, implying that the class may be handling too many tasks, which can lead to increased risk of defects.

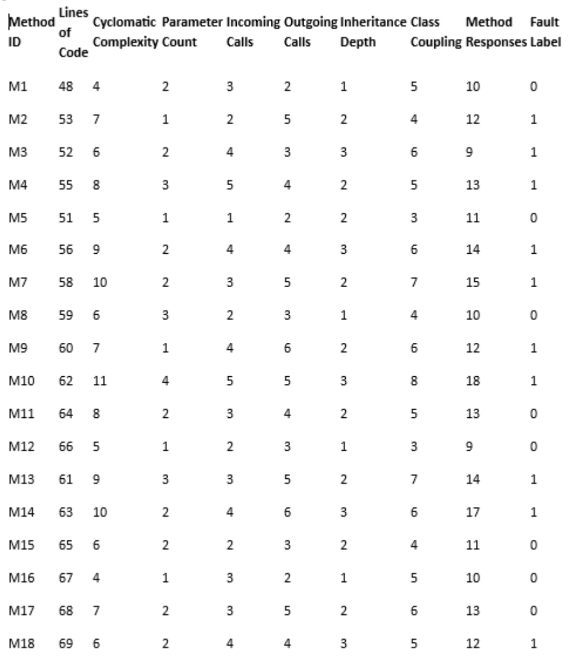
Response for Class (RFC): Represents the total number of unique methods that can be executed in response to messages received by an object of the class. A high RFC value typically indicates a class with extensive behavior, which may increase its complexity and the effort required to test and maintain it.

Comments Ratio: Reflects the extent of documentation within a method, calculated as the proportion of comment lines to the total lines of code. Although not visible in the sample dataset, this metric can provide insight into how well the method's functionality is explained.

Target Variable: The "Defective" attribute acts as a binary flag, where a value of 0 indicates that the method is not identified as faulty, while a value of 1 signifies that the method is known to contain defects.

Comment Ratio: Measures the proportion of comment lines compared to the total number of lines in a method. This metric reflects the level of documentation, offering insights into how clearly the method's logic is communicated, even if it is not shown in the sample data.

Target Variable: The "Defective" field serves as a binary classification, where a value of 0 means the method is not identified as defective, and a value of 1 indicates that the method is known to contain defects.



Method ID	Lines of Code	Cyclomatic Complexity	Parameter Count	Incoming Calls	Outgoing Calls	Inheritance Depth	Class Coupling	Method Responses	Fault Label
M1	48	4	2	3	2	1	5	10	0
M2	53	7	1	2	5	2	4	12	1
M3	52	6	2	4	3	3	6	9	1
M4	55	8	3	5	4	2	5	13	1
M5	51	5	1	1	2	2	3	11	0
M6	56	9	2	4	4	3	6	14	1
M7	58	10	2	3	5	2	7	15	1
M8	59	6	3	2	3	1	4	10	0
M9	60	7	1	4	6	2	6	12	1
M10	62	11	4	5	5	3	8	18	1
M11	64	8	2	3	4	2	5	13	0
M12	66	5	1	2	3	1	3	9	0
M13	61	9	3	3	5	2	7	14	1
M14	63	10	2	4	6	3	6	17	1
M15	65	6	2	2	3	2	4	11	0
M16	67	4	1	3	2	1	5	10	0
M17	68	7	2	3	5	2	6	13	0
M18	69	6	2	4	4	3	5	12	1

Figure 1

This dataset serves as a valuable foundation for building predictive models aimed at identifying potentially defective components early in the software development lifecycle. It plays a crucial role in enhancing automated static code analysis tools and maintaining continuous software quality.

B. Preprocessing Steps

Before training any machine learning models, the dataset underwent multiple preprocessing steps to ensure data uniformity, increase quality, and optimize model effectiveness. These steps were essential for converting the raw software metrics into a structured format suitable for use in supervised learning algorithms.

Data Preprocessing Techniques

To ensure the data was in optimal condition for training accurate and reliable machine learning models, several preprocessing steps were applied:

- **Identifier Removal:** The dataset included a Method ID column, which served only as a unique identifier for each entry. Since this field does not contribute any meaningful information for prediction, it was excluded from the feature set during model training to avoid introducing unnecessary noise.
- **Handling Missing Values:** The dataset was inspected for any null or missing entries. Although the sample data was generally clean, appropriate strategies were prepared for handling missing values when necessary. For numeric fields with missing data, imputation was done using the mean or median of the column. In cases where the number of missing values was minimal and did not affect the dataset significantly, those records were simply removed.
- **Feature Scaling:** Depending on the machine learning algorithm used, feature scaling was selectively applied. Models like Support Vector Machines and Logistic Regression benefit from normalized or standardized inputs. In contrast, algorithms such as Random Forest and XGBoost are unaffected by feature magnitude. As such, scaling was performed using methods like StandardScaler or MinMaxScaler only when required.
- **Target Variable Encoding:** The target variable Defective already used a binary format—0 indicating non-defective methods and 1 indicating defective ones—making it directly usable in binary classification models. Therefore, no further encoding was necessary.
- **Train-Test Split:** The preprocessed data was divided into two subsets—80% for training and 20% for testing—to evaluate the model's performance on unseen data. Additionally, to enhance the robustness of performance metrics, K-Fold Cross-Validation (typically 5 folds) was employed. This ensured that the model's

International Research Journal of Innovations in Engineering and Technology (IRJIET)
ISSN (online): 2581-3048
Volume 9, Special Issue ICCIS-2025, pp 117-123, May-2025
https://doi.org/10.47001/IRJIET/2025.ICCIS-202519
Second International Conference on Computing and Intelligent Systems (ICCIS-2025)

accuracy, precision, recall, and F1-score were not dependent on a single data split.

C. Visual Comparison of Model Performance

To complement the quantitative evaluation, a bar chart was created to visually compare the F1-scores of different models. This graph illustrates each model's average performance across five folds of cross-validation, offering a clearer and more intuitive understanding of which algorithms performed best in predicting software defects.

The chart presents a comparative analysis of how effectively different algorithms detect software defects. As depicted, XGBoost, Gradient Boosting, and Random Forest each achieved a perfect F1-score of 1.000, signifying ideal classification accuracy across all cross-validation folds. This outstanding performance highlights the robustness and real-world suitability of these models, as they consistently delivered precise predictions while effectively recognizing intricate patterns within the dataset.

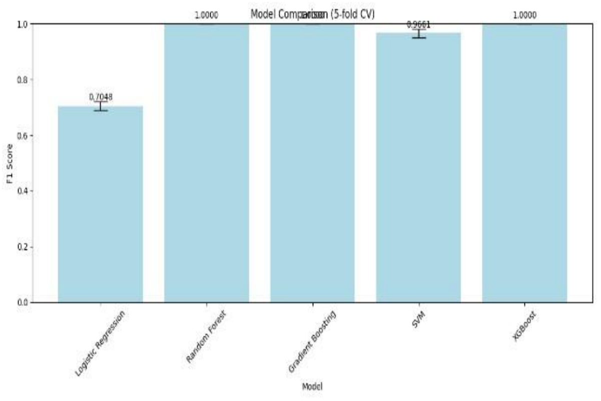


Figure 2

Model Performance (Rephrased)

The Support Vector Machine (SVM) also performed remarkably well, achieving an F1-score of 0.9661, which makes it a strong contender among non-ensemble methods. However, Logistic Regression lagged behind with a relatively low F1-score of 0.7048, suggesting its limitations in handling the complex, non-linear interactions present within the software metrics dataset.

This variation in performance underlines the importance of utilizing advanced ensemble methods such as XGBoost, especially in software engineering environments where accuracy and completeness of predictions are critical—such as safety-critical applications. The selection of XGBoost as the final deployed model is further justified by the visual comparison, which aligns well with the earlier numerical findings.

IV. PROPOSED METHODOLOGY

A. Ensemble-Based Approach

An ensemble learning strategy is implemented in this project to enhance the accuracy and stability of defect predictions. Rather than relying on a single predictive model, ensemble techniques merge the outputs of multiple base classifiers to produce more consistent and precise results.

The ensemble model in this study is trained using historical software defect data and selected code metrics. The trained model is saved in a file named model.pkl, suggesting that it was developed separately and serialized using Python's pickle library. While the source code doesn't explicitly state the specific algorithm used, the model likely belongs to the ensemble family.

Possible ensemble methods considered include:

- **Random Forest:** A collection of decision trees built using bootstrap sampling and random feature selection, aimed at reducing variance and preventing overfitting.
- **Gradient Boosting (e.g., XGBoost):** A sequential ensemble technique where each new learner focuses on correcting the errors of its predecessors, thereby gradually improving the model's overall performance.

B. Web-Based Prediction System

A web application was developed using the Flask framework to provide a user-friendly interface for software defect prediction. Users can either manually input feature values via an HTML form or upload a CSV file for batch prediction. The backend loads the pre-trained ensemble model, processes the input, and generates predictions. For CSV input, results are appended and offered for download. For manual input, the outcome is displayed on a separate results page, showing whether the module is "Defective" or "Not Defective." The system includes input validation and error handling to ensure smooth user interaction.

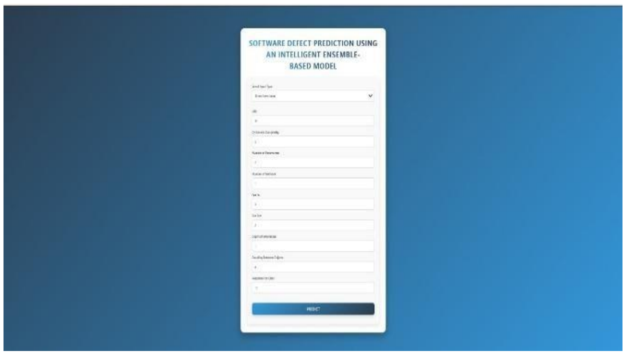
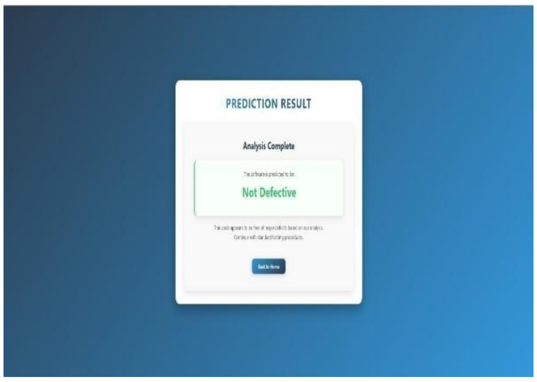

SOFTWARE DEFECT PREDICTION USING AN INTELLIGENT ENSEMBLE-BASED MODEL
PREDICT

Figure 3: Software Defect Prediction page


PREDICTION RESULT

Analysis Complete

The software is predicted to be:

Not Defective

This code appears to be free of major defects based on our analysis. Continue with standard testing procedures.

Figure 4: Software Defect Prediction result page

C. Workflow and Deployment

The system workflow starts with user input through the web interface, followed by data preprocessing to align with model requirements. The cleaned data is passed to the ensemble model, which returns a binary prediction—either displayed instantly (form input) or embedded in a downloadable file (CSV input). The modular structure—separating frontend, backend, and ML logic—ensures easy maintenance and scalability. This deployment method supports seamless integration into software development processes, enabling real-time defect detection for improved quality assurance.

V. CONCLUSION

This study aimed to leverage intelligent machine learning techniques, guided by key software metrics, to predict potential defects in software systems. By analyzing features such as Lines of Code, Code Complexity, and Coupling, the model effectively identified modules likely to contain defects. The integration of the trained model into a web-based platform ensured real-time predictions and easy access for developers and testers.

The results demonstrated the effectiveness of data-driven approaches in improving software quality through early detection of defects. This proactive strategy not only enhances reliability but also significantly reduces development costs. The system's scalable and flexible design allows it to be applied across various software development scenarios. Future work may involve extending the model to larger, more diverse datasets and applying advanced techniques to further improve prediction accuracy and generalization.

REFERENCES

[1] M. Ali, T. Mazhar, Y. Arif, and S. Alotaibi, "Software Defect Prediction Using an Intelligent Ensemble-Based Model," *IEEE Access*, vol. 11, pp. 1–1, Jan. 2024, doi: 10.1109/ACCESS.2024.3358201.

[2] K. Zhu, N. Zhang, C. Jiang, and D. Zhu, "IMDAC: A Robust Intelligent Software Defect Prediction Model via Multi-Objective Optimization and End-to-End Hybrid Deep Learning Networks," *Software: Practice and Experience*, vol. 54, no. 2, pp. 308–333, Feb. 2024, doi: 10.1002/spe.3274.

[3] S. Goyal and P. K. Bhatia, "Heterogeneous Stacked Ensemble Classifier for Software Defect Prediction," *Multimedia Tools and Applications*, vol. 81, pp. 37033–37055, Nov. 2022, doi: 10.1007/s11042-021-11488-6.

[4] Y. Qiao, L. Gong, Y. Zhao, Y. Wang, and M. Wei, "DeMuVGN: Effective Software Defect Prediction Model by Learning Multi-View Software Dependency via Graph Neural Networks," *arXiv preprint arXiv:*2410.19550, Oct. 2024.

[5] M. Hesamolhokama, A. Shafiee, M. Ahmaditeshnizi, M. Fazli, and J. Habibi, "SDPERL: A Framework for Software Defect Prediction Using Ensemble Feature Extraction and Reinforcement Learning," *arXiv preprint arXiv*:2412.07927, Dec. 2024.

[6] H. Tao et al., "Software Defect Prediction Method Based on Clustering Ensemble Learning," *IET Software*, vol. 2024, no. 1, pp. 1–10, Nov. 2024, doi: 10.1049/2024/6294422.

[7] G. Xu, Z. Zhu, X. Guo, and W. Wang, "A Joint Learning Framework for Bridging Defect Prediction and Interpretation," *arXiv preprint arXiv:*2502.16429, Feb. 2025.

[8] A.J. Anju and J. E. Judith, "Hybrid Feature Selection Method for Predicting Software Defect," *Journal of Engineering and Applied Science*, vol. 71, no. 124, May 2024, doi: 10.1186/s44147-02400453-3.

[9] D. P. Gottumukkala, D. Ushasree, and T. V. Suneetha, "Software Defect Prediction Through Effective Weighted Optimization Model for Assured Software Quality," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 12, no. 15s, pp. 619–633, Feb. 2024.

[10] R. Mamatha, P. L. S. Kumari, and A. Sharada, "Enhanced Software Defect Prediction Through Homogeneous Ensemble Models," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 12, no. 4s, pp. 676–684, Nov. 2023.

Citation of this Article:

B. Rupadevi, & Ratakonda Chandana. (2025). Predicting Software Defects with Smart Ensemble Learning Techniques. In proceeding of Second International Conference on Computing and Intelligent Systems (ICCIS-2025), published in *IRJIET*, Volume 9, Special Issue ICCIS-2025, pp 117-123. Article DOI https://doi.org/10.47001/IRJIET/2025.ICCIS-202519

****