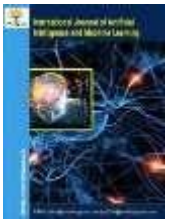
SvedbergOpen
DISSEMINATIONOF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/



Research Paper Open Access

Effective Software Metrics Prediction for Bug Detection Using Machine Learning

Sushma Saini[1], Jai Bhagwan[1]*, Seema Rani[2], Sanjeev Kumar[1]

[1]Department of Computer Science & Engineering, Guru Jambheshwar University of Science & Technology, Hisar – 125001, India.
[2]Department of Computer Science & Engineering, Ch. Devi Lal State Institute of Engineering & Technology, Sirsa, India.

*Corresponding Author E-mail Address: drjaicse@gmail.com, ORCID: 0000-0002-8708-4029

Abstract

In the corporate world, with the growth of software business, it is desired to improve the overall software quality. For any software development organization to increase software quality, metrics are a necessary component. Cost-effective test strategy planning and monitoring require the measurement of a software process. Software metrics offer a quantitative method for creating and verifying software process models. Metrics give an organization the information it needs to keep increasing efficiency, cut down on errors, and boost customer acceptability of procedures, goods, and services—all while achieving the intended outcome. This research analyzes the most effective software metrics for bug prediction on various projects NASA datasets: PC1, CM1, KC2 by using machine learning techniques. Most effective metrics contribute significantly to determining bug prediction and help in achieving software reliability. We have performed experimental analysis on different types of performance metrics like precision, recall, F1_score, accuracy and ROC AUC for machine learning models like XGBoost, Logistic Regression (LR), Support Vector Machine (SVM), Random Forest (RF), and Gradient Boosting Classifier. The feature selection selects important subsets of top 15 metrics among 21 metrics using different ML techniques. The Gradient Boosting feature selection on the PC1 dataset achieved the highest accuracy of 89.8%. The Gradient Boosting feature selection on the CM1 dataset achieved the second highest accuracy of 85%.

Keywords- Bug Prediction, Effective Software Metrics, Feature Selection, Gradient Boosting, Software Metrics, SMOTE, XGBoost

1. Introduction

A software failure is when the system deviates from its required behaviour. An error is the mismatch between expected and actual functionality, and the hypothesized cause of an error is called a fault or bug. Software errors consume significant time and resources. Learning from past experience can help predict bugs in new products. Defect prediction models are vital for the early identification of defect-prone modules within the Software Development Life Cycle (SDLC). This early identification is necessary to produce reliable, high-quality software on time and within budget [4, 16]. Metrics play a crucial role for any software development organization striving to enhance software quality. Additionally, measuring software processes is essential for the planning and monitoring of cost-effective testing strategies. Analysing metric trends helps in taking appropriate actions for process improvement and in stills confidence in the software's reliability and performance [18]. A metric is a common unit of measurement used to quantify outcomes. Software metrics are measurements used to assess software processes, products, and services. Software metrics are measurement-based techniques that are used to engineer and manage information about processes, products, and services. The information is processed to improve the processes, products, and services, if necessary [17].

1.1 Software Metrics

Software metrics provide numerical analyses that assign numerical values or symbols to characteristics of the subject under measurement. An entity's measurable qualities or aspects are represented by its attributes, which include length, age, and cost. Applications' source code or tasks performed during the software development process are examples of entities. Defect prediction models can be developed by taking measurements from these entities and connecting them to risk variables. These models aid in detecting potential problems and raise the standard of the software development process as a whole [20]. Finding and fixing problems early on saves a lot of money. It is much cheaper to correct defects during development than to fix them after the product is released [22, 40].

They contribute valuable information to decision-making at different stages of the software lifecycle, which reduces risk. A major component of the software development process is software testing [7]. A high degree of confidence in software reliability may arise from extensive testing and metrics evaluation because software is always expanding in terms of size, power, and complexity. In order to lower the danger of software release schedule overruns, test metrics results are utilized to assess the present state of a project and aid in the prioritization of different tasks. The subsequent software version is frequently altered, and regression testing is carried out as a result of new additions or client-end testing [18]. The metrics fall into three categories: project, process, and product metrics. Software process metrics, commonly referred to as quality metrics, measure the characteristics of the software. These metrics include cost, size, complexity, style, efficacy, portability, reusability, usability, functionality, performance, and reliability. These metrics assess the size, documentation, and complexity of the program design [23]. A wide range of abilities and tasks are necessary for effective test management, such as identifying, gathering, and analyzing various test-related and quality-related metrics as well as metrics linked to tracking, managing, and controlling test status; conducting appropriate reviews of test documentation and supporting materials; and establishing precise standards for impartially determining when a system is prepared for operational use and when it is ready for piloting [33, 37].

1.2 Importance of Metrics

Metrics are used to increase the productivity and quality of goods and services, which in turn leads to higher customer satisfaction. It is easy for management to understand a single figure and go deeper if necessary. Various trend metrics serve as a watchdog when a process spirals out of control. Metrics help improve the present process [17]. It is not an original idea to visualize metrics, which is the process of presenting data in a way that makes it easier for readers to comprehend and/or analyze the information.

Table1: Element of Metrics

Element	Description
Metric	The metric's name that will be applied
Metric Description	An explanation of the measurement
Measurement Procedure	What is the measurement method for the metric
Measurement Frequency	How frequently is the measurement conducted
Current Thresholds	The current range of values that the metric considers normal
Target Value	Maximum value that the metric can have
Thresholds Estimation	How are the cutoff points determined
Units	Measurement units

When visualization is done well, it may show concepts that were hidden or ignored before. It is much more than just a collection of various graph kinds. Making better decisions may result from knowing this information. Metrics are used in software development projects for four main purposes: 1) to inform the project manager about the project's status in terms of completion; 2) to provide data for decision-making; 3) to serve as the foundation for estimates for subsequent projects; and 4) to inform management about the final product's quality and dependability [33]. Before being used, software metrics should be well described; Table 1 lists the components that need to be done so. Software metrics allow us to - Analyze requirements, Forecast development resources, Keep tracks on growth progress and to understand maintenance expenses [21].

1.3 Metrics Lifecycle

The steps used for setting the metrics:

Analysis: Identify which metrics to utilize, what metrics have been identified, and what parameters should be used to identify the identified metrics. Fig. 1 shows metrics lifecycle process [17].

Communicate: Describe the significance of metrics to the testing team and stakeholders. Teach the testing team how to record crucial information for processing metrics.

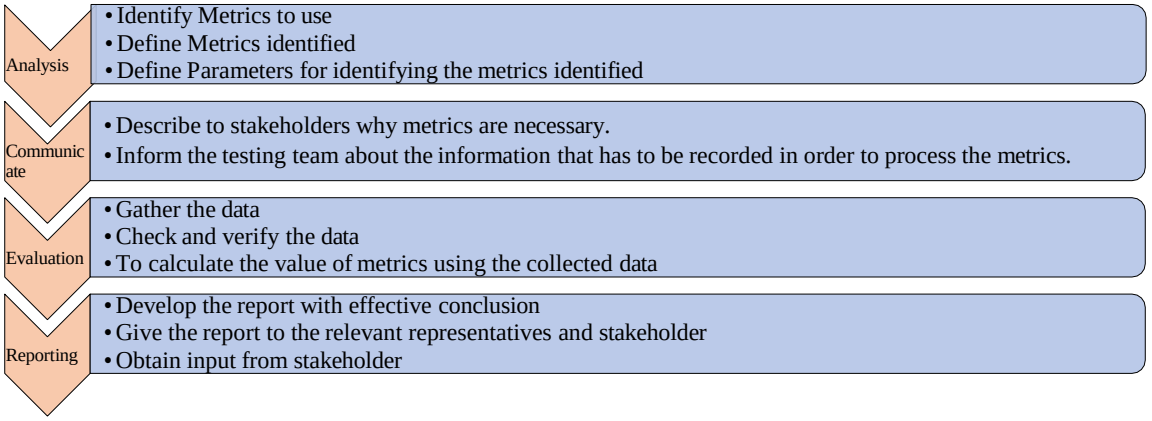
Evaluation: Data needs to be captured and verified. Calculating the metric values based on the gathered data.

Report: Write a strong conclusion. Distribute the report to key stakeholders and representatives. Gather their input for final review.

RQ1. Which are the most effective software metrics with different ML techniques for predicting bugs?

RQ2. What is each ML classifier's ability to predict bugs in different datasets?

Fig. 1: Software Metrics Lifecycle



2. Software Complexities

Software complexity refers to how challenging it is to use and understand software. The degree to which features that make software maintenance difficult exist and are influenced by program complexity is known as software maintainability [24]. Well-known software metrics serve as the foundation for software complexity [23]. Instead of using intuition, software complexity should be assessed using accessible, valid, dependable, and objective measurements. These measurements could be helpful in 1) Creating quality standards or contractual obligations, 2) Verifying software specification compliance, and 3) Designing trade-offs between development and maintenance expenses [29]. Fig. 2 shows the importance of software Complexities. Various measurement scales could be used, including:

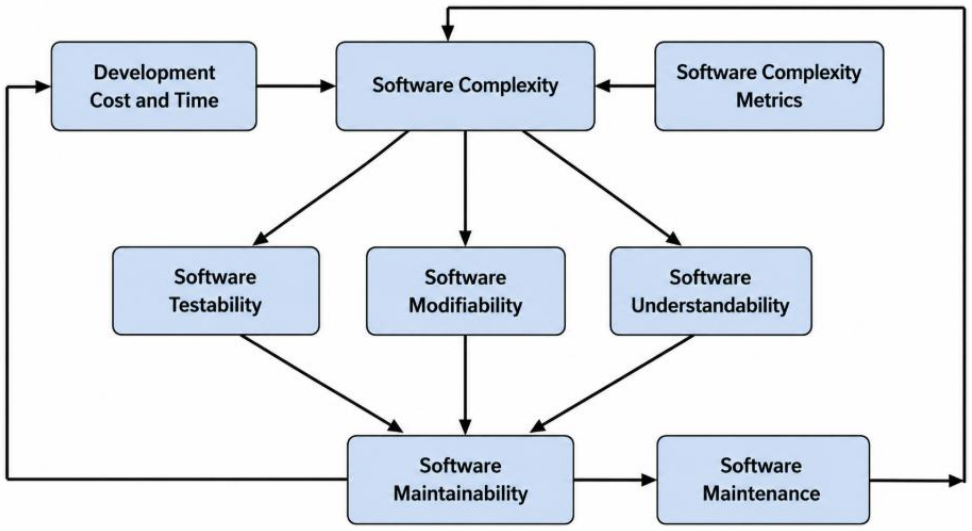


Fig. 2: Importance of Software Complexity

a) Ordinal Scale: To provide a somewhat basic comparison, the complexity of various applications has to be ranked [29].
b) Interval Scale: Two programs' differences in complexity are represented in units; program B is six units more difficult than program A, for instance [29].
c) Ratio Scale: The ratio between two programs' levels of complexity is established. Although flexible, the ratio scale is not useful [29].
d) Nominal Scale: The most basic type of measurement is this one. If a scale assigns no specific order to the categories into which the set of entities is divided, it is considered nominal [21].
e) Absolute Scale: The measurement scale hierarchy's most informative scale is the absolute scale. The counting method is used for measurement in an absolute scale [21].

The characteristics that are employed for measurement allow software complexity measures to be identified. Fig. 3 shows the classification of complexity metrics [29]. The static measure can be classified into three types:

2.1 Size Based Metrics

One of the most important characteristics of software systems is size, which manages how much money and manpower are spent on the systems' development and upkeep. Size-based metrics measure the actual size of the software module and are derived from basic counts like line numbers, effort, volume, size, length, etc. These metrics stipulate the complexity of software by size or volume attributes and aid in forecasting the cost of system maintenance [23].

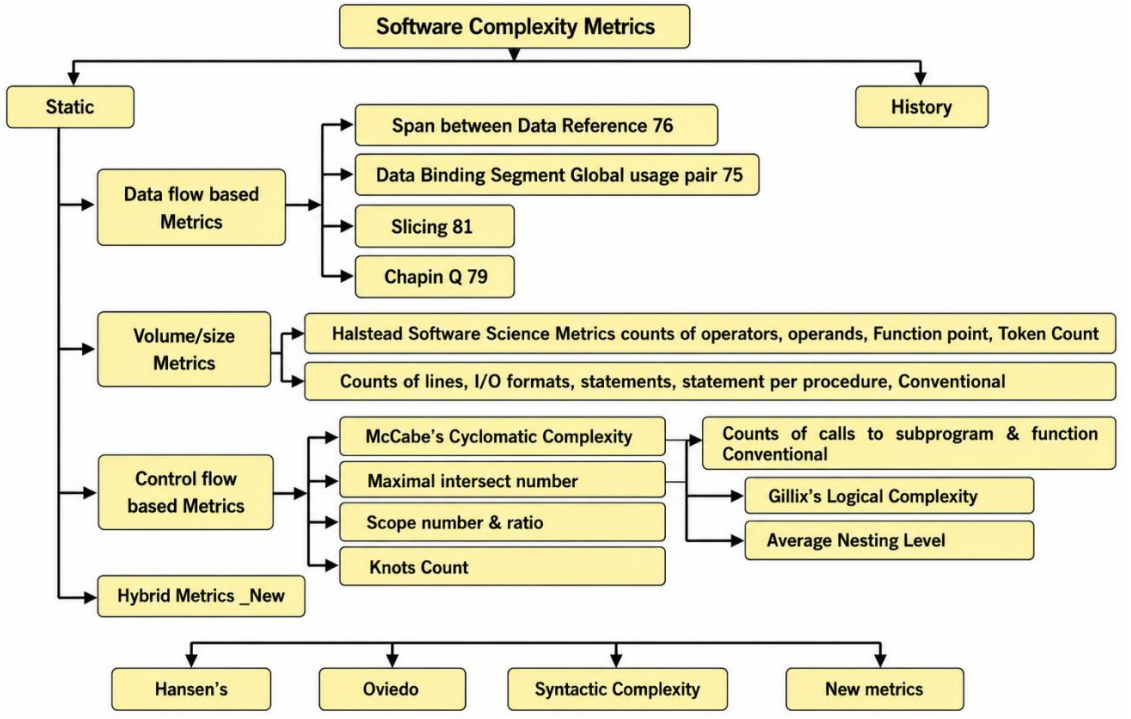


Fig. 3: Classification of Complexity Metrics

2.1.1 Conventional Metrics

The complexity of development is revealed by Lines of Code (LOC), a measure of program size. Hatton (1977) proposed the simplest software complexity metric. This measure is rather simple to use and counts the number of source instructions required to solve a problem [23]. LINES: It contains all source lines, including comments, specifications, executable statements, compiler control, and I/O formats [29].

STMTS: It includes executable statements such as arithmetic statements, functions, assignments, control, and I/O [29].

FOMTS: Indicates how many I/O formats there are [29].

UNITS: It calculates programming modules: subroutines, functions, etc [29].

LN- CM: It excludes comments and counts all source lines [29].

STMT/Unit: It is a programming module's average length [29].

2.1.2 Halstead's Software Metrics

When determining the program length, vocabulary, volume, potential volume, estimated program length, difficulty, effort, and time, these operators and operands must be taken into account [23].

Operators and Operands: It involves metrics defined by some key constituents of a program implementing an algorithm. Operator consists of Basic operators, Keywords, and special classes. All variable names and constants, including REAL, FALSE, and TRUE [29], are considered operands.

2.2 Control Flow Based Metrics

Metrics based on control flow assess how understandable control systems are. These metrics also define how a program's logic structures relate to its overall complexity. These metrics are derived from a program's control structure [24]. The control structure of a program serves as the basis for the control flow complexity measurements. Nejmeh created the control flow measure via NPATH, a metric that counts the number of execution pathways across a function and measures acyclic execution paths. One illustration of a control flow metric is NPATH [23].

Gilb: The two metrics that Gilb suggests - There is some empirical support for CL, or absolute logical complexity (number of binary decisions), and cL, or relative logical complexity (ratio of CL to STMTS). Since the latter takes into consideration some volume metrics, it may be considered an enhancement above pure control metrics [29].

McCabe Cyclomatic Complexity: One statistic that is focused on information/control flow rather than program size is the Cyclomatic Number. Its foundation is the specification flow graph format that Thomas J. McCabb created in 1976. Control flow is shown using a program graph. Processing tasks are represented by nodes, while control flow between nodes is represented by edges. One type of control flow measure is McCabe's metrics [23]. It is calculated as:

$$V(G) = \text{edges} - \text{nodes} + 2 \times \text{units} \quad (1)$$

KNOTS Count: Since each node is a series of statements without internal branches, a knot forms where two control transfers converge. There are two related metrics including KNOT1-The quantity of knots that can be proven and KNOT2- the overall count of possible knots, presuming that each node has a single statement [29].

Scope and Ratio: RECEIVING nodes are those with an out-degree of 0 or 1. SELECTION ones are those whose out-degree is more than 1. Given the selection node, it is feasible to find at least one "lower bound" node that follows each of its immediate successors. The lower-bound node that precedes all other lower bounds is known as the GREATEST LOWER BOUND (GLB). The number of nodes that appear before and after the GLB plus one is the ADJUSTED COMPLEXITY (AC) of that selection node. It displays the chosen node's "influence" range. The SCOPE metric is created by adding up each node's adjusted complexity [29].

2.3 Data Flow Based Metrics

Data flow-based metrics quantify how data is used and how dependent on it (both in terms of data visibility and interactions) [24].

2.4 Hybrid Metrics_NEW_1

Provide a unique hybrid metric that integrates software science with the breadth of measure and represents both volume and control organization. E_j is the raw complexity of Node V_j.

$$E_j = N_j \text{Log} n_j / L \quad (2)$$

Where L is a previously established global parameter and N_j and n_j are local parameters of node V_j.
A selection node's adjusted complexity is equal to the sum of the $E\hat{}_j$ values of all the nodes that fall inside its SCOPE as well as the selection node's value [29].
Define the metric NEW_1 as:

$$\text{NEW\_1} = (1.0 - \sum \text{Raw Complexities} / \sum \text{Adjusted Complexities}) \times 100\% \quad (3)$$

with increasing complexity, NEW_1 rises towards 100 percent.

3. Literature Review

Nowrin Muhaimin Shailee et al. [1] highlight how important it is to use machine learning for early bug finding in software development. R. Siva et al. [2] present a novel approach for software bug prediction that combines the deep learning algorithm and the metaheuristic optimization method. To enhance the Bandit Algorithm technique, Yukasa Murakami et al. [3] proposed a model to decrease the omission of faults, particularly in the early stages of testing. Kaibo Liu et al. [38] suggest AID, an automated method that combines the power of LLM and differential testing to create test cases that discover defects in plausible programs. Guisheng Fan et al. [5] introduced DP-AM, which uses an RNN to generate semantic features from code, applies self-attention and global attention for crucial features, and integrates static metrics to enhance defect prediction. Systematic Literature Review (SLR) consists of three stages: Review of Plan, Review of Conduct, and Review of Document, as shown in Fig. 4 [35].

DongGyun Han et al. [6] presented 56 unique micro-interaction metrics (MIMs) that make use of the interaction data that developers have saved in the Mylyn data. Lucija Sikic et al. [39] proposed new aggregated change metrics, created by chronologically aggregating data from all software changes between two versions. Taek Lee et al. [8] presented that MIMs boost defect prediction accuracy, are cost-effective, and offer intuitive feedback, helping developers identify inefficient behaviors in software development. Stefano Dalla Palma et al. [9] proposed a comprehensive machine-learning framework for repository crawling, metrics gathering, model building, and IaC defect prediction assessment. Dimah Al-Fraihat et al. [10] introduced if Ensemble Learning models outperform single models in bug prediction and examined the impact on their accuracy. Xiao Cheng et al. [11] proposed BTP metrics, utilizing bug-triggering paths, to quantitatively and precisely evaluate learning-based vulnerability

detection, complementing traditional metrics. Sikandar Ali et al. [12] investigated the significance of code smell metrics in prediction models for identifying bug-prone code modules. Sadia Sahar et al. [13] presented the DP-CCL model, which uses a language model that has been trained to extract the source code's semantic properties. Misbah Ali et al. [14] presented an efficient ensemble-based strategy for predicting software defects by combining various classifiers.

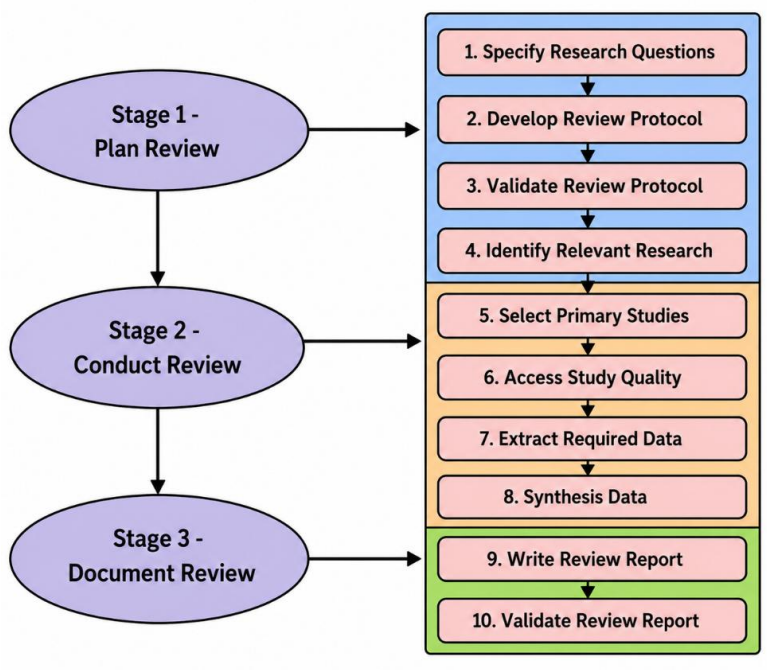


Fig. 4: Process of Systematic Literature Review

Dimah Al-Fraihat et al. [15] findings show EL models surpass single hypothesis models, and the proposed model's accuracy improves post-optimization. K. Punitha et al. [16] said the goal of their research is to increase the data mining algorithm's classification accuracy. Premal B Nirpal et al. [17] highlight the metrics lifecycle, software testing metrics, their importance, evaluation process, and ideal conclusions. Yogesh Singh et al. [18] presented surveys, classification, and propose new software product testing metrics. Yguarat˜A Cerqueira Cavalcanti et al. [19] presented an article that looks into, describes, and suggests a solution for the issue of duplicate bug reports.

Review of Plan: Develop specific research questions, create a structured review procedure, and thoroughly examine it to ensure accuracy and reliability. This methodical technique improves the evaluation process, reduces bias, and increases the trustworthiness of the findings. A well-defined technique serves as a firm platform for reaching significant conclusions and enhancing overall study quality.

Review of Conduct: During the implementation phase of a systematic literature review, discover relevant research and carefully choose primary papers to analyze. Evaluate the quality of these studies to ensure credibility and dependability. Extract relevant data and carefully synthesize the findings to provide a well-organized and accurate review. This procedure ensures a thorough review of current material, resulting in valuable insights and well-founded conclusions. By taking a rigorous approach, the review adds vital knowledge to the profession and promotes evidence-based decision-making.

Review of Document: In the reporting phase of a systematic literature review, gather and organize the findings into a comprehensive review report. This step involves clearly presenting the extracted data, analysis, and conclusions. Once the report is assembled, it must be carefully validated to ensure accuracy, consistency, and clarity. Validation helps identify any errors, inconsistencies, or gaps, ensuring that the final report is reliable and well-structured. A thoroughly reviewed and validated report strengthens the credibility and impact of the research findings.

4. Proposed Methodology

4.1 Dataset Description

Three benchmark datasets from the NASA collection, PC1, CM1, and KC2, are made publicly available for the research's conclusions. The corresponding Table 2 provides a detailed description of each dataset, including the total number of features, the defective and non-defective samples, total samples and the percentage of defective samples. Table 3 shows the list of attributes within the dataset, accompanied by short descriptions of each feature and the respective data types stored in each.

Table 2: Details of the Nasa Dataset

Projects	Total Features	Defective Samples	Non-Defective Samples	Total Samples	Defective Samples %	Language Used
PC1	22	77	1032	1109	6.94%	NA
CM1	22	49	449	498	9.83%	C
KC2	22	107	415	522	20.49%	C++

Table 3: Dataset Metrics or Attributes

Metrics	**Short Description**	**Type**
Loc	McCabe's line count of code	Numeric
v(g)	McCabe's Cyclomatic Complexity	Numeric
ev(g)	McCabe's Essential Complexity	Numeric
iv(g)	McCabe's Design Complexity	Numeric
N	Halstead total operator and operands	Numeric
V	Halstead Volume	Numeric
L	Halstead Program length	Numeric
D	Halstead Difficulty	Numeric
I	Halstead Intelligence	Numeric
E	Halstead Effort	Numeric
B	Halstead Number of reported defects	Numeric
T	Halstead Time to write Program	Numeric
Locoed	Halstead's line count	Numeric
loComment	Halstead's count of lines of comments	Numeric
loBlank	Halstead's count of blank lines	Numeric
loCodeAndComment	Halstead's count of comments and code lines	Numeric
uniqOp	Halstead's total number of unique operators	Numeric
uniqOpnd	Halstead's total number of unique operands	Numeric
totalOp	Halstead's total number of operators	Numeric
totalOpnd	Halstead's total number of operands	Numeric
uniq_Opnd	Shows to unique operand	Numeric
Defects	Module has/ has not one or more defects	Boolean

4.2 Data Preprocessing

In this study, the input dataset is NASA-owned by the promise repository for bug prediction [34]. The class labels and software metrics of the software modules in the real-world dataset are identical. These recurrent events have a negative impact on machine learning. They also prolong the training period and reduce the model's performance. Pre-processing is used to scale the data, eliminate noise, and remove duplicate rows [25, 26]. Data preprocessing is important in machine learning because it converts raw data into an organized, clean format for analysis [27]. In order to ensure that the dataset is accurate and prepared for analysis, effective preprocessing takes care of missing values, noise, and inconsistencies [28]. Proper data preparation improves model performance by enhancing data quality, while insufficient preprocessing can result in unreliable models and erroneous predictions [30]. The instances create new instances to train the bug prediction model after pre-processing. The outcome is given by the prediction model in terms of clean and buggy instances [31].

SMOTE (The Synthetic Minority Over-sampling Technique) is used to resample the data to address the imbalance problem with the datasets. The SMOTE approach is used to prevent the loss of important data in the majority class [32]. SMOTE balances the dataset by adding new samples to the minority class through artificially generated data, ensuring that the total number of samples in both classes is equal [36].

First of all, we load our dataset from the NASA repository and drop duplicate records using drop_duplicates(). After that, it may have any missing values; we fill them using the mean() function. We apply IsolationForest(contamination=0.02, random_state=42) to remove outliers. MinMaxScaler is used as a feature scaling standardization approach. The balanced performance of machine learning (ML) approaches requires this data standardization strategy. Features are changed by scaling them into a range (0, 1).

The next stage is to split our dataset using an 80/20 ratio into training and testing sets. This makes it possible to train our model with one portion (80) and verify its performance with another portion (20) to ensure unbiased findings. Fig. 5 illustrates the step-by-step methodology of our experimental work. It shows the complete process from dataset selection through pre-processing, model training, and performance evaluation.

4.3 Effective Software Metrics Evaluation and Correlations Analysis

Find the most effective software metrics for bug prediction on different dataset. Variable correlation analysis is a statistical tool for determining how strongly and in which direction two or more variables are associated with defects.

Fig. 6 and Fig. 7 show the most effective 15 software metrics (out of 21 metrics) on PC1 and CM1 Dataset with Gradient Boosting Classifier respectively. Fig. 8 shows the most effective 15 software metrics (out of 21 metrics) on KC2 Dataset with XGBoost Classifier.

Fig. 9 shows the topmost 15 metric correlation Analysis on PC1 and CM1 Dataset with Gradient Boosting Classifier. Fig.10 shows the topmost 15 metric correlation Analysis on KC2 Dataset with XGBoost Classifier.

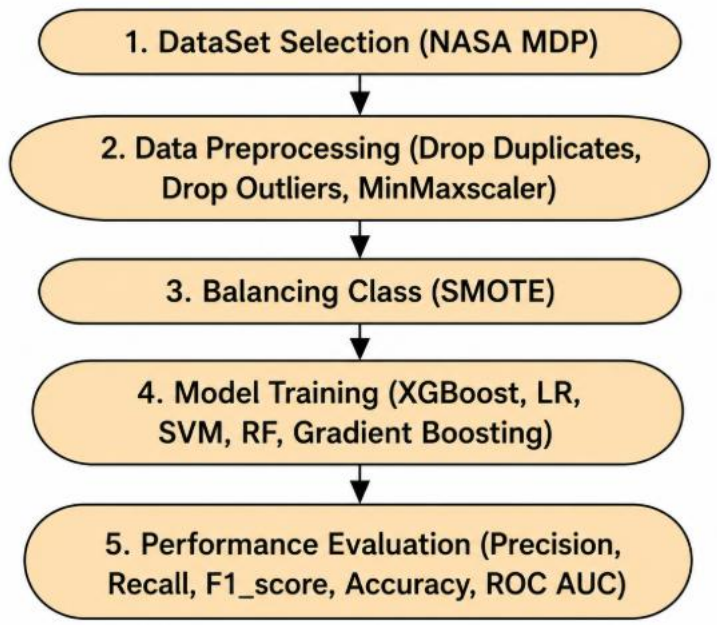


Fig. 5: Experimental Analysis Workflow

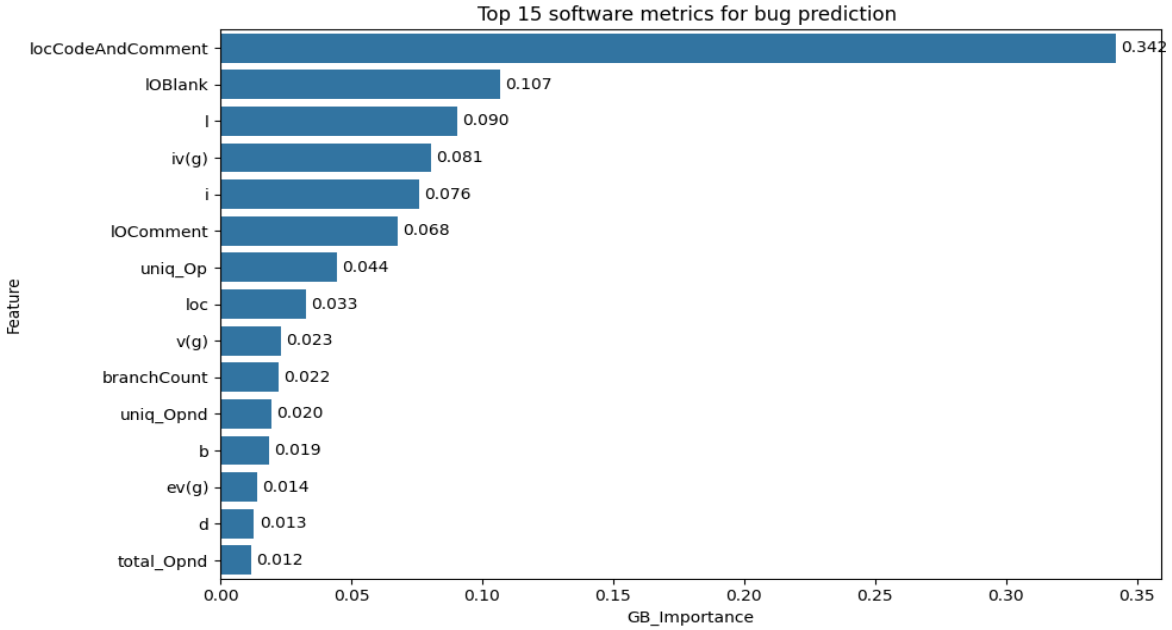


Fig. 6: Most Effective Software Metrics on PC1 Dataset with Gradient Boosting Classifier

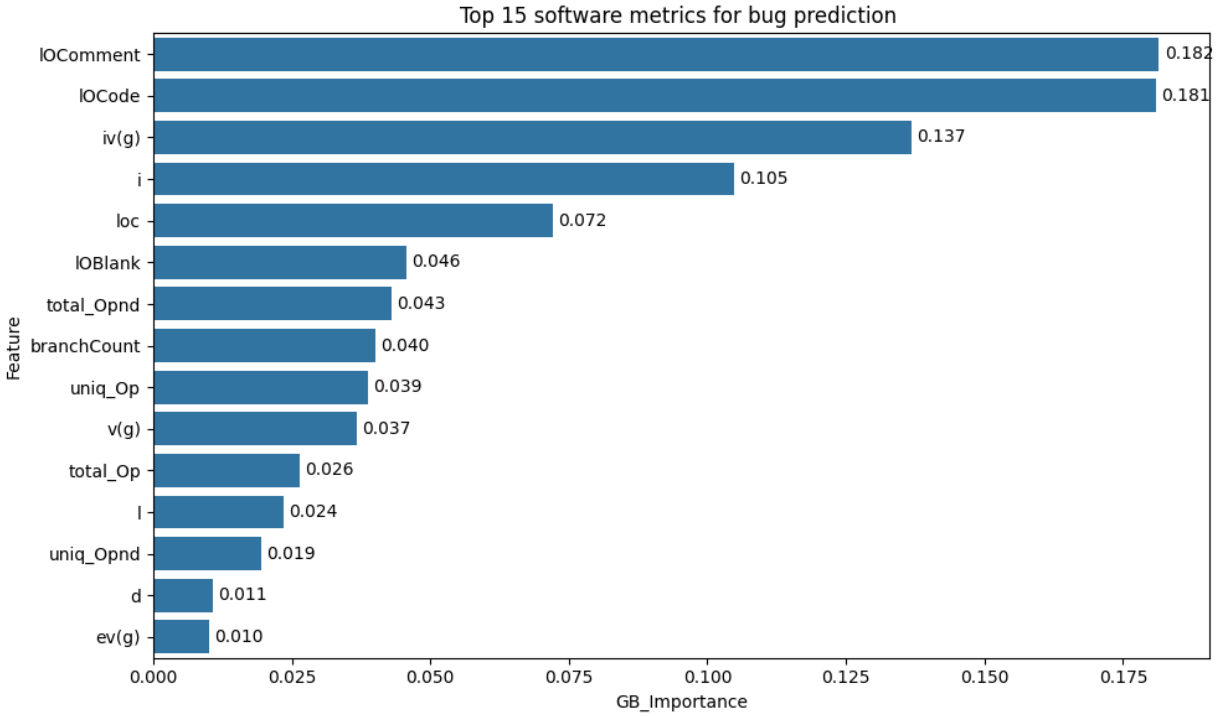


Fig. 7: Most Effective Software Metrics on CM1 Dataset with Gradient Boosting Classifier

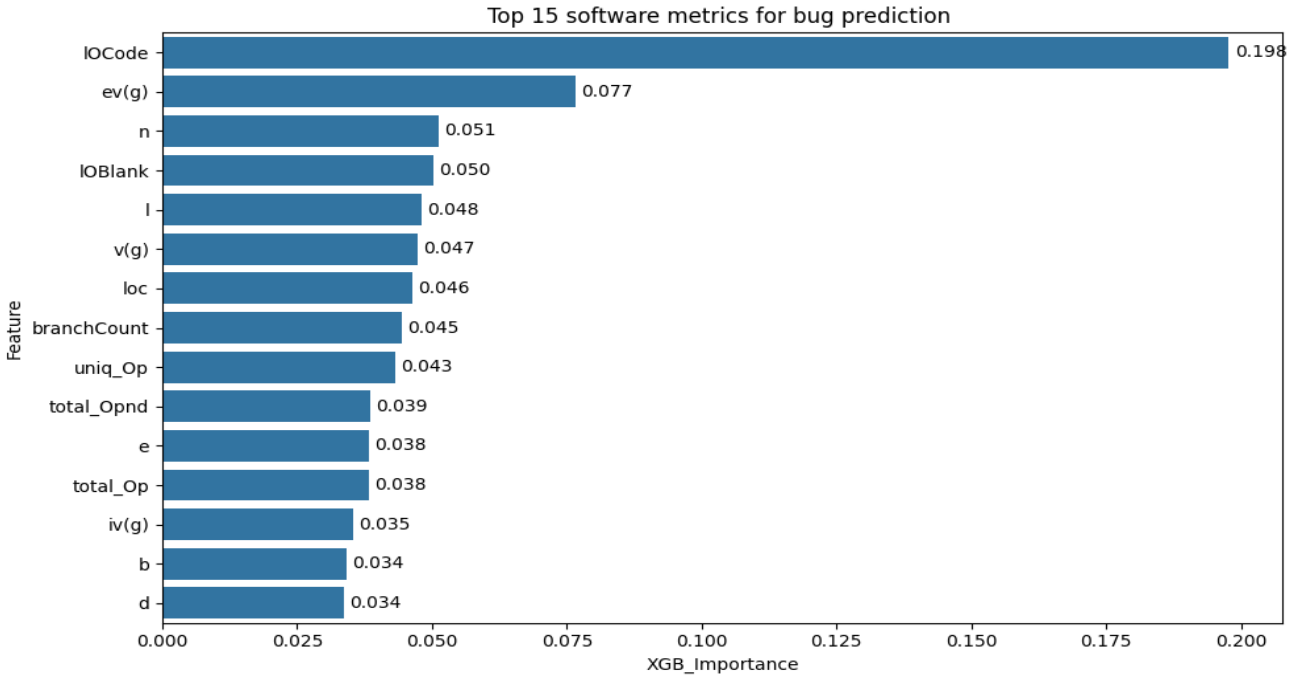


Fig. 8: Most Effective Software Metrics on KC2 Dataset with XGBoost Classifier

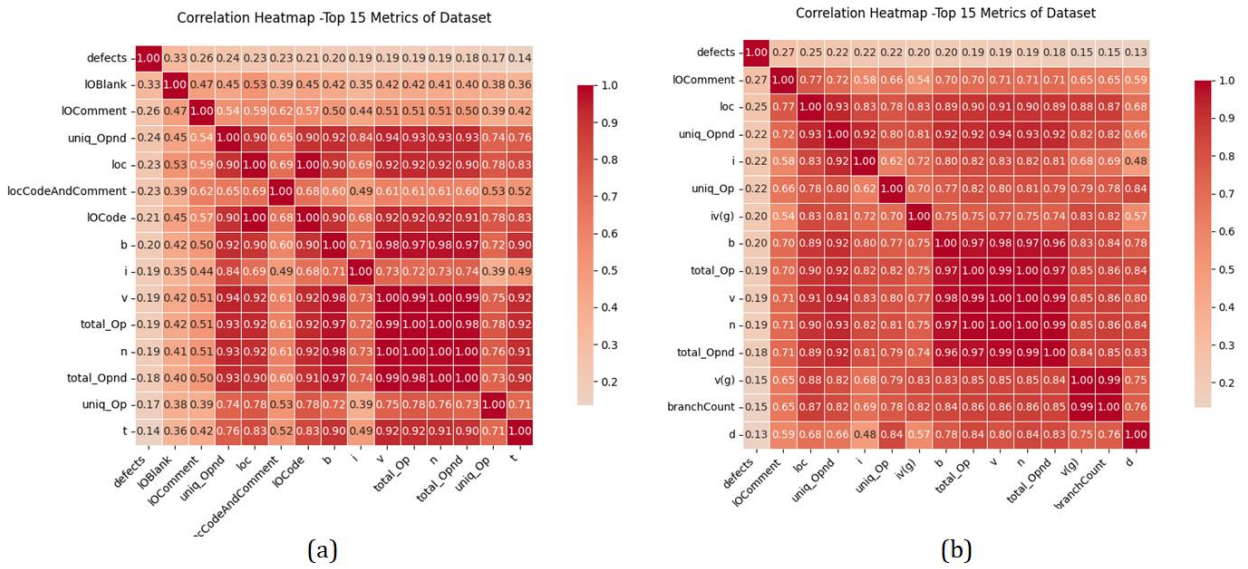


Fig. 9: Correlation Analysis on PC1 and CM1 Dataset with Gradient Boosting Classifier

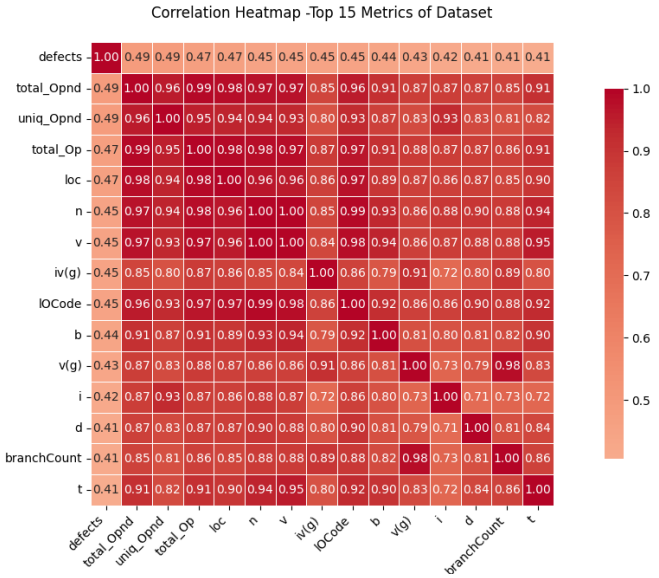


Fig. 10: Correlation Analysis on KC2 Dataset with XGBoost Classifier

4.4 Machine Learning Model

In this paper, we analyze performance indicators for five machine learning models. Firstly, we implement XGBClassifier(n_estimators=200, max_depth=5, learning_rate=0.1, subsample=0.8, random_state=42, use_label_encoder=False,eval_metric='logloss') optimizations like parallelization, regularization, and handling missing data are all part of an extremely effective gradient boosting implementation. Here, 200 boosting trees are to be constructed, 5 used as the depth of the tree individually. Each iteration's step size is determined by the learning rate, and in more recent versions, the default label encoding behavior is disabled to prevent problems. LogisticRegression (max_iter=1000, class_weight='balanced', random_state=42) with a maximum number of iterations of 1000.

SVC(kernel='rbf', probability=True, class_weight='balanced', random_state=42) is a distance-based model. An ensemble learning approach called RandomForestClassifier (n_estimators=200, random_state=42, class_weight='balanced') mixes many decision trees to increase prediction accuracy and decrease overfitting. In this case, the number of boosting trees (rounds) to build is indicated by n_estimators=200.

GradientBoostingClassifier(n_estimators=300, max_depth=3, learning_rate=0.1, subsample=0.8, random_state=42) it is an ensemble learning approach that creates models in order, with each new model fixing the mistakes of its predecessors and so on. In order to promote generalization and prevent overfitting, it is helpful

for managing complex data distributions. Here, n_estimators=300 denotes the number of boosting trees (rounds) to construct, and max_depth=3 indicates the depth of each individual DT.

5. Results and Experimental Analysis

RQ1. Which are the most effective software metrics with different ML techniques for predicting bugs?
Five machine learning models—XGBoost, Logistic Regression, Support Vector Machine, Random Forest, and Gradient Boosting—have been thoroughly developed and evaluated using PC1, CM1, and KC2 datasets. Out of 21 metrics of these datasets, the top 15 metrics—locodeandcomment, loblank, l, iv(g), i, locomment, uniq_op, loc, v(g), branchcount, uniq_opnd, b, ev(g), d, and total_Opnd—are the most effective, consisting of the highest accuracy for bug prediction. The feature selection selects important subsets of top 15 metrics among 21 metrics using different ML techniques.

RQ2. What is each ML classifier's ability to predict bugs in different datasets?
The Gradient Boosting feature selection on the PC1 dataset achieved the highest accuracy of 89.8%. The next best Gradient Boosting feature selection on the CM1 dataset achieved the accuracy of 85%. The XGBoost feature selection on the KC2 dataset achieved the accuracy of 77%.

Table 4: Performance Metrics on ML Models with PC1 Dataset

Classifier	Features	Precision	Recall	F1_Score	Accuracy	ROC AUC
XGBoost	21	0.300	0.461	0.363	0.887	0.861
	15	0.250	0.307	0.275	0.887	0.816
RF	21	0.204	0.769	0.322	0.775	0.825
	15	0.204	0.769	0.322	0.775	0.815
SVM	21	0.218	0.923	0.352	0.764	0.832
	15	0.192	0.769	0.307	0.759	0.830
LR	21	0.195	0.692	0.305	0.780	0.823
	15	0.191	0.692	0.300	0.775	0.824
Gradient Boosting	21	0.294	0.384	0.333	0.893	0.833
	15	0.333	0.461	0.387	**0.898**	0.849

5.1 Model Evaluation

The trained model was evaluated based on Precision, Recall, F1_Score, Accuracy, and ROC AUC. The performance comparison between ML Techniques such as XGBoost, LR, SVM, and RF is shown in Tables 4, 5, and 6 for the PC1, CM1, and KC2 datasets.

In Table 4, the Gradient Boosting Classifier on the PC1 dataset achieved the highest accuracy of 89.8%. XGBoost classifier at 88.7% is the second best, providing a good alternative with the PC1 dataset.

In Table 5, the Gradient Boosting Classifier on the CM1 dataset achieved the highest accuracy of 85%. XGBoost at 83.9% is the second best, providing a good alternative with the CM1 dataset.

In Table 6, the XGBoost Classifier on the KC2 dataset achieved the highest accuracy of 77%.

Table 5: Performance Metrics on ML Models with CM1 Dataset

Classifier	Features	Precision	Recall	F1_Score	Accuracy	ROC AUC
XGBoost	21	0.200	0.222	0.210	0.827	0.735
	15	0.222	0.222	0.222	0.839	0.719
RF	21	0.206	0.666	0.315	0.701	0.743
	15	0.241	0.777	0.368	0.724	0.747
SVM	21	0.291	0.777	0.424	0.781	0.867
	15	0.280	0.777	0.411	0.770	0.868
LR	21	0.280	0.777	0.411	0.770	0.851
	15	0.280	0.777	0.411	0.770	0.853
Gradient Boosting	21	0.222	0.222	0.222	0.839	0.716
	15	0.250	0.222	0.235	**0.850**	0.710

Table 6: Performance Metrics on ML Models with KC2 Dataset

Classifier	Features	Precision	Recall	F1_Score	Accuracy	ROC AUC
	21	0.529	0.450	0.486	0.743	0.736

XGBoost	15	0.578	0.450	0.564	**0.770**	0.745
RF	21	0.500	0.600	0.545	0.729	0.759
	15	0.480	0.600	0.533	0.716	0.752
SVM	21	0.482	0.700	0.571	0.716	0.774
	15	0.500	0.700	0.583	0.729	0.783
LR	21	0.466	0.700	0.560	0.702	0.764
	15	0.406	0.700	0.560	0.702	0.765
Gradient Boosting	21	0.470	0.400	0.432	0.716	0.692
	15	0.476	0.500	0.487	0.716	0.695

5.2 Performance Metrics

TPs are positive samples that the classifier correctly predicts as such. Positive samples that were mistakenly categorized as positive are known as FNs. Similarly, TNs are accurately predicted negative samples, while FPs are negative samples that are mistakenly classified as positive. Equation (4) defines a classifier's accuracy.

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \tag{4}$$

The area under the curve between the True Positive Rate (TPR) and False Positive Rate (FPR) for various feature selection comparisons is known as AUC-ROC [41].

$$\text{TPR (Recall)} = \frac{TP}{TP+FN} \tag{5}$$

$$\text{FPR} = \frac{FP}{FP+TN} \tag{6}$$

Fig. 11 shows the ROC Curve with Gradient Boosting Feature Selection Comparison with PC1 dataset. Fig. 12 shows the ROC Curve with Gradient Boosting Feature Selection Comparison with CM1 dataset. Fig. 13 shows the ROC Curve with XGBoost Feature Selection Comparison with KC2 dataset.

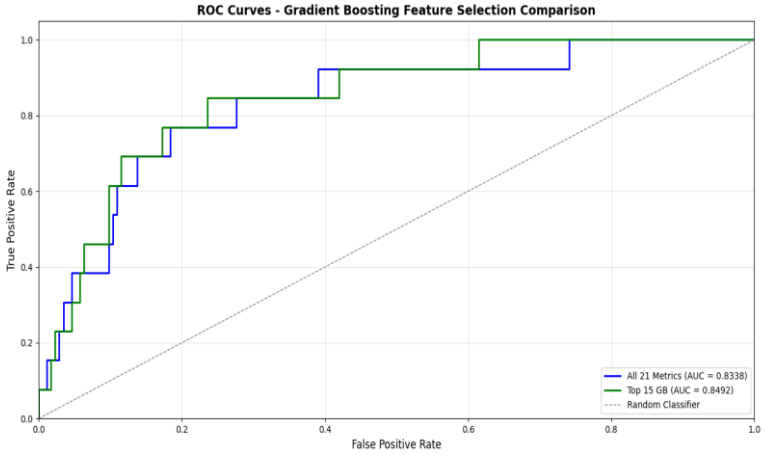


Fig. 11: ROC Curve of PC1Dataset

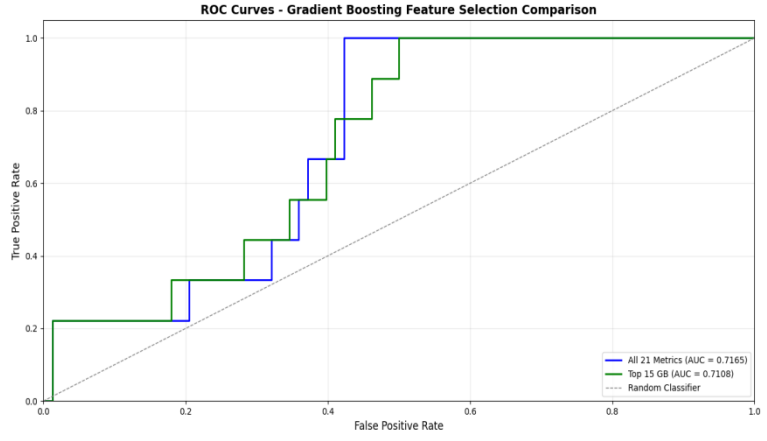


Fig. 12: ROC Curve of CM1 Dataset

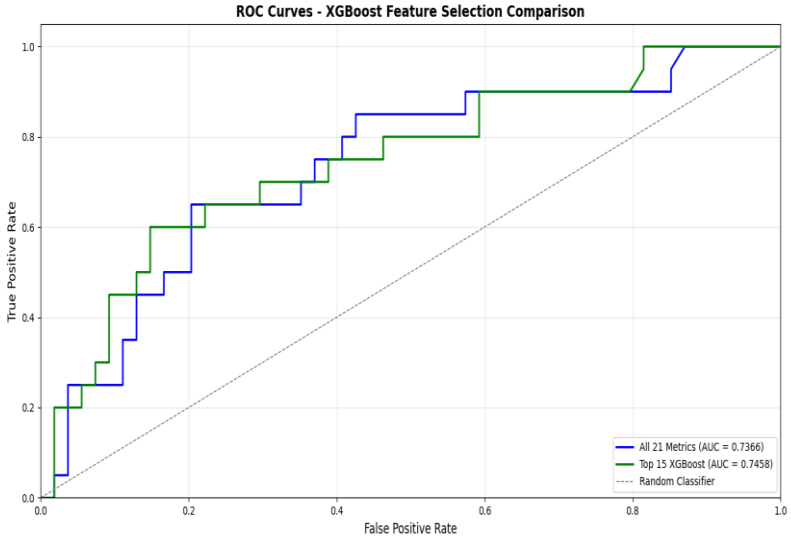


Fig. 13: ROC Curve of KC2 Dataset with XGBoost

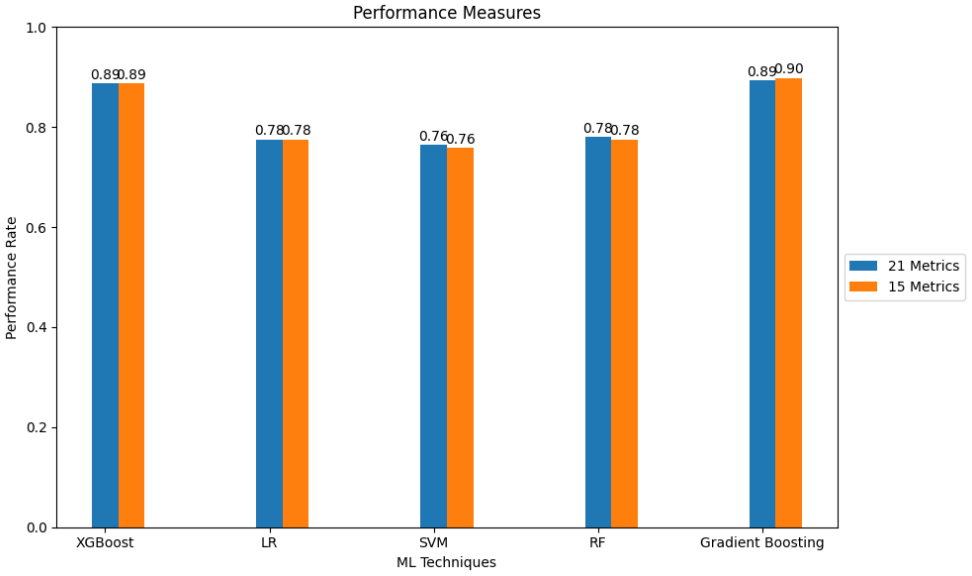


Fig. 14: Comparative Analysis of ML Models with Top 15 Effective Metrics on PC1

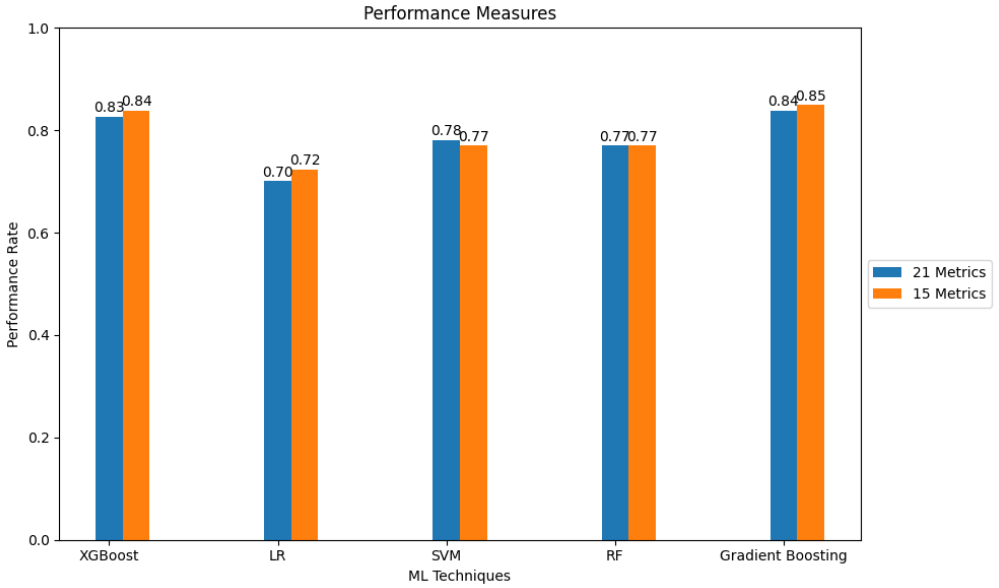


Fig. 15: Comparative Analysis of ML Models with Top 15 Effective Metrics on CM1

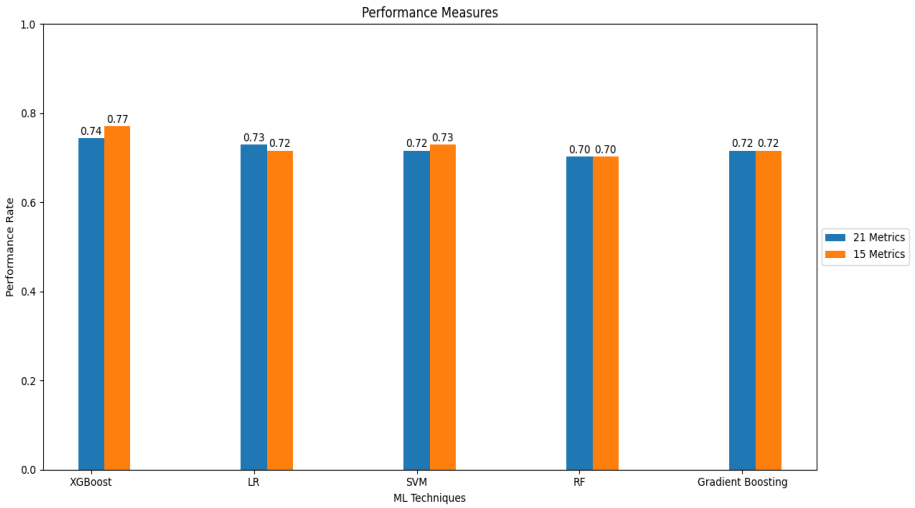


Fig. 16: Comparative Analysis of ML Models with Top 15 Effective Metrics on KC2

Figs. 14, 15, and 16 show the **c**omparative analysis of ML models with Feature Selection on PC1, CM1, and KC2 datasets. Out of 21 metrics of these datasets, the top 15 metrics—locodeandcomment, loblank, l, iv(g), i, locomment, uniq_op, loc, v(g), branchcount, uniq_opnd, b, ev(g), d, and total_Opnd—are the most effective, consisting of the highest accuracy for bug prediction as shown in Fig. 14.

6. Conclusions and Future Work

This paper analyzes the most effective software metrics for bug prediction. The most popular dataset for predicting bug research is the PROMISE repository's NASA datasets, which offer a variety of parameters for analysis. We evaluated Machine Learning models, XGBoost, Logistic Regression, Support Vector Machine, Random Forest and Gradient Boosting on three Software Bug Prediction datasets: PC1, CM1 and KC2. In this study, we find the most effective software metric values for bug prediction on these three different datasets PC1, CM1, and KC2. Out of 21 metrics of these datasets, the top 15 metrics—locodeandcomment, loblank, l, iv(g), i, locomment, uniq_op, loc, v(g), branchcount, uniq_opnd, b, ev(g), d, and total_Opnd—are the most effective, consisting of the highest accuracy for bug prediction. The Gradient Boosting feature selection on the PC1 dataset achieved the highest accuracy of 89.8%. Our findings show that, as compared to employing all metrics, various feature ranking strategies for dimensionality reduction enhance the predictive model's performance.

However, these same metrics can also be applied to other established projects, offering opportunities for further research. In order to minimize computation without significantly reducing performance, it is recommended to use feature selection strategies in various datasets.

The approach could be expanded in the future to include more advanced correlation metrics, such as mutual information and distance correlation, as well as feature interaction analysis. Additionally, extending the benchmarking process to incorporate contemporary feature selection and reduction techniques and experimenting with deep learning systems could enhance generalizability. Furthermore, testing various learning rate schedules and assessment metrics may help validate the method's resilience and effectiveness across different contexts.

References

[1] Shailee NM, Alam A, Ahmed T, Mamun Rudro RA, Nur K (2024) Software Bug Prediction using Machine Learning on JM1 Dataset. In: 2024 International Conference on Advances in Computing, Communication, Electrical, and Smart Systems (iCACCESS). Dhaka, Bangladesh, pp 1-6, doi: 10.1109/iCACCESS61735.2024.10499572

[2] Siva R, Hariharan B, Premkumar N (2023) Automatic Software Bug Prediction Using Adaptive Artificial Jelly Optimization With Long Short-Term Memory. Wireless Pers Commun 132:1975–1998

[3] Murakami Y, Yamasaki Y, Tsunoda M, Monden A, Tahir A, Bennin KE, Nakasai K (2024) The Effect of Defect (Re) Prediction on Software Testing. arXiv preprint arXiv:2404.11040

[4] Rani S, Kaur A (2020) Efficient framework for fully automatic test case generation and prioritization using genetic algorithm in software testing. J Comput Theor Nanosci 17(11):5198–5204.

[5] Fan G, Diao X, Yu H, Yang K, Chen L (2019) Deep semantic feature learning with embedded static metrics for software defect prediction. In: 2019 26th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp 244-251
[6] Han D, Hoh IP, Kim S, Lee T, Nam J (2011) Micro interaction metrics for defect prediction. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering
[7] Rani S, Kaur A (2021) Automatic test case generation and fault-tolerant framework based on N-version and recovery block mechanism. In: Cyber Security and Digital Forensics: Proceedings of ICCSDF 2021. Springer, Singapore, pp 65-74
[8] Lee T, Nam J, Han D, Kim S, In HP (2016) Developer micro interaction metrics for software defect prediction. IEEE Trans Softw Eng 42:1015–1035.
[9] Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA (2021) Within-project defect prediction of infrastructure-as-code using product and process metrics. IEEE Trans Softw Eng 48:2086–2104.
[10] Al-Fraihat, D., Sharrab, Y., Al-Ghuwairi, A. R., Alshishani, H., & Algarni, A. (2024). Hyperparameter optimization for software bug prediction using ensemble learning. IEEE Access, 12, 51869-51878.
[11] Cheng X, Nie X, Li N, Wang H, Zheng Z, Sui Y (2022) How about bug-triggering paths?—understanding and characterizing learning-based vulnerability detectors. IEEE Trans Depend Secure Comput.
[12] Ali S, Wahid F, Baseer S, Alkhayyat A, Al-Radaei A (2023) Smell-aware bug classification. IEEE Access.
[13] Sahar, S., Younas, M., Khan, M. M., & Sarwar, M. U. (2024). DP-CCL: A supervised contrastive learning approach using CodeBERT model in software defect prediction. IEEE Access, 12, 22582-22594.
[14] Ali, M., Mazhar, T., Arif, Y., Al-Otaibi, S., Ghadi, Y. Y., Shahzad, T., ... & Hamam, H. (2024). Software defect prediction using an intelligent ensemble-based model. IEEE Access, 12, 20376-20395.
[15] Al-Fraihat, D., Sharrab, Y., Al-Ghuwairi, A. R., Alshishani, H., & Algarni, A. (2024). Hyperparameter optimization for software bug prediction using ensemble learning. IEEE Access, 12, 51869-51878.
[16] Punitha K, Chitra S (2013) Software defect prediction using software metrics—a survey. Proc Int Conf Inf Commun Embedded Syst (ICICES), pp 555–558. IEEE, February 2013
[17] Nirpal PB, Kale KV (2011) A brief overview of software testing metrics. Int J Comput Sci Eng 3(1):204–211.
[18] Singh Y, Kaur A, Suri B (2008) An empirical study of product metrics in software testing. In: Innovative techniques in instruction technology, e-learning, e-assessment, and education. Springer, pp 64–72.
[19] Cavalcanti YC, de Almeida ES, de Lemos Meira S (2010) A bug report analysis and search tool and its validation. In: Anais do IX Simpósio Brasileiro de Qualidade de Software. Sociedade Brasileira de Computação (SBC), pp 455–472, June 2010
[20] Karim S, Warnars HLHS, Gaol FL, Abdurachman E, Soewito B (2017) Software metrics for fault prediction using machine learning approaches: a literature review with PROMISE repository dataset. In: Proc IEEE Int Conf Cybern Comput Intell (CyberneticsCom). IEEE, November 2017, pp 19–23.
[21] Farooq SU, Quadri SMK, Ahmad N (2011) Software measurements and metrics: role in effective software testing. Int J Eng Sci Technol 3(1):671–680.
[22] Pradhan D, et al. (2025) Enhancing software bug prediction with extreme learning machine: an empirical analysis using NASA datasets. In: Proc 1st Int Conf Adv Comput Sci Electr Electron Commun Technol (CE2CT), Bhimtal, Nainital, India, pp 72–77.
[23] Debbarma MK, et al. (2013) A review and analysis of software complexity metrics in structural testing. Int J Comput Commun Eng 2(2):129–133.
[24] Magel, K., Kluczny, R. M., Harrison, W. A., & Dekock, A. R. (1982). Applying software complexity metrics to program maintenance.
[25] Nam, J., Pan, S. J., & Kim, S. (2013, May). Transfer defect learning. In 2013 35th international conference on software engineering (ICSE) (pp. 382-391). IEEE.
[26] Kim, S., Zhang, H., Wu, R., & Gong, L. (2011, May). Dealing with noise in defect prediction. In Proceedings of the 33rd International Conference on Software Engineering (pp. 481-490).
[27] Pradhan, D., & Muduli, D. (2023, July). Software defect prediction model using AdaBoost based random forest technique. In 2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT) (pp. 1-6). IEEE.
[28] Efendioglu, M., Sen, A., & Koroglu, Y. (2018). Bug prediction of systemc models using machine learning. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(3), 419-429.
[29] Li HF, Cheung WK (1987) An empirical study of software metrics. IEEE Trans Softw Eng 13(6):697–708. https://doi.org/10.1109/TSE.1987.233475
[30] Pradhan, D., Sekhar Dalai, S., & Behera, M. P. (2020). A comparative study on evolutionary model for software development. Int J Eng Res Technol, 8(1), 1-3.
[31] Rani, S., & Gupta, D. (2018). A comparative study of different software testing techniques: a review. J. Adv. Shell Program, 5(1), 1-8.

[32] Dudjak, M., & Martinović, G. (2020). In-depth performance analysis of SMOTE-based oversampling algorithms in binary classification. International journal of electrical and computer engineering systems, 11(1), 13-23.
[33] Lazić L, Mastorakis N (2008) Cost effective software test metrics. WSEAS Trans Comput 7(6):599–619.
[34] https://github.com/ApoorvaKrisna/NASA-promise-dataset-repository
[35] Alsulami M (2021) A systematic literature review on software metrics. Int Trans J Eng Manag Appl Sci Technol 12(12):12A12G:1–13. https://doi.org/10.14456/ITJEMAST.2021.238
[36] Kaope, C., & Pristyanto, Y. (2023). The effect of class imbalance handling on datasets toward classification algorithm performance. MATRIK: Jurnal Manajemen, Teknik Informatika dan Rekayasa Komputer, 22(2), 227-238
[37] Rani, S., & Gupta, D. (2019). A Faster Software Fault Prediction using White-Box Testing (LT) and Black-Box Testing (BVA) Techniques. International Journal of Computer Science & Communication (IJCSC), 10(1), 250-257.
[38] Liu K, Liu Y, Chen Z, Zhang JM, Han Y, Ma Y, Huang G (2024) LLM-Powered Test Case Generation for Detecting Tricky Bugs. arXiv preprint arXiv:2404.10304
[39] Šikić, L., Afrić, P., Kurdija, A. S., & Šilić, M. (2021). Improving software defect prediction by aggregated change metrics. IEEE Access, 9, 19391-19411.
[40] Saini, S., Bhagwan, J., Rani, S., Kumar, S., Godara, S., & Chaba, Y. (2024). Early Software Bug Prediction: A Literature Review and Current Trends. Grenze International Journal of Engineering & Technology (GIJET), 10.
[41] Sharma, T., Misra, S., & Colomo-Palacios, R. (2025). CorrBoost: a feature selection technique and utility of tabular deep neural networks in software fault prediction. Neural Computing and Applications, 37(32), 26845-26886.