ABSTRACT

This project presents a machine learning-based approach to predict and classify bug-prone software modules using software metrics. The datasets were collected from prominent open- source software systems, including Eclipse JDT, PDE, Lucene, Equinox, and Mylyn, and consist of metrics such as Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Fan-Out, Lines of Code (LOC), and Weighted Methods per Class (WMC). These metrics reflect various software complexities, including size, cohesion, and coupling, which are key indicators of defect-proneness in software modules.

The data preprocessing phase involved cleaning, handling missing values, scaling features, and addressing class imbalances to ensure high-quality input for modeling. Machine learningmodels such as Random Forest Classifier, K-Nearest Neighbors (KNN), and K-Means Clustering were implemented for supervised and unsupervised analysis. Hyperparameter tuning using GridSearchCV optimized model performance.

The models were evaluated using metrics such as Accuracy, ROC-AUC, and F1-Score, providing a comprehensive assessment of their effectiveness. The results revealed that Random Forest Classifier achieved the highest performance, accurately identifying bug-prone modules and showcasing its suitability for classification tasks. The KNN model also performed well, while K-Means Clustering provided insights into hidden patterns within the data.

By automating bug detection, this project streamlines software testing processes, reduces manual debugging efforts, and enhances software quality assurance. The insights derived from the analysis help developers prioritize testing and maintenance efforts, contributing to the development of reliable and robust software systems. Future scope includes the integration of advanced deep learning techniques, real-time monitoring systems, and expanding the analysisto more diverse and large-scale datasets.

INDEX

CHAPTER 1 : INTRODUCTION

1. OVERVIEW

The project centers on leveraging machine learning techniques for software bug detection and classification, providing an automated solution to an otherwise manual and labor-intensive task. It integrates datasets from several well-known software systems such as Eclipse JDT, Eclipse PDE, Equinox, Lucene, and Mylyn. These datasets include detailed software metrics along with bug occurrence data, offering a comprehensive view of various software modules. The metrics considered in this project encompass several critical software attributes, such as coupling, fan-in, fan-out, number of lines of code (LOC), inheritance depth, and other complexity measures that offer insights into the internal structure and behavior of the software. These attributes are crucial for understanding how the architecture of a system may influence the presence of bugs.

The project's data preprocessing phase involves various stages to ensure the dataset is clean, complete, and ready for model training. This includes handling missing values using imputation or removal techniques, standardizing the data to ensure all features are on a comparable scale, and normalizing values where necessary. This preprocessing step is essential for eliminating noise, ensuring data consistency, and preparing the dataset for the next stages of analysis. The data is then split into training, validation, and testing sets, ensuring that the models are trained and evaluated in a way that mimics real-world scenarios and avoids overfitting.

In the core of this project, machine learning models, including Random Forest Classifier, K-Nearest Neighbors (KNN), and K-Means Clustering, are employed to analyze the relationship between the collected software metrics and bug occurrences. These algorithms are chosen based on their ability to handle diverse types of data and their effectiveness in both classification and clustering tasks. Random Forest Classifier, an ensemble learning method, utilizes multiple decision trees to improve prediction accuracy. KNN, a simple yet powerful model, classifies software modules by evaluating the similarity between a given module and its neighbors based on feature values. K-Means Clustering, on the other hand, groups similar software modules together to identify patterns in bug-prone modules based on their characteristics.

Hyperparameter tuning plays a crucial role in optimizing the performance of these machine learning algorithms. The GridSearchCV technique is applied to exhaustively search through the hyperparameter space and identify the optimal values for the models. This ensures the models perform at their best and generalize well to unseen data, preventing underfitting or overfitting.

To assess the performance of the models, several evaluastion metrics are used, including accuracy, ROC-AUC (Receiver Operating Characteristic - Area Under Curve), and F1-score. These metrics are chosen because they offer a comprehensive evaluation of the models' classification performance, accounting for both true positives and false positives, as well as the balance between precision and recall. These metrics provide insights into how well the models distinguish between bug-prone and non-bug-prone software modules, ensuring their practical applicability in real-world scenarios.

Additionally, data visualization techniques are employed throughout the project to aid in understanding class distribution, feature relationships, and model performance. Visualizations like histograms, bar plots, and scatter plots help to provide a clearer picture of how various features relate to bug occurrences, the distribution of bugs across the modules, and the overall performance of the machine learning models. By visually inspecting the results, developers and stakeholders can gain insights into the effectiveness of the proposed models and further refine the process.

In conclusion, this project aims to provide an automated and effective solution for bug detection and classification using machine learning. By integrating various software metrics and employing a combination of supervised and unsupervised learning techniques, the project strives to identify bug-prone modules early in the development process, allowing developers to prioritize testing efforts and improve software quality.

2. PURPOSE

The primary purpose of this project is to enhance the process of software bug detection and classification by leveraging machine learning techniques to analyze software metrics. By utilizing advanced algorithms, the project aims to automate and improve the accuracy of identifying bug-prone software components, ultimately contributing to more reliable and efficient software development practices. The specific objectives of the project include:

1. **Identifying bug-prone software components**: The project seeks to predict the likelihood of bugs in different software modules based on complexity measures and other key metrics, such as Coupling Between Objects (CBO), Lines of Code (LOC), Fan-In, Fan-Out, and Depth of Inheritance Tree (DIT). By analyzing these features, the system identifies which software modules are more susceptible to bugs, providing developers with insights into potential problem areas.

2. **Improving software reliability**: One of the project's core objectives is to support early detection and correction of software issues by automating the identification of bug-prone modules. By pinpointing these modules early in the development process, developers can address potential bugs before they lead to system failures, ultimately improving the overall software quality and reliability. This proactive approach reduces the chances of bugs slipping into production, ensuring better user experiences and fewer post-release issues.

3. **Supporting efficient software testing**: The project helps developers and testers focus their efforts on the software components most likely to contain bugs. By predicting which modules are bug-prone, the machine learning models allow for more targeted and efficient testing. Testers can prioritize high-risk modules, reducing unnecessary tests on stable components and improving the overall efficiency of the software testing process.

4. **Comparing machine learning models**: The project evaluates multiple machine learning algorithms, including Random Forest, K-Nearest Neighbors (KNN), and K-Means, to determine the most effective approach for bug classification. By comparing the performance of these algorithms, the project identifies the best model that provides accurate and reliable predictions of bug occurrences across different software modules. The evaluation process also includes hyperparameter optimization to enhance the models' performance further.

5. **Understanding feature importance**: By analyzing how different software metrics influence bug occurrences, the project offers valuable insights into which metrics are most significant in predicting bugs. This understanding of feature importance allows developers to focus on the most critical aspects of their code, improving decision-making and software design. Metrics like coupling, code size, and inheritance depth play a central role in identifying potential defects, and their relationships with bug occurrences provide a deeper understanding of software quality.

CHAPTER 2 : LITERATURE SURVEY

1. EXISTING PROBLEM

In large-scale software systems, identifying and predicting bugs across various modules presents significant challenges. Software modules often exhibit varying levels of complexity, coupling, and code size, which influence their likelihood of containing defects.[1] Traditional methods of bug detection, such as manual code reviews and testing, are time-consuming, costly, and prone to human error. Additionally, the vast amount of code and data generated in modern software development makes it difficult for developers to prioritize components that require immediate attention.[2] Existing approaches often fail to accurately leverage software metrics to identify patterns that contribute to bugs, leading to inefficient allocation of testing resources and delayed issue resolution. As a result, software quality and reliability are compromised, impacting end-user satisfaction and system performance.[3]

The limitations of traditional testing methods are particularly evident in low-level code, where conventional static analysis and software testing may not effectively remove security vulnerabilities.[4]

Furthermore, the increasing complexity of modern software has exposed the limitations of traditional bug-tracking methods, highlighting the need for more advanced approaches to ensure product reliability.[5]

To address these challenges, there is a growing interest in leveraging machine learning techniques for automated bug detection. However, even with these advancements, the scalability of debugging in large-scale systems remains a concern, as traditional methods may not be ideal for removing security vulnerabilities from low-level code.[6]

In summary, the existing problem lies in the inefficiencies and limitations of traditional bug detection methods in large-scale software systems, necessitating the exploration of more effective and scalable solutions to enhance software quality and reliability.

2. PROPOSED SOLUTION

To address these challenges, this project proposes a machine learning-based approach to automate the process of software bug detection and classification, aiming to enhance the efficiency and accuracy of identifying bug-prone software modules. The solution involves several key steps to ensure that software metrics are effectively leveraged to predict bugs and guide developers in allocating resources efficiently:

Data Collection and Integration: To create a comprehensive and diverse dataset, software metrics and bug occurrence data are gathered from a variety of software systems, including but not limited to Eclipse JDT, Eclipse PDE, Equinox, Lucene, and Mylyn. This diverse set of data helps capture a wide range of software characteristics and behaviors, which are crucial for accurate bug prediction. By integrating data from multiple sources, the model can be generalized across various development environments and software types, ensuring that the approach is not limited to a specific domain.

Data Preprocessing: The collected data undergoes a series of preprocessing steps to ensure that it is clean, consistent, and ready for model training. This includes handling missing values through imputation or removal, normalizing the data to ensure that features with different units are comparable, and applying feature scaling (e.g., using StandardScaler) to avoid biases introduced by features with larger numerical ranges. Irrelevant or redundant features, which might not contribute significantly to the bug prediction process, are also eliminated to improve the model's efficiency and focus on the most meaningful metrics.

Feature Selection and Engineering: In this step, careful consideration is given to the software metrics that will serve as the primary predictors for bug occurrence. Key metrics like coupling between objects (CBO), depth of inheritance tree (DIT), fan-in/fan-out, and lines of code (LOC) are used because they are known to correlate with the likelihood of bugs in software modules. Additionally, feature engineering techniques may be applied to generate new features or combine existing features in a way that better represents the complexity of the system. This can improve the model's ability to detect patterns and predict bug occurrences more effectively.

Machine Learning Models: The project explores the use of multiple machine learning algorithms, each offering a unique approach to classification and clustering. The Random Forest Classifier, for example, is an ensemble method that uses decision trees to improve prediction accuracy by aggregating results from multiple trees. K-Nearest Neighbors (KNN), on the other hand, is a simple but effective distance-based model that classifies software modules based on their similarity to other modules. Additionally, K-Means Clustering is used to group similar software modules, allowing the identification of clusters that exhibit high bug-prone characteristics. Hyperparameter tuning is conducted for each model to optimize its performance, ensuring that the chosen parameters yield the best possible results.

Performance Evaluation: The effectiveness of the machine learning models is evaluated using several performance metrics, including accuracy, ROC-AUC (Receiver Operating Characteristic - Area Under Curve), and F1-score. These metrics provide a comprehensive view of model performance, assessing not only how accurately the models classify bug-prone software modules but also how well they distinguish between different types of bugs, especially in cases of class imbalance. The use of multiple evaluation metrics ensures that the model is not overfitting to one particular aspect of performance and can generalize well to unseen data.

Insights and Recommendations: Once the models have been trained and evaluated, the project also aims to provide valuable insights into which software metrics have the most significant impact on bug prediction. By analyzing feature importance, developers can gain a better understanding of the characteristics of software modules that are most prone to bugs. This allows developers to prioritize their efforts, focusing on fixing critical bugs and improving code quality in areas that are more likely to result in defects. Furthermore, the insights gathered can inform future software design and development strategies, ensuring that potential risks are mitigated early in the development lifecycle.

CHAPTER 3 : PROPOSED METHOD

1. THEORETICAL ANALYSIS

Theoretical analysis of this project is grounded in the principles of machine learning and software engineering metrics to address the problem of software bug detection and classification. Software systems are complex, and their quality can be quantitatively evaluated using various software metrics. These metrics capture different dimensions of a software module, such as its size, complexity, and interdependencies, which often influence the presence of bugs. By leveraging these metrics, the project aims to predict bug-prone modules, enhancing the software development and maintenance process.

1. Understanding Software Metrics:

Software metrics are key predictors in this project. They provide measurable attributes of software components, enabling analysis of how complexity and design influence bug occurrences. Commonly used metrics include:

- **Coupling Between Objects (CBO):** Measures the level of interdependence between software classes. High coupling can lead to higher bug probability due to complex relationships. When classes are tightly coupled, changes in one class often require modifications in others, making the system more prone to defects.

- **Depth of Inheritance Tree (DIT):** Indicates the number of classes in a hierarchy. Deeper inheritance trees can make the system harder to manage and test. Complex inheritance structures can create confusion and introduce errors, especially in larger systems with deep hierarchies.

- **Fan-In and Fan-Out:** Fan-In measures how many modules reference a specific module, while Fan-Out measures how many modules a specific module references. Higher fan-in/out values suggest greater complexity and likelihood of defects. A module with high Fan-Out might have a larger number of dependencies, making it more vulnerable to errors when changes occur.

- **Lines of Code (LOC):** A widely used measure representing the size of the codebase. Larger code modules tend to have a higher chance of containing bugs. While LOC is a straightforward metric, it should be interpreted carefully, as it doesn't directly reflect the quality or functionality of the code.
- **Lack of Cohesion of Methods (LCOM):** Measures how methods within a class interact with class variables. Poor cohesion can lead to poorly designed modules, increasing the risk of bugs. Low cohesion indicates that the class is trying to perform multiple, unrelated tasks, which complicates maintenance and testing.

2. Machine Learning for Bug Detection:

Traditional methods of bug detection rely on manual code reviews and testing, which are time-consuming and often inefficient. Theoretical advancements in machine learning provide automated approaches to identify patterns and correlations in software metrics that contribute to bug prediction. Machine learning techniques can be categorized as:

Supervised Learning: In this approach, labeled data is used to train models to predict the presence or number of bugs. The model learns from the input data and labels to make predictions on new, unseen data. Examples include:

- **Random Forest Classifier:** An ensemble method that uses multiple decision trees to make robust predictions. By aggregating the results of various trees, this method enhances accuracy and reduces the risk of overfitting.
- **K-Nearest Neighbors (KNN):** A distance-based model that predicts a module's bug class based on its similarity to other modules. It considers the 'k' closest neighbors and assigns the most common class among them to the new instance.

Unsupervised Learning: When labeled data is unavailable, unsupervised methods like **K-Means Clustering** group similar software modules based on their software metrics. These algorithms do not require labels but identify

patterns and groupings in the data, making them useful for exploratory analysis and anomaly detection.

3. Mathematical Foundation:

The theoretical basis of the machine learning models used in this project includes the following concepts:

- **Random Forest Classifier:** This method uses a set of decision trees, each trained on random subsets of data. The model splits the data based on conditions like "Is CBO > threshold?" to minimize entropy or Gini impurity, which are measures of disorder or uncertainty. The final prediction is obtained through majority voting among the trees, ensuring robustness by leveraging multiple decision paths.

- **K-Nearest Neighbors (KNN):** The KNN algorithm works by calculating the distance (e.g., Euclidean distance) between the input instance and all other data points. It selects the 'k' most similar software modules based on these distances and predicts the bug class based on the majority vote of these neighbors. KNN is simple yet effective for classification tasks, especially when the decision boundary is complex.

- **K-Means Clustering:** This unsupervised method minimizes the sum of squared distances between data points and the centroids of their respective clusters. Each cluster represents a group of software modules that share similar characteristics. By minimizing this distance, K-Means optimizes the clustering process, helping group software modules that are more likely to exhibit similar bug behaviors.

4. Evaluation Metrics:

The theoretical performance of the models is evaluated using well-established metrics to ensure reliable predictions and model effectiveness:

- **Accuracy:** This metric measures how well the model correctly classifies bug occurrences. It is the ratio of correctly predicted instances to the total instances and serves as a general indicator of the model's overall performance.

- **ROC (Receiver Operating Characteristic):** The ROC curve evaluates the model's ability to distinguish between classes (e.g., bug-prone vs. non-bug-prone). It plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various thresholds. The area under the curve (AUC) gives a numerical summary of the model's ability to discriminate between the positive and negative classes.

- **F1-Score:** A harmonic mean of precision and recall, the F1-score is particularly useful when dealing with imbalanced classes. It balances the trade-off between false positives and false negatives, ensuring that the model does not favor one class over another, especially when one class (e.g., bug-prone modules) is much less frequent.

5. **Feature Importance:**

The theoretical analysis also involves understanding feature importance, i.e., which software metrics contribute most to predicting bugs. Some metrics, like **Coupling Between Objects (CBO)** or **Lines of Code (LOC),** may have higher importance than others. Models like Random Forest provide insights into the significance of each feature by evaluating how much each feature reduces impurity in the decision trees. This helps developers prioritize which software attributes are most critical in predicting bug occurrences, leading to more targeted improvements in the software's design and structure.

6. Challenges Addressed:

The theoretical analysis highlights how machine learning models can address common challenges in software bug detection:

- **High-dimensional Data:** Software systems often have many features that contribute to the complexity of the system. Machine learning models can handle large datasets with multiple features, allowing them to identify patterns and relationships that might be overlooked in traditional analysis.

- **Imbalanced Data:** Bug occurrence data is often imbalanced, with most modules having no bugs and a few modules having many bugs. Techniques like resampling (over-sampling the minority class or under-sampling the majority class) and using the F1-score as an evaluation metric help address this imbalance and ensure that the model does not become biased towards the majority class.

- **Complex Relationships:** Machine learning models can capture non-linear relationships between software metrics and bug occurrences. Unlike traditional statistical models that assume linearity or simple relationships, machine learning algorithms can model complex interactions, which is crucial in software systems where the factors contributing to bugs are often intertwined and not easily separable.

2. IMPLEMENTATION

2.1. Methodology

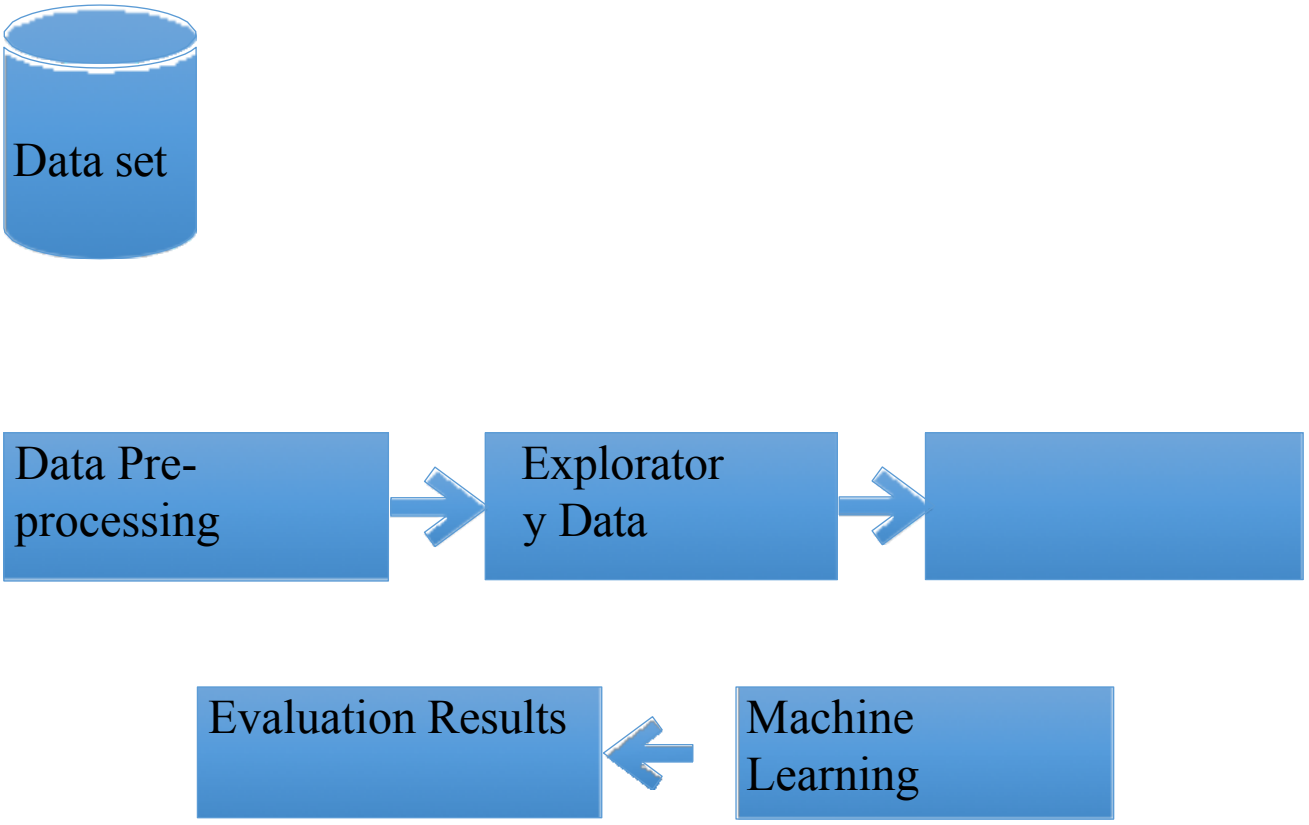


Figure 3.1: Methodology

2.2. Data Collection

For this project, data is collected from five widely used open-source software systems: Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, Lucene, and Mylyn. These datasets contain detailed information about the software modules, including 24 attributes that capture various aspects of software complexity and design. Key attributes include Coupling Between Objects (CBO), which measures interdependencies between classes, Depth of Inheritance Tree (DIT), which indicates the level of inheritance, Fan-In and Fan-Out, which measure the number of referencing and referenced modules, and Lack of Cohesion of Methods (LCOM), which assesses the cohesion of methods within a class. Additional features include basic size metrics like Lines of Code (LOC), Number of Methods, and Number of Attributes, all of which provide insight into the functional complexity of each module.

```
Data Shapes: (997, 24) (1497, 24) (324, 24) (691, 24) (1862, 24)
Full dataframe shape: (5371, 24)
```

Predictors: classname, cbo, dit, fanIn, fanOut, lcom, noc, numberOfAttributes, numberOfAttributesInherited, numberOfLinesOfCode, numberOfMethods, numberOfMethodsInherited, numberOfPrivateAttributes, numberOfPrivateMethods, numberOfPublicAttributes, numberOfPublicMethods, rfc, wmc, bugs, nonTrivialBugs, majorBugs, criticalBugs, highPriorityBugs.

df.describe()

```
df.describe()
```

	cbo	dit	fanIn	fanOut	lcom	noc	numberOfAttributes	numberOfAttributesInherited	numberOfLinesOfCode	numberOfMethods
count	5371.000000	5371.000000	5371.000000	5371.000000	5371.000000	5371.000000	5371.000000	5371.000000	5371.000000	5371.000000
mean	9.650717	1.948985	4.077825	5.808229	131.440328	0.548687	5.648110	20.607522	119.126233	9.465649
std	15.430910	1.339559	12.763334	7.413065	1419.297581	2.320663	32.421878	75.447749	305.121926	13.519683
min	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	3.000000	1.000000	1.000000	1.000000	3.000000	0.000000	1.000000	0.000000	18.000000	3.000000
50%	6.000000	1.000000	1.000000	3.000000	15.000000	0.000000	2.000000	0.000000	47.000000	6.000000
75%	11.000000	2.000000	3.000000	8.000000	55.000000	0.000000	5.000000	2.000000	115.000000	11.000000
max	362.000000	9.000000	355.000000	93.000000	81003.000000	49.000000	2169.000000	563.000000	7509.000000	403.000000

2.3. Data Preprocessing

Data preprocessing involves several crucial steps to prepare the dataset for machine learning, ensuring it is clean, relevant, and ready for model training.

Handling Missing Values: Missing or null values are identified and managed by removing or imputing them, ensuring the dataset remains complete and consistent. Techniques like mean, median, or mode imputation are often used for numerical features, while categorical features may be imputed with the most frequent value.

Feature Selection: Irrelevant or redundant features, like the classname, are removed to reduce noise and focus on meaningful predictors. Feature selection techniques, such as correlation analysis, recursive feature elimination (RFE), or feature importance from models like Random Forest, can help identify the most impactful features for the model.

Scaling and Normalization: Features are scaled using methods like StandardScaler (for normalization) to ensure they are on the same scale, preventing bias in models like KNN, which rely on distance metrics. This step ensures that no feature dominates the model due to its scale and that gradient-based models converge faster.

Shuffling and Splitting: The dataset is shuffled and split into training, validation, and testing sets to ensure fair model evaluation and avoid overfitting. Shuffling ensures that the data is randomly distributed, while splitting it into distinct sets helps in training the model, tuning hyperparameters, and evaluating performance without bias.

These steps ensure the data is clean, relevant, and ready for machine learning model training, improving the performance and reliability of the predictive models.

hyper-parameter tuning and solving the imbalance problem

Classifying data where the classes are:

no bugs, 1 bug, or > 2 bugs

2.4. Exploratory Data Analysis(EDA)

In this project, Exploratory Data Analysis (EDA) is used to understand the dataset and prepare it for machine learning models.

Data Distribution: Summary statistics and visualizations (e.g., histograms, box plots) reveal the distribution of features like LOC, CBO, and Fan-In/Fan-Out, highlighting skewness, outliers, or unusual patterns. These insights help identify

```
Classes: [0, 1, 2]
Counts: [4518, 598, 255]
```

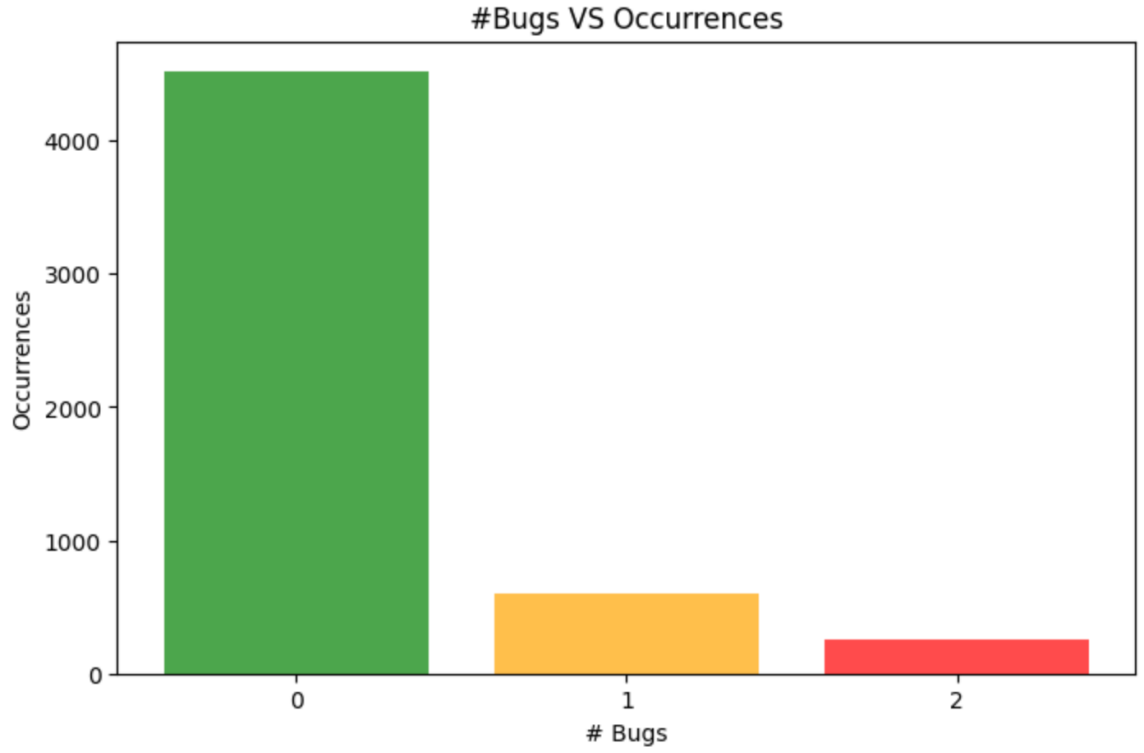


Figure 3.2:Bugs vs Occurrences

which features may require transformation or removal for better model performance.

Bug Distribution: Bar charts display bug counts (0, 1, 2+), helping assess class imbalance and determine if resampling is needed. Identifying imbalances early on ensures that the model isn't biased toward the more frequent class and that it can generalize well across all classes.

Correlation Analysis: Correlation matrices and scatter plots identify relationships between metrics (e.g., CBO, LOC) and bug occurrences, guiding feature selection. Strong correlations help identify relevant features for training, while weak correlations might suggest redundant or less valuable features to exclude.

Missing Data and Outliers: Heatmaps detect missing values and outliers, which are addressed to enhance data quality for model training. Handling missing data through imputation or removal, and dealing with outliers through techniques like clipping or transformation, ensures that the model isn't distorted by inaccurate or incomplete data.

A Seaborn pairplot to visualize the relationships between selected features (rfc, cbo, fanOut, wmc, numberOfLinesOfCode) with respect to the target variable Bugs. It also colors the points based on the Bugs values, helping to identify patterns related to bug occurrences in different classes.

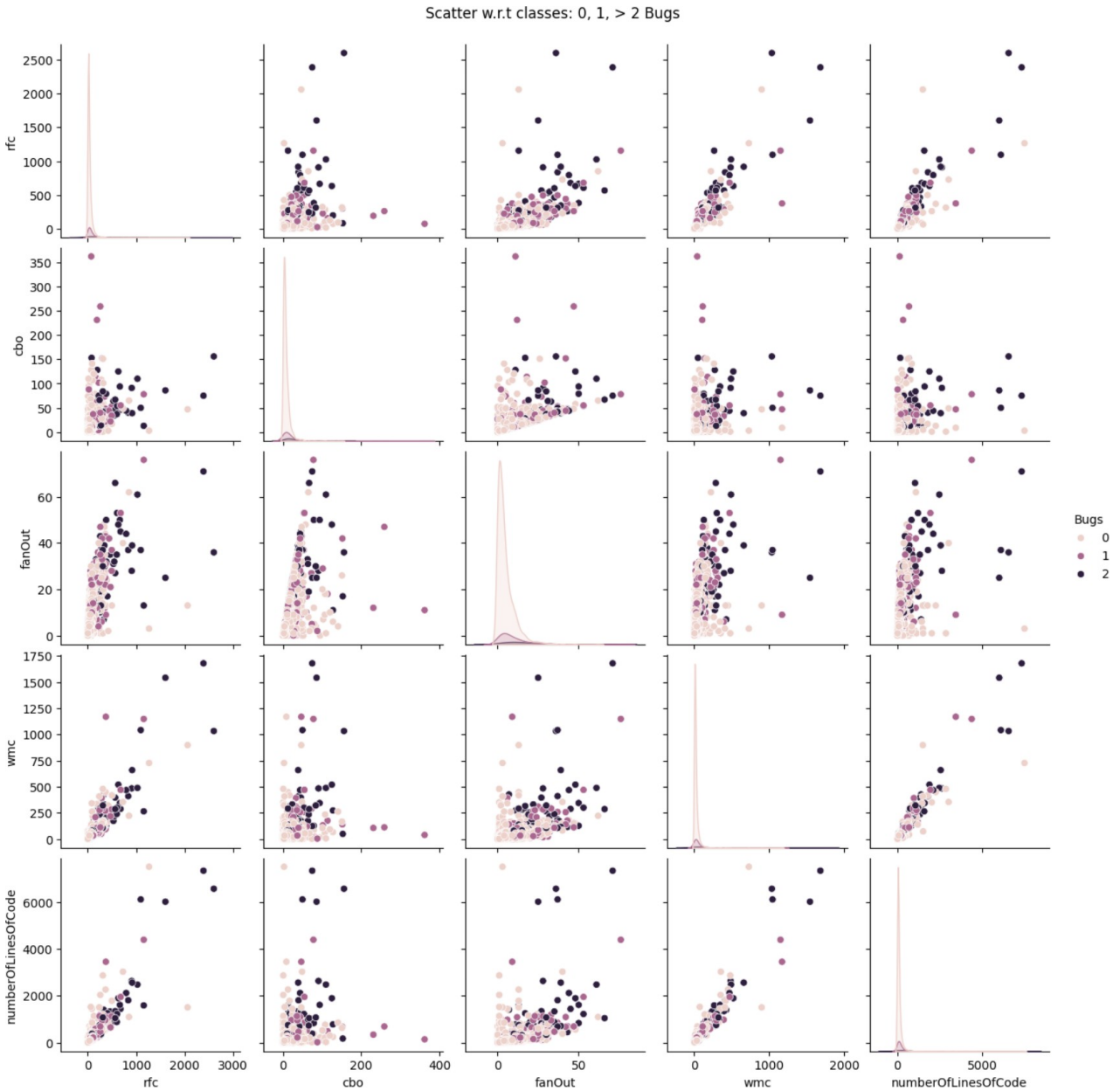


Figure 3.3: Scatter w.r.t 0,1,>2 bugs

Kernel Density Estimations (KDEs) for the rfc feature, separated by different bug classes (0, 1, and >2 Bugs). It visualizes the probability density of rfc values for each bug class with different colors and labels.

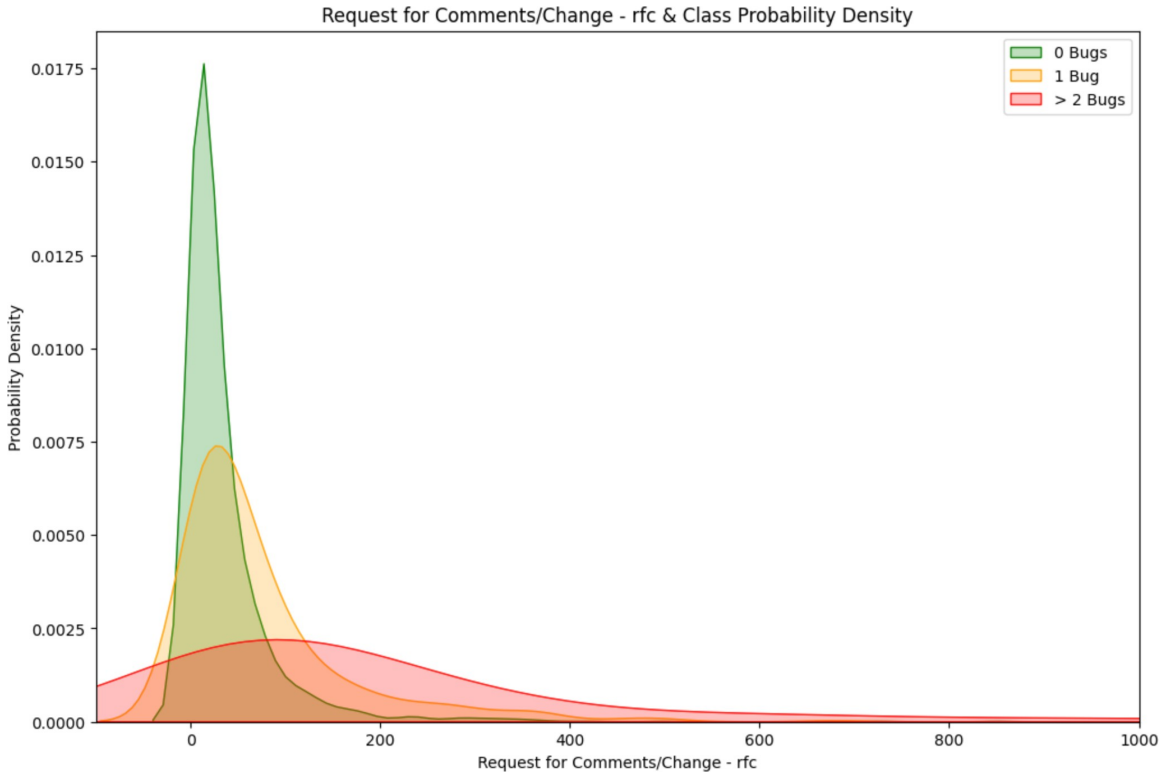


Figure 3.4: rfc & class probability density

Kernel Density Estimations (KDEs) for the cbo feature, separated by different bug classes (0, 1, and >2 Bugs). It shows the probability density of cbo values for each bug class using different colors and labels.

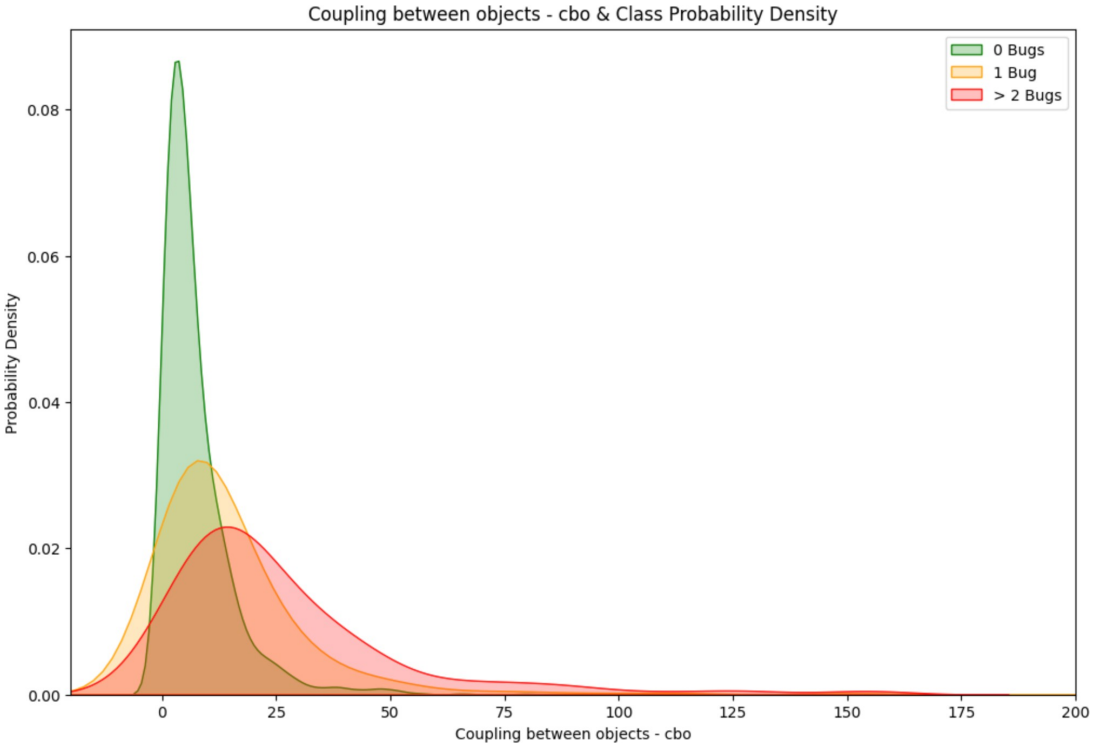


Figure 3.5: cbo & class probability density

Kernel Density Estimations (KDEs) for the fanOut feature, separated by different bug classes (0, 1, and >2 Bugs). It displays the probability density of fanOut values for each class with distinct colors and labels.

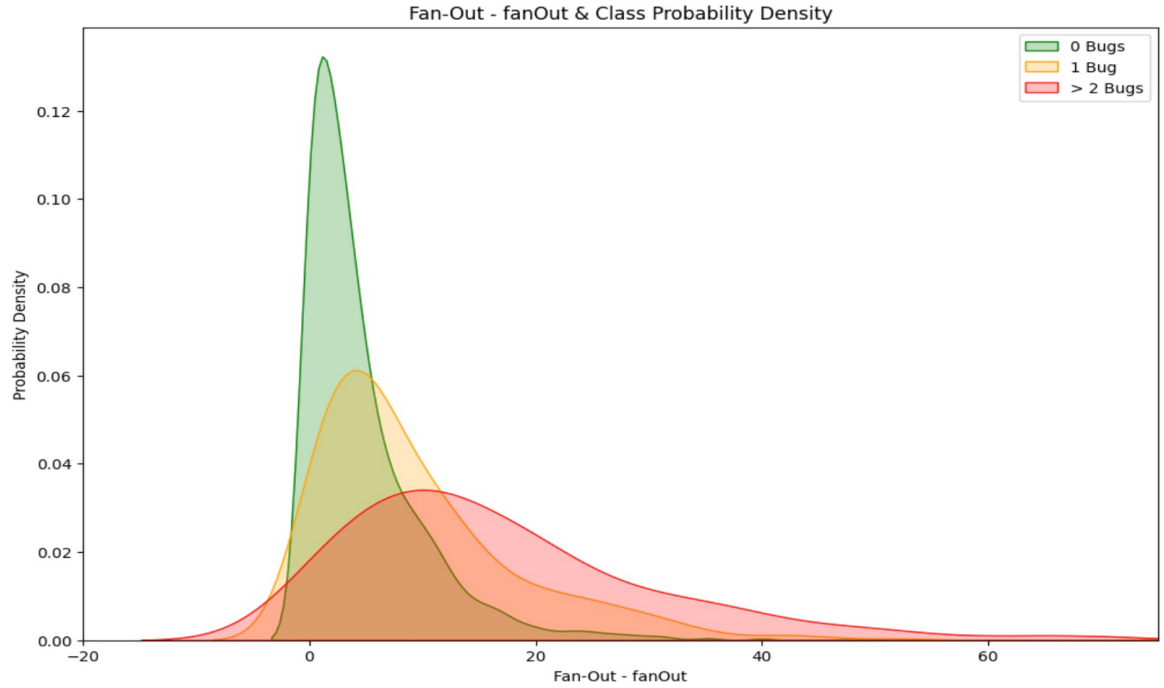


Figure 3.6: FanOut & Class probability density

Kernel Density Estimations (KDEs) for the wmc feature, separated by different bug classes (0, 1, and >2 Bugs). It visualizes the probability density of wmc values for each bug class using distinct colors and labels.

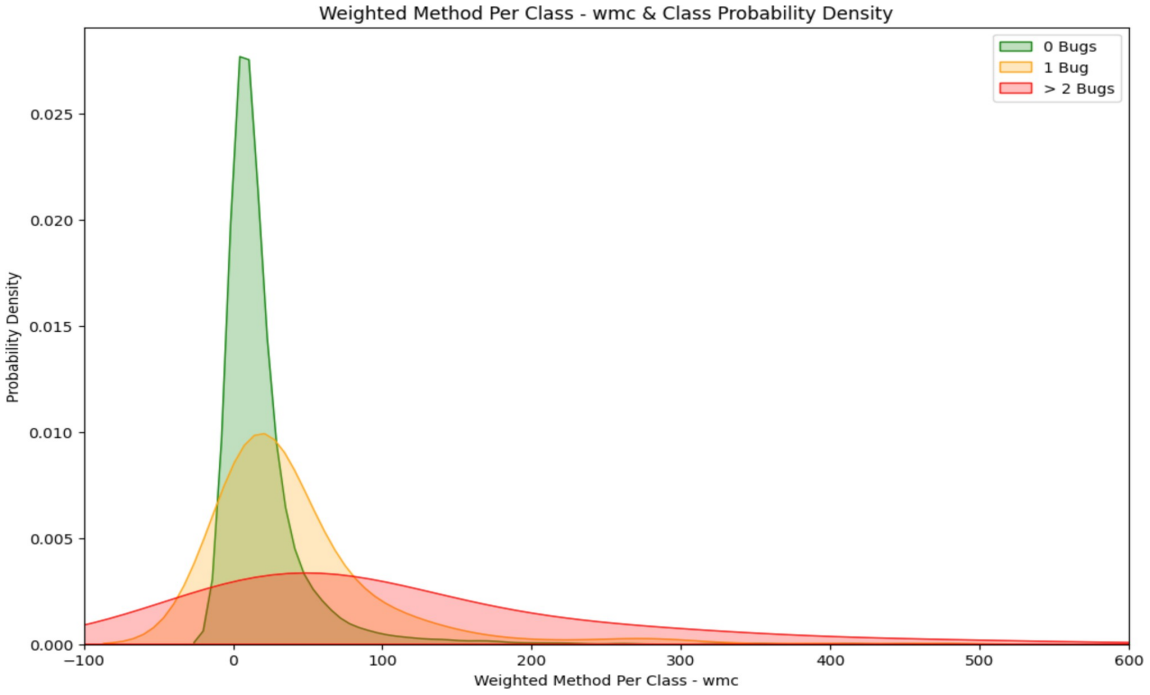


Figure 3.7: cmc & class probability density

Kernel Density Estimations (KDEs) for the numberOfLinesOfCode feature, separated by different bug classes (0, 1, and >2 Bugs). It shows the probability density of numberOfLinesOfCode values for each class with distinct colors and labels.

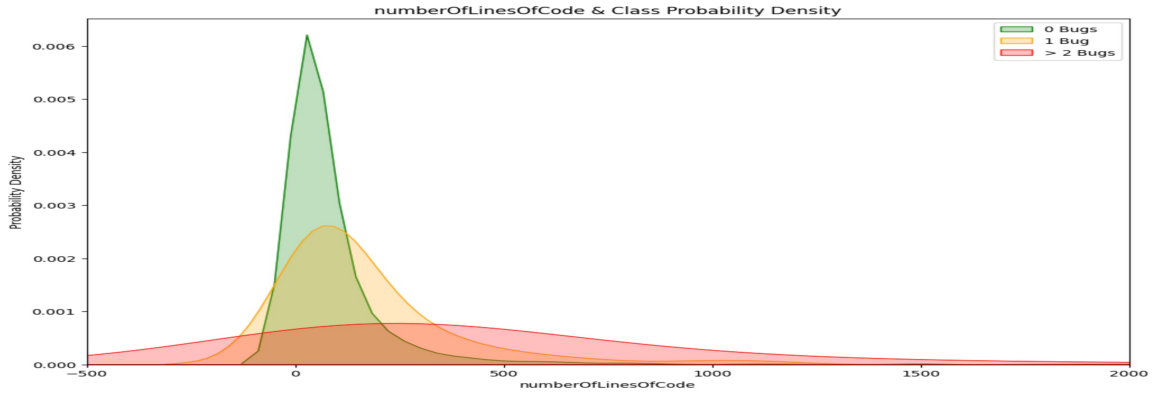


Figure 3.8: number of lines of code & class probability density

a 3D scatter plot visualizing the relationships between fanOut, rfc, and cbo for different bug classes (0, 1, and >2 Bugs). It uses color and marker size to represent bug classes and displays a legend with appropriate labels.

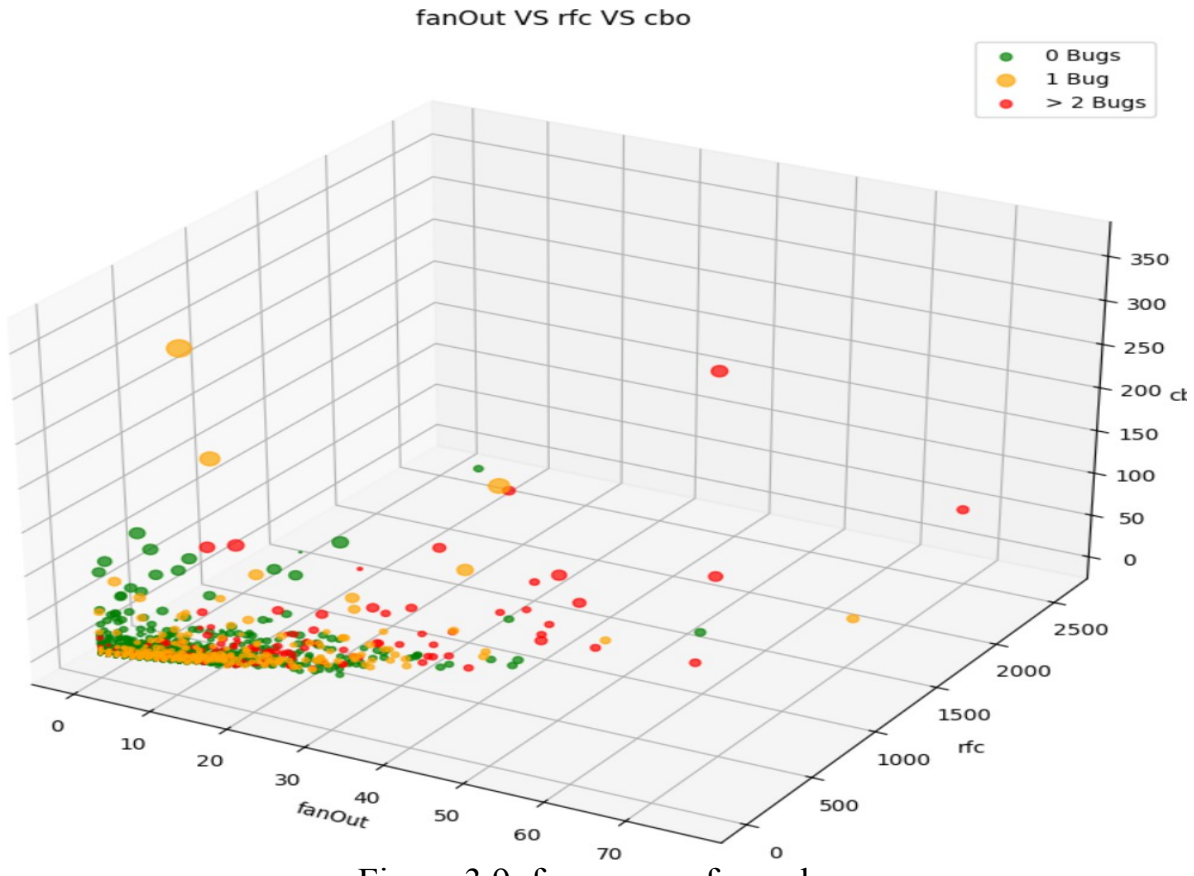


Figure 3.9: famous vs rfc vs cbs

a scatter plot to visualize the relationship between numberOfLinesOfCode and wmc for different bug classes (0, 1, and >2 Bugs). It uses color and marker size to represent bug classes and adds a legend with appropriate labels.

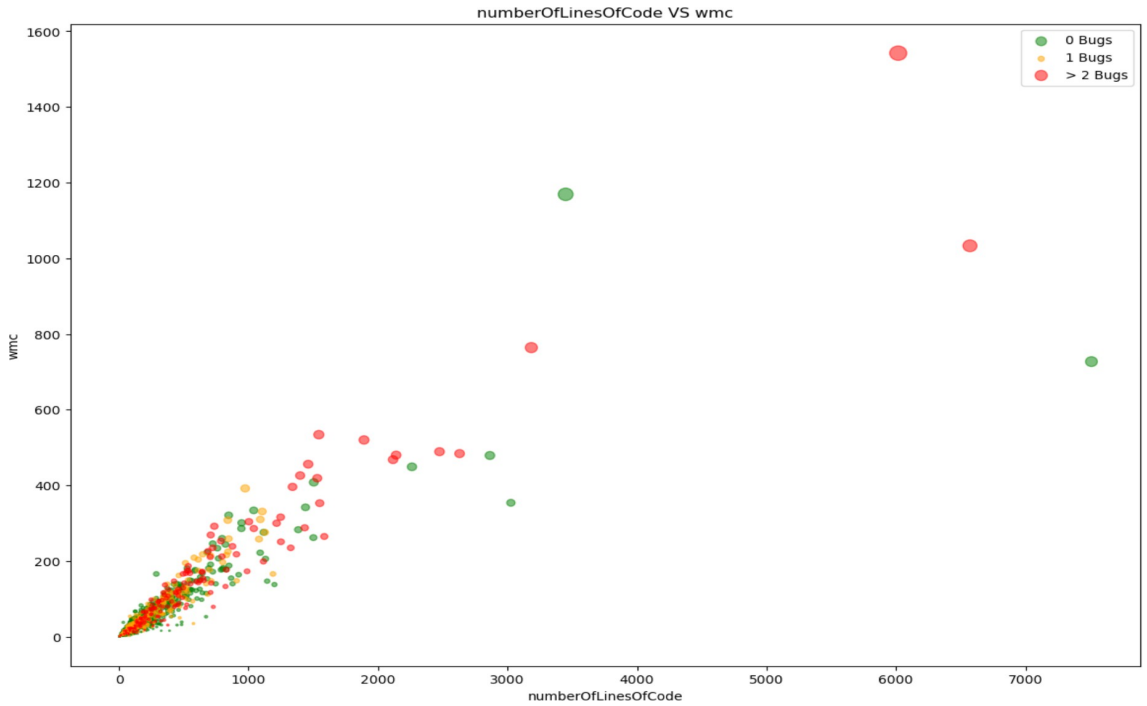
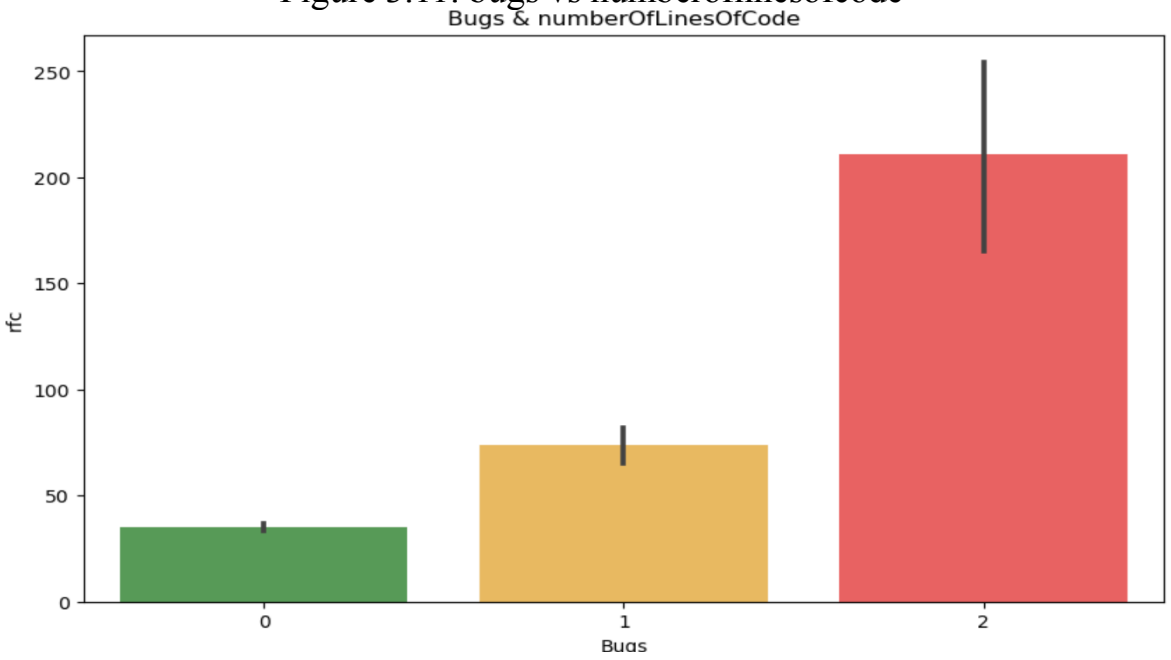


Figure 3.10 : numberslinesofcode vs cmc

a bar plot to visualize the average rfc values for different bug classes (0, 1, and >2 Bugs). It uses different colors to represent each bug class and adds a title to the plot.

Figure 3.11: bugs vs numberoflinesofcode



a bar plot showing "Bugs" on the x-axis and "wmc" on the y-axis, using three colors for different bug categories and adding a title.

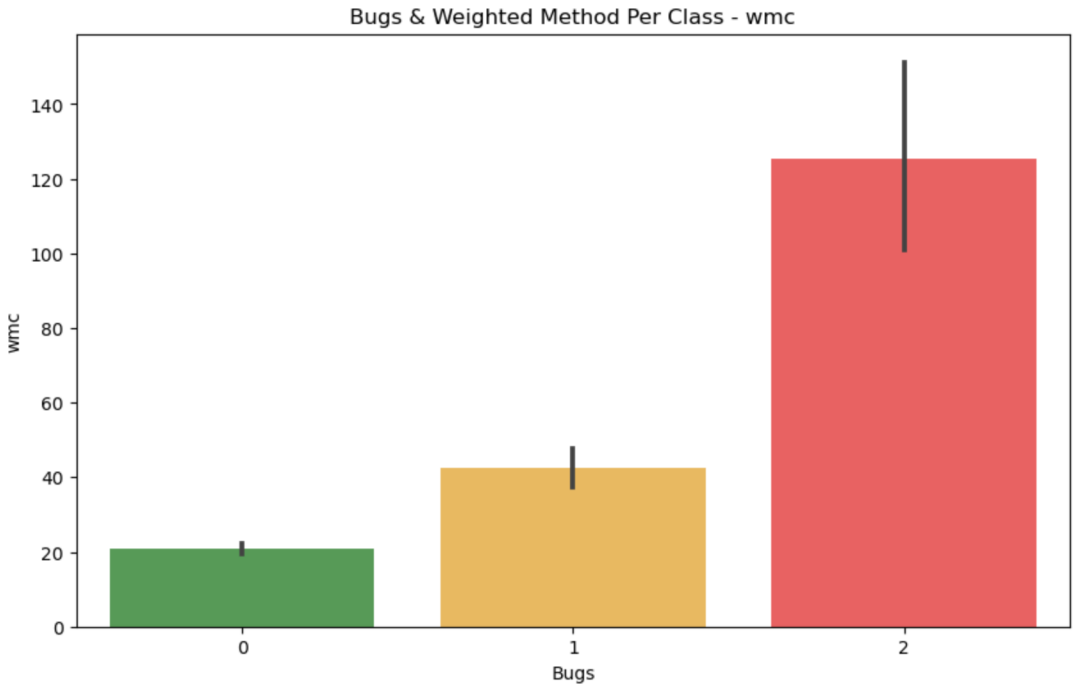


Figure 3.12: bugs & weighted method per class -wmc

The code generates a bar plot with "Bugs" on the x-axis and "numberOfLinesOfCode" on the y-axis, using different colors for each bug category and setting the title "Bugs & numberOfLinesOfCode."

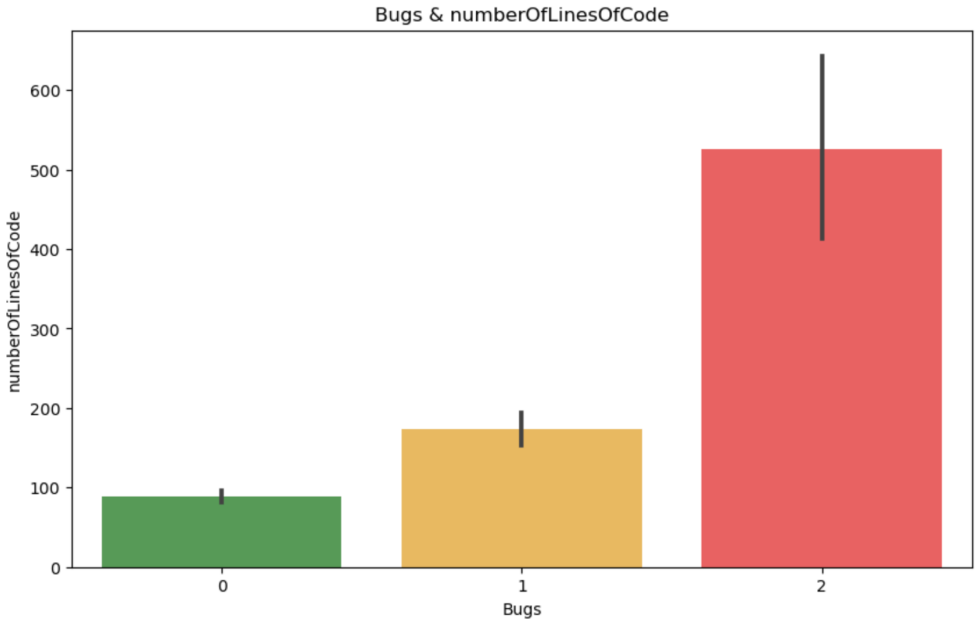


Figure 3.13: bugs & numberoflinesofcode

a bar plot to display the average cbo values for different bug classes (0, 1, and >2 Bugs). It uses distinct colors for each class and adds a title to the plot.

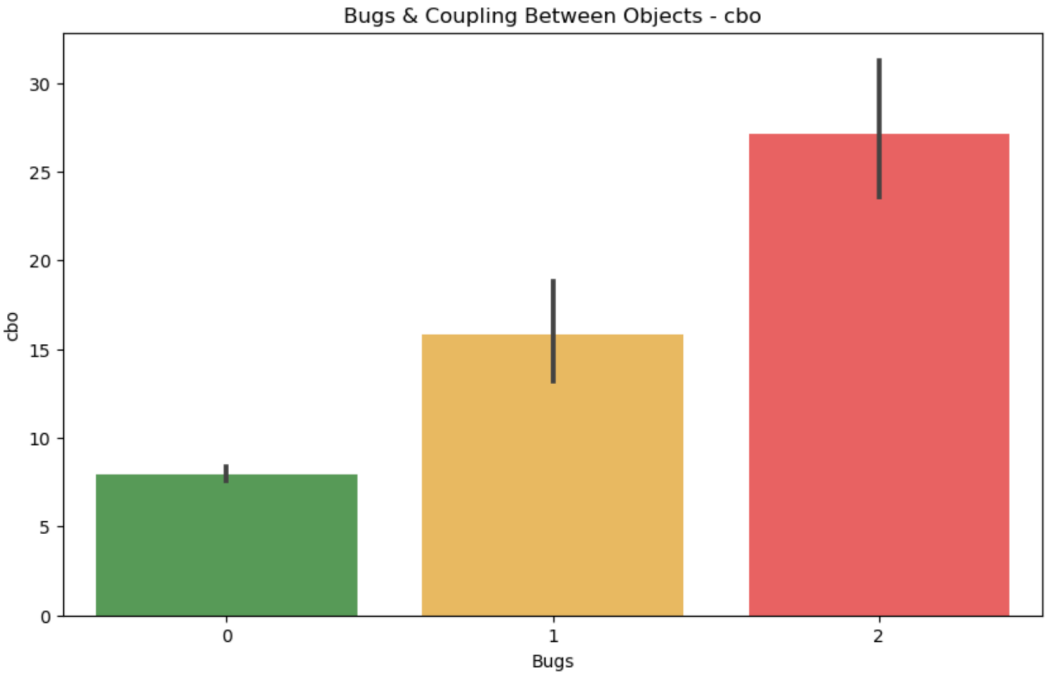


Figure 3.14: bugs & Coupling between objects -cbo

a bar plot to visualize the average fanOut values for different bug classes (0, 1, and >2 Bugs). It uses different colors for each class and includes a title for the plot.

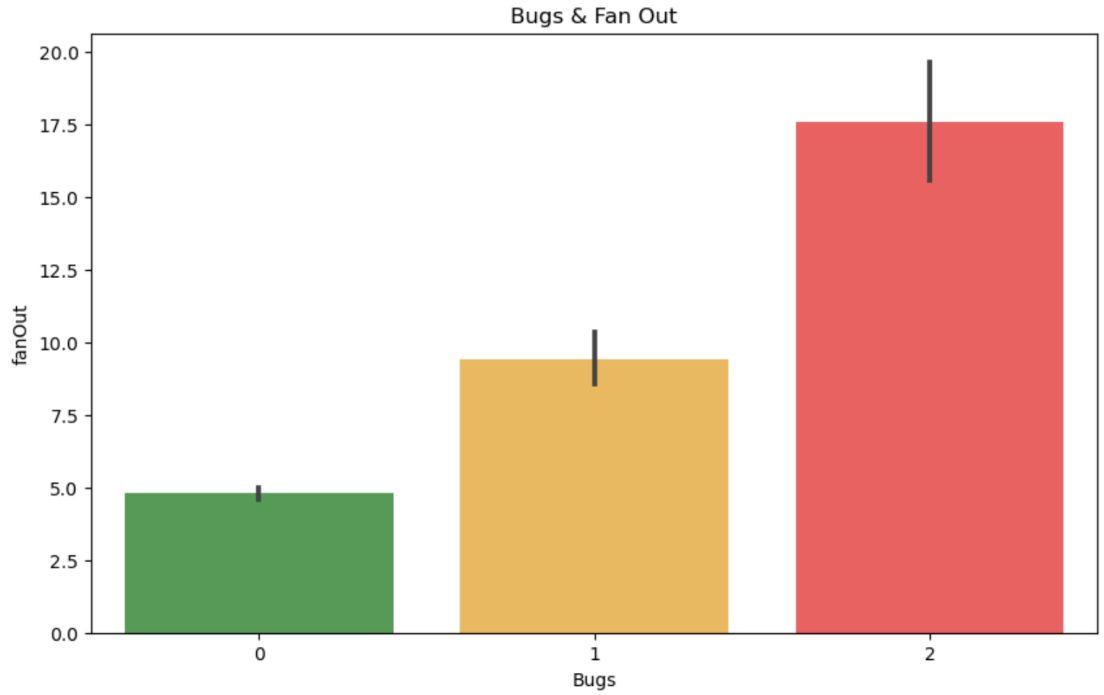


Figure 3.15: bugs & fan out

3. Model Building

3.1. train_test_split

```
[ ]  X_train_scaled = pd.DataFrame(StandardScaler().fit_transform(X_train.values), columns=X_train.columns, index=X_train.index)
     X_test_scaled = pd.DataFrame(StandardScaler().fit_transform(X_test.values), columns=X_test.columns, index=X_test.index)

     print("Train:", X_train.shape, y_train.shape,
           "Test:", X_test.shape, y_test.shape,
           "Cross Validation", X_cv.shape, y_cv.shape)
```

Train: (3759, 17) (3759,) Test: (806, 17) (806,) Cross Validation (806, 17) (806,)

The code standardizes the training and test datasets using StandardScaler, then stores the transformed data in new DataFrame objects with preserved column names and indices. The print statement outputs the shapes of the training, test, and cross-validation datasets, ensuring proper data alignment before model use.

a bar chart displaying the number of occurrences for each class with different colors. The plot is titled "#Bugs VS Occurrences" and labeled with the x-axis as "# Bugs" and the y-axis as “Occurrences."

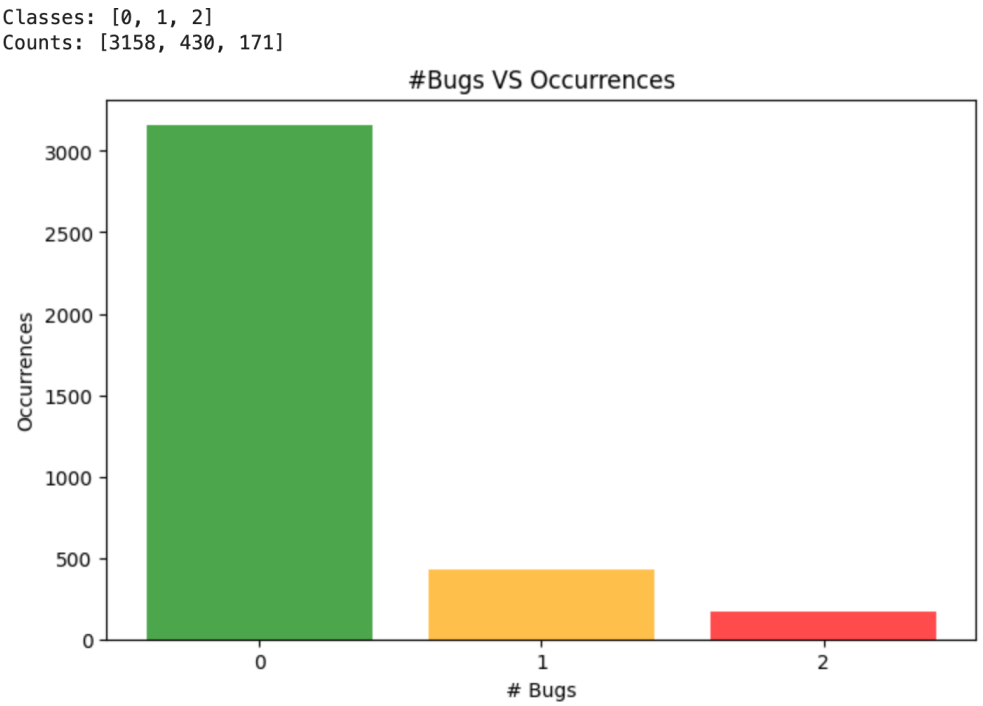


Figure 3.16 : bugs vs occurrences in training set

3.2. K-Nearest Neighbor

K-Nearest Neighbors (KNN) is a straightforward, non-parametric algorithm used for classification and regression tasks. It operates by identifying the 'k' closest data points to a new input based on a chosen distance metric, such as Euclidean or Manhattan distance. For classification, the algorithm assigns the input to the most common class among its nearest neighbors; for regression, it predicts the value by averaging the values of these neighbors.

Advantages:

- **Simplicity and Ease of Implementation:** KNN is simple to implement and intuitive to understand, making it accessible for various applications.
- **Versatility:** KNN can be used for both classification and regression problems, making it a versatile algorithm that can be applied to a wide range of use cases.

Disadvantage:

- **Computational Complexity:** KNN can be computationally intensive, especially with large datasets, because it requires calculating distances to all training samples for each prediction.

```
[ ]  knn_params = {
         'clf__n_neighbors': [3, 5, 11, 17], # Try not to put even numbers
         'clf__weights': ['uniform', 'distance'],
         'clf__metric': ['euclidean', 'manhattan'],
     }

     knn = Modeler(KNeighborsClassifier, X_train_scaled, y_train, X_cv, y_cv, knn_params, scale=True)

     Best Parameters for model: {'metric': 'manhattan', 'n_neighbors': 11, 'weights': 'uniform'}
```

The knn_params dictionary defines hyperparameters for the KNN model, such as n_neighbors, weights, and metric. The Modeler class trains and evaluates a KNeighborsClassifier on scaled data (X_train_scaled, y_train) with cross-validation on X_cv and y_cv. The scale=True ensures feature standardization before training.

3.3. Random Forest Classifier

A **Random Forest Classifier** is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of their classifications for new data. This approach enhances predictive accuracy and controls overfitting by averaging the results of various trees, each trained on different subsets of the data.

Advantages:

- **High Accuracy:** By aggregating the results of multiple decision trees, random forests often achieve higher accuracy compared to individual models.
- **Ability to Learn Non-linear Decision Boundaries:** Random forests can model complex, non-linear relationships between features and the target variable, making them versatile for various datasets.

Disadvantage:

- **Computational Complexity:** Random forests can be computationally expensive, particularly when working with large datasets, as they require significant memory and processing power to build and combine numerous trees.

```
[ ]  rfc_params = {
         'clf__n_estimators'      : [200, 500, 1000],
         'clf__max_depth'         : [10, 20, 50],
         'clf__max_features': [1.0, 0.7, 0.4],
         'clf__criterion' :['gini', 'entropy']
     }

     rfc = Modeler(RandomForestClassifier, X_train_scaled, y_train, X_cv, y_cv, rfc_params, scale=True)
```

```
Best Parameters for model: {'criterion': 'gini', 'max_depth': 20, 'max_features': 0.4, 'n_estimators': 500}
```

The rfc_params dictionary defines hyperparameters for tuning the Random Forest model. The Modeler class trains and evaluates the model with RandomForestClassifier, using scaled data (X_train_scaled, y_train) and cross-validation on X_cv, y_cv.

3.4. K-Means

K-Means is an unsupervised clustering algorithm that partitions data into 'k' distinct clusters based on feature similarity. It iteratively assigns data points to clusters by minimizing the variance within each cluster, aiming to ensure that points within the same cluster are as similar as possible.

Advantages:

- **Simplicity and Efficiency:** K-Means is straightforward to implement and computationally efficient, making it suitable for large datasets.
- **Scalability:** The algorithm performs well with large datasets, maintaining efficiency as the number of data points increases.

Disadvantage:

- **Sensitivity to Initial Centroids:** The outcome of K-Means can be significantly affected by the initial placement of centroids, potentially leading to suboptimal clustering results.

```
[ ] kmeans_params = {
        'clf__max_iter': [200, 500, 1000], # Try not to put even numbers
        'clf__n_init': [10, 30]
    }

    kmeans = Modeler(KMeans, X_train_scaled, y_train, X_cv, y_cv, kmeans_params, scale=True)
```

```
/usr/local/lib/python3.10/dist-packages/numpy/ma/core.py:2820: RuntimeWarning: invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
Best Parameters for model: {'max_iter': 200, 'n_init': 10}
```

The kmeans_params dictionary defines hyperparameters for tuning the KMeans model, such as max_iter (maximum iterations) and n_init (number of initializations). The Modeler class trains and evaluates the model using KMeans on scaled data (X_train_scaled, y_train) and performs cross-validation on X_cv, y_cv.

CHAPTER 4: RESULTS

Accuracy:

Accuracy measures the overall proportion of correct predictions made by the model, indicating how well it performs across all classes. It is calculated as:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Number of Instances}}$$

While accuracy provides a general sense of performance, it may be misleading in cases of imbalanced datasets, where the number of instances in different classes varies significantly.

ROC (Receiver Operating Characteristic):

ROC (Receiver Operating Characteristic) evaluates the model's ability to distinguish between different classes. The ROC curve plots the True Positive Rate (Sensitivity) against the False Positive Rate (1 - Specificity) at various threshold settings. The area under this curve, known as the AUC (Area Under the Curve), quantifies the overall ability of the model to discriminate between positive and negative classes. A higher AUC value indicates better performance.
The True Positive Rate (TPR) and False Positive Rate (FPR) are defined as:

$$\text{TPR} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

The ROC curve provides insight into the trade-off between sensitivity and specificity across different thresholds, making it a valuable tool for evaluating model performance, especially in scenarios with varying class distributions.

F1-Score:

F1-Score is the harmonic mean of precision and recall, balancing false positives and false negatives. It is particularly useful in imbalanced datasets, ensuring a model does not favor one class over another. The F1-Score is calculated as:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Where precision and recall are defined as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

The F1-Score provides a single metric that balances the trade-off between precision and recall, offering a more comprehensive evaluation of model performance, especially when dealing with imbalanced classes.

These metrics collectively offer a comprehensive understanding of a model's performance, each highlighting different aspects of its predictive capabilities.

Confusion Matrix:

The **Confusion Matrix** is a structured table used to evaluate the performance of classification models. It categorizes the model's predictions into four key outcomes:

- **True Positives (TP):** Instances where the model correctly predicts the positive class.
- **True Negatives (TN):** Instances where the model correctly predicts the negative class.
- **False Positives (FP):** Instances where the model incorrectly predicts the positive class.
- **False Negatives (FN):** Instances where the model incorrectly predicts the negative class.

ROC Curve: The ROC (Receiver Operating Characteristic) curve is a graphical representation of a classifier's performance at various thresholds. It plots the True Positive Rate (TPR) against the False Positive Rate (FPR). The Area Under the Curve (AUC) measures how well the model distinguishes between classes, with higher values indicating better performance.

1. K-Nearest Neighbor

K-Nearest Neighbor | Multi

Accuracy: 0.8486352357320099

ROC: 0.5630345939129245

F1-Score: 0.444136349701492

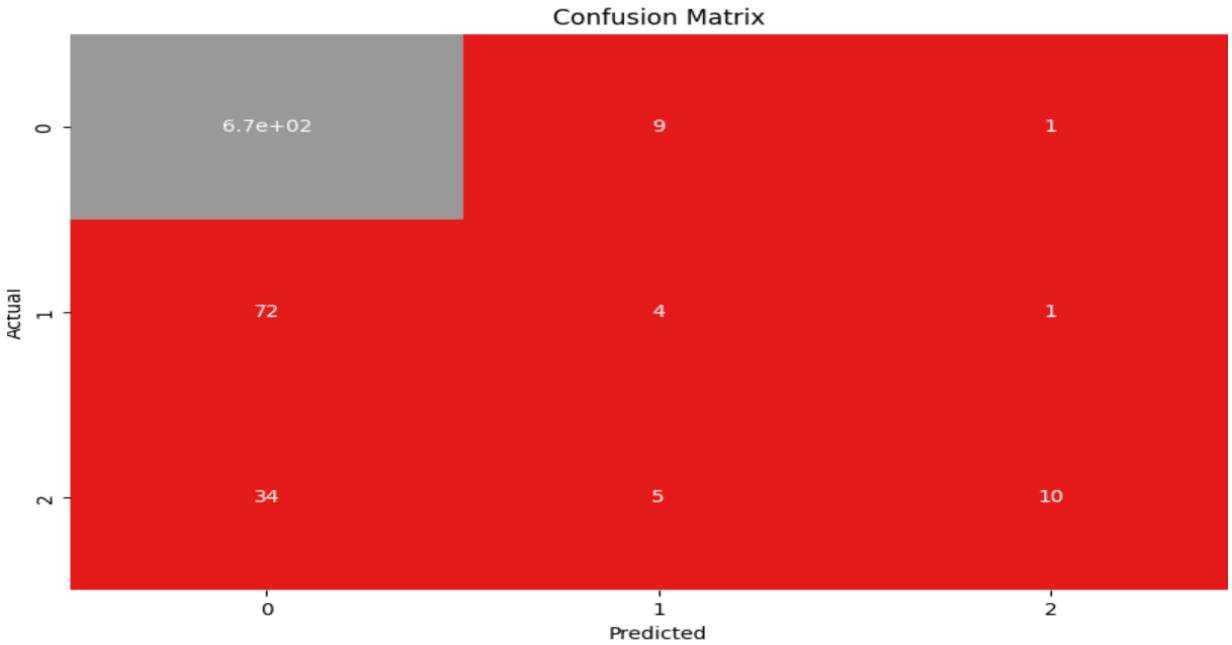


Figure 4.1: Confusion Matrix for K-Nearest Neighbour

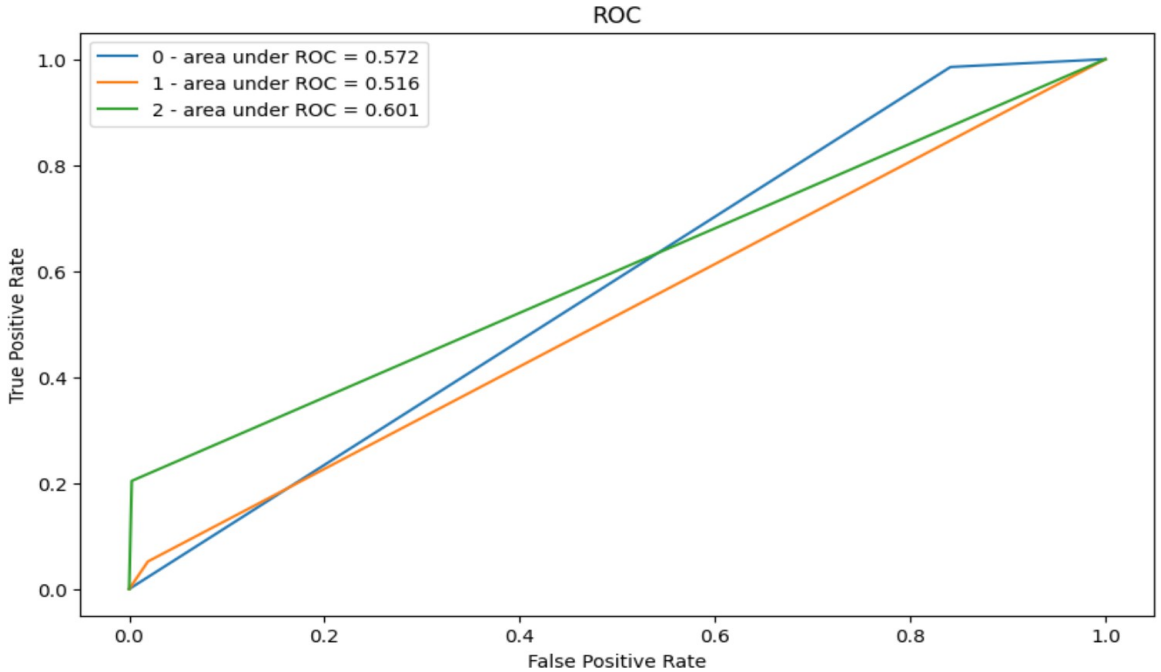


Figure 4.2: ROC for K-Nearest Neighbour

2. Random Forest Classifier

Random Forest | Multi:

Accuracy: 0.8511166253101737

ROC: 0.5672725742513184

F1 Score: 0.45758437432463017

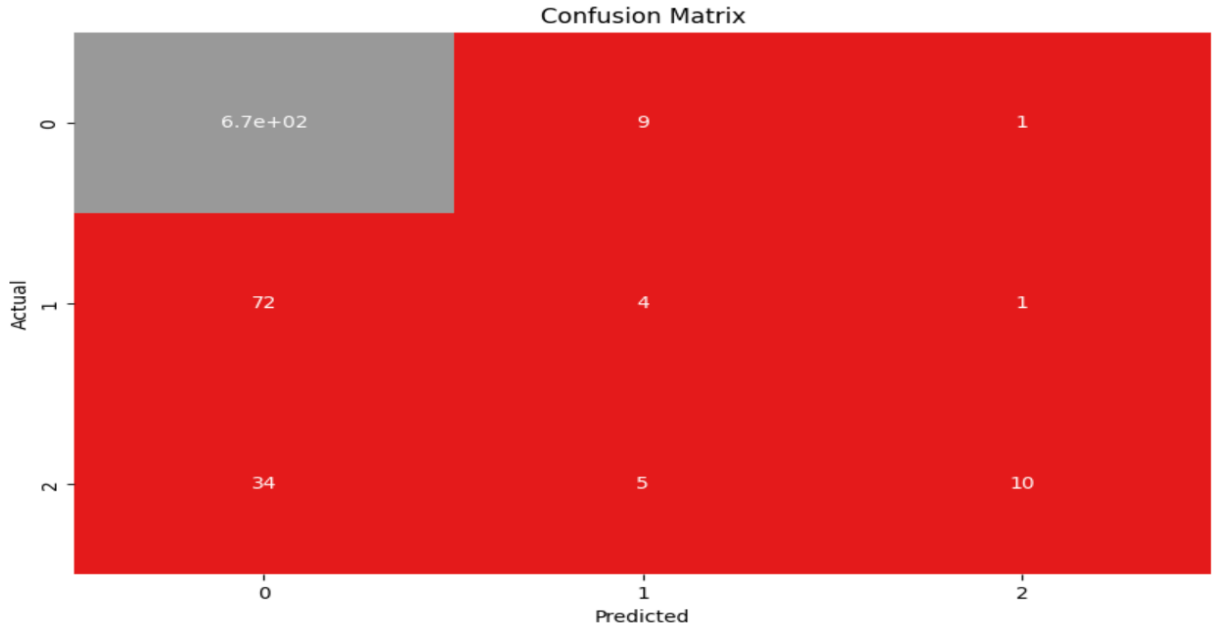


Figure 4.3: Confusion Matrix for Random Forest Classifier

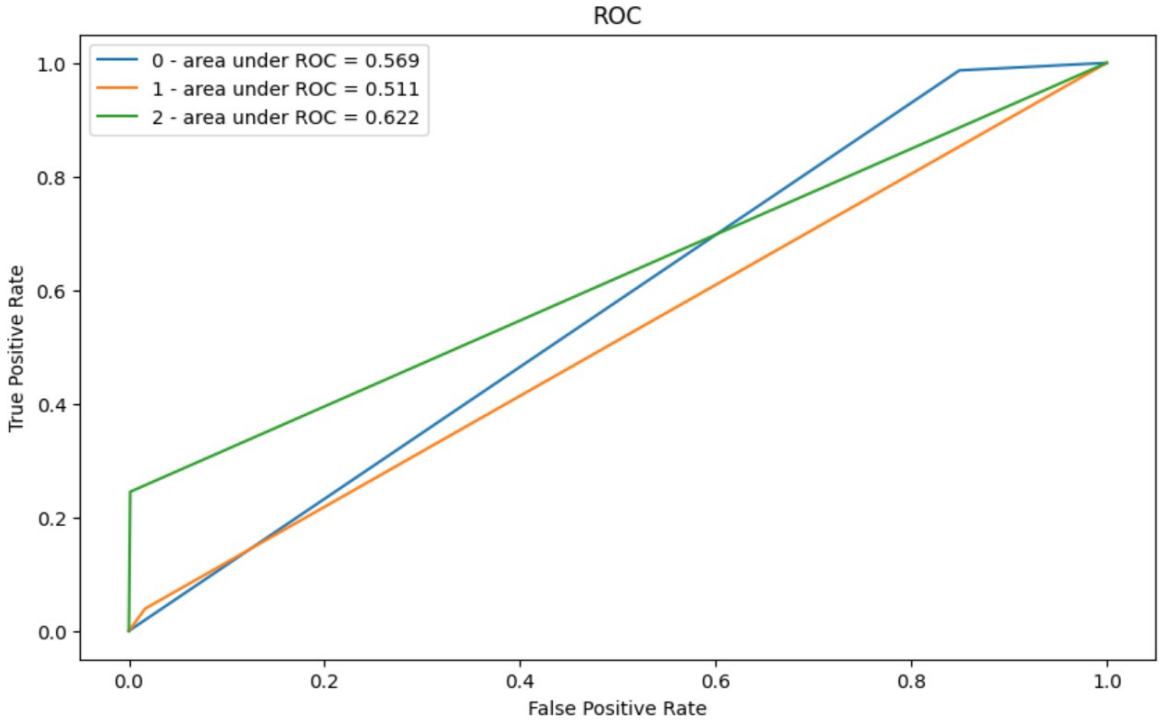


Figure 4.4: roc for Random Forest Classifier

3. K-Means

K-Means | Multi:

Accuracy: 0.6501240694789082

ROC: 0.532689499445535

F1 Score: 0.14787641572285146

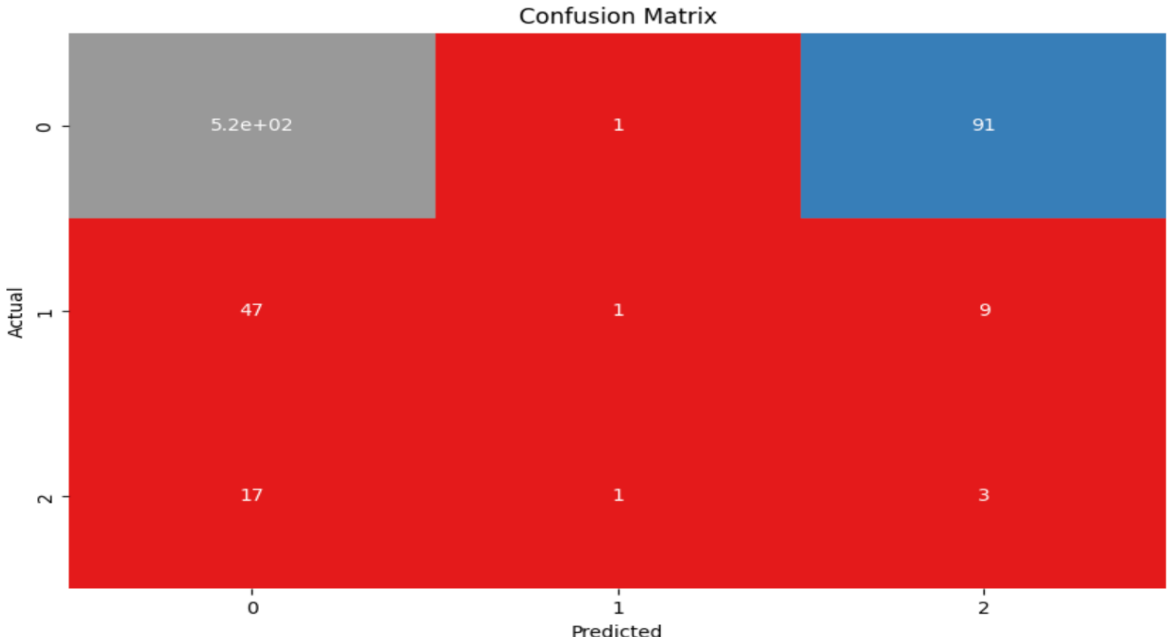


Figure 4.5: Confusion Matrix for K-Means

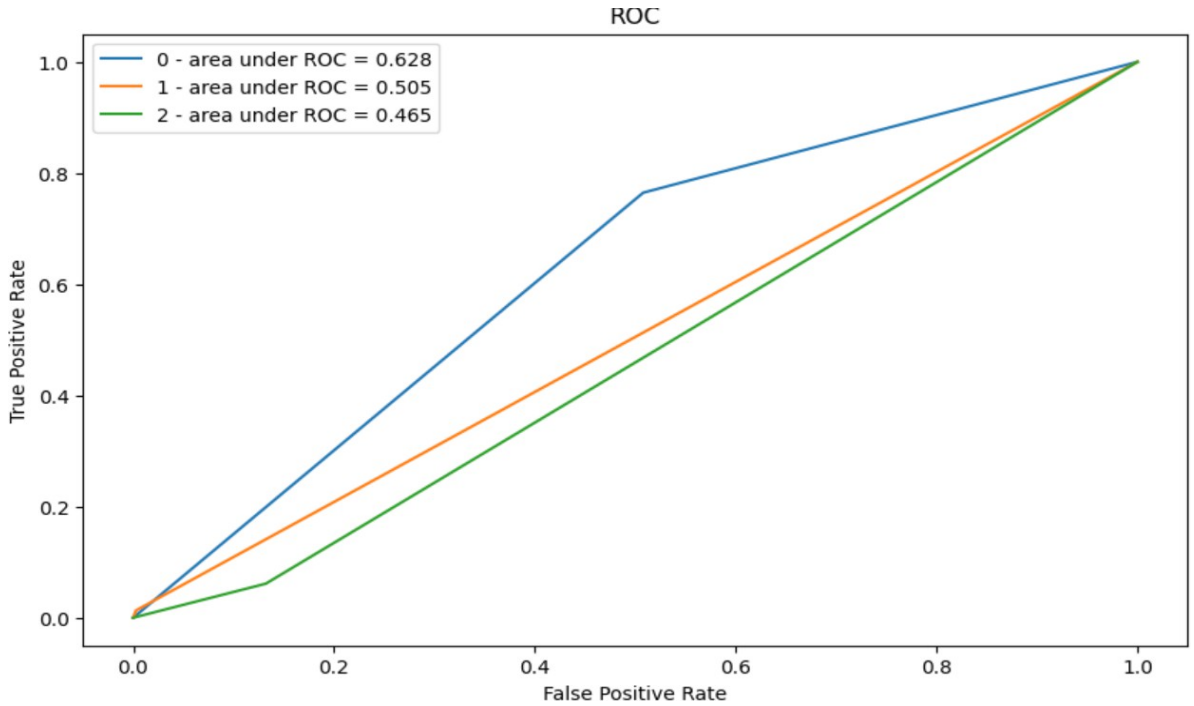


Figure 4.6: ROC for K-Means

CHAPTER 5 : APPLICATIONS

This project offers significant applications in software development and maintenance, particularly in automating bug detection and prioritization:

Bug Prediction & Detection

By analyzing software metrics such as Coupling Between Objects (CBO), Lines of Code (LOC), and Fan-Out, the model predicts which modules are likely to contain bugs. This enables developers to focus on high-risk areas, thereby improving overall software quality and reducing the likelihood of critical failures in production. The predictive nature of the model facilitates proactive interventions, preventing the escalation of potential issues.

Software Testing Efficiency

The model aids quality assurance by identifying modules that require more thorough testing, improving testing efficiency. By focusing testing efforts on areas with the highest likelihood of defects, testing teams can achieve better coverage with fewer resources. This targeted testing approach minimizes redundancy and ensures that critical areas are rigorously evaluated, leading to a more reliable software product.

Resource Allocation

By predicting bug-prone modules, resources such as developer time and computational power can be allocated more effectively, minimizing time spent on less critical areas. This strategic allocation ensures optimal utilization of available resources, reducing costs associated with unnecessary testing or debugging. Additionally, this allows project managers to prioritize efforts based on risk assessment, enhancing project management effectiveness.

Continuous Monitoring

The approach can be applied in Continuous Integration/Continuous Deployment (CI/CD) pipelines to flag potential issues early in the development process. By integrating the model into CI/CD workflows, development teams can achieve real-time monitoring and immediate feedback. This ensures that code quality is consistently maintained throughout the development lifecycle, reducing the risk of introducing defects into production.

Automated Debugging

The implementation reduces the need for extensive manual inspection, accelerating the software debugging process. By providing actionable insights into the root causes of detected bugs, the system streamlines the debugging process and reduces developer workload.

Early Risk Assessment

The model can be utilized during the initial stages of software design to identify potentially problematic areas based on design metrics. This helps mitigate risks before code is written, improving the reliability of the software architecture and reducing downstream costs associated with major redesigns or rework.

Enhanced Collaboration

By providing clear and data-driven insights, the system fosters better collaboration among development, testing, and quality assurance teams. The model's predictions can serve as a common reference point, aligning team efforts and improving communication throughout the software development lifecycle.

Integration with Software Analytics Tools

The model's output can be integrated with software analytics tools for comprehensive reporting and trend analysis. These integrations provide stakeholders with valuable insights into software performance and quality trends, aiding in long-term planning and decision-making.

Cost Optimization

By streamlining bug detection, testing, and debugging processes, the system significantly reduces costs associated with software maintenance and support. This makes it particularly valuable for large-scale projects with tight budgets and timelines.

Knowledge Enhancement for Teams

The insights generated by the system can be used to train and guide development teams on common pitfalls and best practices. By analyzing recurring patterns in bug-prone modules, organizations can foster a culture of continuous learning and improvement.

CHAPTER 6 : CONCLUSION & FUTURE SCOPE

1. Conclusion

This project successfully integrates machine learning models to predict bug-prone software modules based on various key metrics, such as Coupling Between Objects (CBO), Lines of Code (LOC), and Fan-Out. By leveraging these metrics, the project enhances software quality assurance processes, significantly reducing the time and effort required for bug detection and debugging. The implementation of these models allows for a more systematic and data-driven approach to identifying potential areas of concern within the codebase, enabling proactive measures to mitigate risks.

The machine learning models utilized in this project, including Random Forest and K-Nearest Neighbors (KNN), demonstrated strong performance and reliability, offering an automated solution to prioritize testing efforts. This efficiency-focused approach not only minimizes human intervention in the initial bug detection stages but also optimizes resource allocation during software development. As a result, the system contributes to a streamlined software development lifecycle, improving both efficiency and reliability while promoting higher software quality standards.

By addressing the challenge of early bug detection, the project lays a foundation for further advancements in automated quality assurance tools. This has a transformative impact on the software development process, ensuring that critical bugs are addressed early, thereby reducing overall project costs and enhancing customer satisfaction.

2. Future Scope

- **Integration with Deep Learning Models:** The use of advanced models like Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), or hybrid approaches has the potential to achieve even higher bug prediction accuracy. These models can capture intricate patterns within the software metrics, enabling more precise predictions and broader applicability.

- **Scalability:** Extending the study to larger and more diverse datasets from various software domains can enhance the system's robustness and generalization. By evaluating performance across a wide range of real-world applications, the predictive capability of the models can be further validated and improved.

- **Real-Time Implementation:** Integrating the developed system into real-world Continuous Integration and Continuous Deployment (CI/CD) pipelines for continuous bug detection and monitoring offers a seamless and practical application of this research. This real-time implementation ensures immediate feedback during software development, enabling developers to address issues promptly.

- **Feature Expansion:** Exploring additional software metrics, such as code churn, cyclomatic complexity, or real-time performance indicators, can lead to more precise predictions. These additional features would enable the models to account for dynamic aspects of software behavior and development practices, further refining bug prediction capabilities.

- **Automated Bug Management:** Beyond bug prediction, there is significant scope to develop tools that suggest and automate potential bug fixes. This involves leveraging machine learning to analyze historical bug fix patterns, offering developers actionable recommendations for resolving identified issues. Such automation can further reduce debugging time and improve developer productivity.

CHAPTER 7 : REFERENCESS

1. Y. Tohman, K. Tokunaga, S. Nagase, and M. Y., “Structural approach to the estimation of the number of residual software faults based on the hyper-geometric districation model,” IEEE Trans. on Software Engineering, pp. 345–355, 1989.

2. A. Sheta and D. Rine, “Modeling Incremental Faults of Software Testing Process Using AR Models ”, the Proceeding of 4th International Multi-Conferences on Computer Science and Information Technology (CSIT 2006), Amman, Jordan. Vol. 3. 2006.

3. D. Sharma and P. Chandra, "Software Fault Prediction Using Machine-Learning Techniques," Smart Computing and Informatics. Springer, Singapore, 2018. 541-549.

4. R. Malhotra, "Comparative analysis of statistical and machine learning methods for predicting faulty modules," Applied Soft Computing 21, (2014): 286-297

5. Malhotra, Ruchika. "A systematic review of machine learning techniques for software fault prediction." Applied Soft Computing 27 (2015): 504-518.

6. D'Ambros, Marco, Michele Lanza, and Romain Robbes. "An extensive comparison of bug prediction approaches." Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on. IEEE, 2010

7. Gupta, Dharmendra Lal, and Kavita Saxena. "Software bug prediction using object-oriented metrics." Sādhanā (2017): 1-15..

8. M. M. Rosli, N. H. I. Teo, N. S. M. Yusop and N. S. Moham, "The Design of a Software Fault Prone Application Using Evolutionary Algorithm," IEEE Conference on Open Systems, 2011.

9. T. Gyimothy, R. Ferenc and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," IEEE Transactions On Software Engineering, 2005.

10. Singh, Praman Deep, and Anuradha Chug. "Software defect prediction analysis using machine learning algorithms." 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence, IEEE, 2017.

11. M. C. Prasad, L. Florence and A. Arya, "A Study on Software Metrics based Software Defect Prediction using Data Mining and Machine Learning Techniques," International Journal of Database Theory and Application, pp. 179-190, 2015.

12. Okutan, Ahmet, and Olcay Taner Yıldız. "Software defect prediction using Bayesian networks." Empirical Software Engineering 19.1 (2014): 154-181.

13. Bavisi, Shrey, Jash Mehta, and Lynette Lopes. "A Comparative Study of Different Data Mining Algorithms." International Journal of Current Engineering and Technology 4.5 (2014).

14. Y. Singh, A. Kaur and R. Malhotra, "Empirical validation of objectoriented metrics for predicting fault proneness models," Software Qual J, p. 3–35, 2010.

15. Malhotra, Ruchika, and Yogesh Singh. "On the applicability of machine learning techniques for object oriented software fault prediction." Software Engineering: An International Journal 1.1 (2011): 24-37.

16. A.TosunMisirli, A. se Ba¸ S.Bener,"A Mapping Study on Bayesian Networks for Software Quality Prediction", Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, (2014).

17. T. Angel Thankachan1, K. Raimond2, "A Survey on Classification and Rule Extraction Techniques for Data mining",IOSR Journal of Computer Engineering ,vol. 8, no. 5,(2013), pp. 75-78.

18. T. Minohara and Y. Tohma, "Parameter estimation of hyper-geometric distribution software reliability growth model by genetic algorithms", in Proceedings of the 6th International Symposium on Software Reliability Engineering, pp. 324–329, 1995