IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p-ISSN: 2278-8727, Volume 28, Issue 5, Ser. I (Sep. – Oct. 2026), PP 32-46
www.iosrjournals.org

An Adaptive Boosting Ensemble Pipeline for Software Defect Prediction over Imbalanced Dataset

Stephen Bassi Joseph[1], Kunduli Mustapha[2], Abideen Adekunle Ismail[1]
[1](Department of Computer Engineering, Faculty of Engineering University of Maiduguri, Borno State Nigeria)
[2](Department of Electrical Engineering, Faculty of Engineering Technology, Applied Design and Fine Art, Kabale University, Kabale, Uganda)

Abstract:
Background*: Software Defect Prediction (SDP) is a proactive quality assurance strategy that uses static code complexity metrics to catch software bugs before a system is deployed. software metrics are highly imbalanced. Because clean code segments vastly outnumber defective ones, standard machine learning models tend to ignore the minority class, leading to a dangerous bias that leaves actual defects undetected. This study develops a reliable, end-to-end machine learning pipeline engineered to neutralize class imbalance, eliminate feature-scaling distortions, and establish a dependable balance between precision and recall.*
Materials and Methods*: The proposed workflow processes the NASA CM1 dataset by enforcing strict out-of-sample data binarization and Z-score normalization to prevent data leakage. We then deploy an adaptive ensemble framework that links ten deep decision tree base learners under the discrete SAMME AdaBoost algorithm.*
Results*: After implementing the proposed method on an independent 33% test data, the ensemble pipeline achieves an average overall accuracy of **99.40%**. This shows the veracities of the dataset's skewness, the model isolates defective modules with a localized precision of **0.98** and a recall of **0.98**, while maintaining a stable macro-averaged F_1-score of **0.97** and a weighted F_1-score of **0.99** across all code streams.*
Conclusion: *The overall results show that combining strict training-boundary and normalization with sequential error correction vectors prevent minority-class blindness in the model. Without using noisy artificial data oversampling, this method enables the ensemble engine to map complex, non-linear code trajectories and identify irregularities spontaneously.*
Key Word*: Software Defect; Machine Learning; Hyperparameter Tuning; Ensemble Learning.*

Date of Submission: 03-09-2026 Date of Acceptance: 13-09-2026

I. Introduction

Several software development organizations aim to anticipate bugs in order to preserve software quality for client satisfaction and reduce testing costs. Software defects prediction (SDP) is an important phase in the software development life cycle. Machine learning (ML) algorithms and past defects data are used for software defect prediction [1], [2]. Software testing becomes very difficult and challenging as the number of software applications keeps growing. Software evaluators, the task of identifying possible errors in millions of lines of code and large documentation is a nightmare [3]. However, improving software quality requires optimizing error detection. This becomes critical in high-stakes situations, where small or tiny undiscovered bugs might have serious consequences, [4], [5].

In modern software engineering, waiting to find critical faults during software product deployment is no longer a worthwhile option. Software Defect Prediction has emerged as a pillar proactive software quality control. Historical data repositories are employed for the prediction of faulty source code components before reaching an operational environment [6]. The fundamental principles of data-driven SDP depend on the extraction of static source code characteristics, particularly physical line counts, Halstead measures, and McCabe's cyclomatic complexity [7]. Software engineers can automate the process by converting these static signatures into organized numerical feature matrices. This transforms previously subjective code review sessions into pipelines for objective pattern identification.

The goal of SDP is to deliver dependable, high-quality software while optimizing the use of scarce resources. This will enable software developers to prioritize computer resource use at every stage of the software development process [1]. Many companies that produce different kinds of software want to anticipate software flaws in order to minimize testing costs and preserve software quality for customer satisfaction. SDP is used to improve the quality of the program, and testing may be done effectively by building different categorization

models using different machine learning techniques. To improve software quality and lower software testing expenses, a variety of machine learning techniques have been studied thus far to predict mistakes in software modules.

According to the authors in [8] the leading causes of software problems include: requirements (16.5%), architecture and design (25.0%), code (27.0%), security flaws (6.0%), documentation (10.5%), and inadequate solutions (15.0%). Additionally, the categories of "methods," "human resources," "technology," and "documentation" are the main causes of these problems [9]. Syntax problems, runtime issues, and design flaws like code smells are some of the several categories of software faults. Bad design decisions that adversely affect code quality, maintainability, changeability, and comprehensibility are known as "code smells." These decisions make systems difficult to modify, which may lead to developers introducing bugs.

Software defects are categorized using symptom descriptions such as runtime problems, link-time problems, and compile-time problems; design errors are further divided into data-structure problems, algorithm problems, and issues with user-interface, software-interface, and hardware-interface specifications. Initialization errors, binding errors, memory issues, control-flow issues, and typographical errors are examples of coding errors [10]. Cognitive psychology examined information-processing models of human intelligence to explain observed human faults in the design and operation of complex automated systems. Human error is a significant contributor to software problems. Technical debt, which is the result of incomplete or delayed software project tasks that affect future maintenance effort, can also lead to defects. Defects may accumulate and communication over their resolution may increase as a result of inconsistent or incomplete feature definitions and delayed testing detection. Inadequate specifications are a factor because the likelihood of defects is increased by more requirements and limitations, and the risk of defects is further increased by incomplete, inconsistent, or frequently changing specifications. The majority of errors in operational software are attributed to poor specifications rather than implementation faults, and the tendency for feature creep and late additions frequently results in needless complexity and debugging challenges. Figure 1 illustrates causes of software defects.

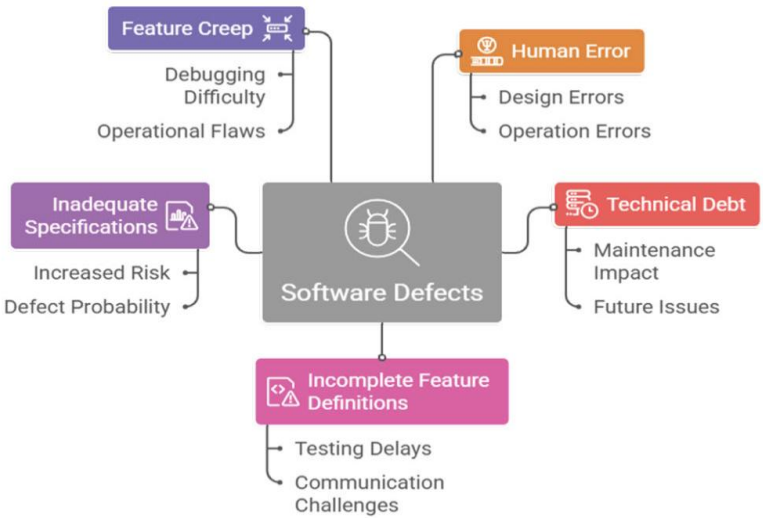

Fig. 1: Causes of Software Defects

At present, machine learning techniques are highly sought after in a number of industries, including software development, chemical engineering, healthcare, education, road infrastructure, agriculture, and renewable energy. Specifically, machine learning-based software defect prediction is crucial since these models use past data to predict whether or not future software instances will be problematic. This predictive ability greatly enhances the quality of software and maximizes resources while it is being developed and maintained [3].

Many researchers have developed defect prediction models based on code metrics over the last 20 years by using machine learning technology. Ensemble approaches and other ML techniques with feature selection methods like PCA are recent developments in machine learning [1]. Among the most widely used and well researched techniques in the most recent literature on software defect prediction are machine learning (ML) algorithms, such as Random Forest (RF), Bayesian methods, and Logistic Regression (LR). These methods have demonstrated good performance in SDP, offering significant benefits in improving software quality [11].

Motivation:

The main inspiration behind automated software defect predictions systems is the expensive nature of resolving bugs in a software after development. Identifying a major architectural or functional defect during the testing phase of software development slows down the development process and can also be very expensive. Also, early identification of systemic weak links within a codebase can aid project managers can prioritize limited quality assurance assets. This focused diagnostic approach directs costly manual code reviews and intricate unit testing blocks at high-risk modules. Subsequently creating much more stable software releases and reduces overall long-term maintenance cost by concentrating the resources.

Major software developing companies are putting a lot of effort to producing software modules that are bug free. Software quality is lowered by errors, which prevent programs from doing tasks precisely and efficiently. Software defect prediction is still the most imperative phase of SDLC. It is the most significant phase of software testing [12]. There is more to defect management than just improving software quality. Software defects management inspires developers to put quality first throughout during software development. This steady dedication inevitably results in enhancing subsequent software releases. Software bugs must be identified and corrected to improve software's dependability and usefulness.

State-of-the-Art:

Modern defect prediction has moved beyond simple linear regressions. Instead, it relies on complex machine learning frameworks. This shift is necessary to keep up with the exponential growth of modern coding environments [13]. Conventional supervised learning classifiers such as Support Vector Machines (SVMs), Naive Bayes, and k-Nearest Neighbors (k-NN) these algorithms today serve as standards for benchmarking. Software developers utilize them to evaluate other proposed systems [1]. Recently, state-of-the-art architectures depends heavily on multi-estimator ensemble configurations. By employing frameworks such as Random Forests, stacked meta-learners, and advanced Gradient Boosting engines [14]. Combining multiple base estimators eliminates localized prediction variance. This approach maps complex, non-linear classification boundaries far better than single learning algorithms.

Gaps in Literature:

Despite the rapid evolution in software defect prediction approaches, there are still two persistent and critical issues yet to be resolved. These issues include:

1. **Blind Spot of Class Imbalance in Datasets:** Real world software systems are naturally different; clean, fully operational packages mostly more than broken or vulnerable ones [15]. Typical machine learning algorithms are designed to maximize global accuracy scores. As a result, during operation, these algorithms frequently ignore underrepresented data segments. This phenomenon results in models that appear statistically flawless on paper but are completely blind to uncommon occurrences. The system becomes heavily biased in favor of the ruling class since they are unable to identify actual problems.

2. **Feature Scaling Distortion and Data Leakage:** Software metrics are not uniformly distributed. Rather, they show significant levels of multi-collinearity and scale unevenly. In order to address this problem, several existing systems apply generalized global feature normalization strategies. Nevertheless, these systems commonly compute scaling parameters across the complete dataset without keeping folds independent. This process produces data leakages, artificially expanding validation benchmarks. As a result, making models to perform poorly when subjected to entirely new datasets.

Contribution:

This study proposes an end-to-end machine learning pipeline strategy to address identified methodological gaps. The system is designed to precisely separate and classify minority software defects with high level of reliability. The main specific contributions of this work include:

- Development of a preprocessing pipeline that prevents data leakages and transforms target anomalies into clear binary format and standardizes highly uneven metric parameters only allocated training datasets.
- Design and develop a multi estimator Adaptive Boosting (AdaBoost) meta classifier driven by deep decision tree base learners and uses discrete Stagewise Additive Modeling using a Multiclass Exponential loss function (SAMME) algorithm.
- Presentation an evaluation strategy that replaces misleading measurement strategies.

The challenges of severe data imbalance and feature scale distortion demand a paradigm shift from conventional, machine learning architectures for software defect prediction. To resolve these limitations, this paper introduces a highly structured, leak-free processing pipeline that pairs a cost-sensitive AdaBoost classifier with high-capacity decision tree base learners optimized via the discrete SAMME algorithm [16]. The core significance of this methodology lies in its ability to handle stark class skewness entirely within the learning logic rather than relying on noisy, artificial data oversampling.

This paper is further organized as follows. The relevant related works is discussed in Section II. The materials and procedures utilized in the experiments are described in Section III. The model evaluation diagnostic reporting are presented in Section IV. Section V finally concludes the study and presents actionable future works.

II. Related Works

This section outlines the state-of-the-art methodologies in software defect prediction (SDP). It highlights three key themes in recent literature: baseline static metric extraction, standard supervised machine learning adaptations, and ensemble learning frameworks designed to mitigate data distribution anomalies.

A. **Static Code Complexity Metrics and Evaluation Foundational Baselines:** Early researches in software engineering proved that structural code metrics relates strongly with software quality and defect occurrence. Conventional systems historically depend on Halstead metrics such as volume, abstraction, and developmental effort and McCabe cyclomatic complexity like control flow mechanics to convert source code files into measurable software data arrays [6]. Whereas typical experimental data collection employs foundational open-source datasets repositories, particularly the NASA Metric Data Program (MDP) benchmark datasets like the CM1, PC1, JM1, and KC1 [13]. Despite the fact these static metrics are universally accepted as reference descriptive features, current systematic researches prove otherwise. Raw static metrics show high collinearity and skewed distributions, which requires complex mathematical transformations before being used for pattern recognition [13].

B. **Machine Learning Paradigms in Software Quality Analysis**: Recently supervised machine learning classifiers has replaced the use of conventional statical regression models for SDP. Baseline evaluations in convention SDP projects rely on traditional single estimator models such as Naive Bayes (NB), Logistic Regression (LR), Decision Trees (DT), Support Vector Machines (SVM), and k-Nearest Neighbors (k-NN) [17]. The application of Decision Trees for SDP offers high structural interpretability, while on the other hand Support Vector Machines isolate geometric boundaries on unbalanced small dataset clusters [17]. Nevertheless, these conventional classifiers suffer significant performance drops when using highly non-linear, multi-dimensional complex large software defects datasets [18].

C. **Ensemble and Adaptive Boosting Classifiers for Class Imbalance**: To solve the high prediction variance and overcome the shortcomings of single prediction learners, modern prediction models rely on ensemble classification schemes. Advanced pipelines models build combined frameworks like Random Forests (RF), stacked meta-learners, and advanced Gradient Boosting engines for prediction [19]. While typical machine models improve general accuracy of single models, they are plagued with high blind spots due to dataset imbalance [14]. Several modern proposed frameworks mitigate this problem by integrating synthetic oversampling like SMOTE or deep neural network connectors. However, they produce artificial noise or increase the complexity of training computation [17]. On the contrary, cost-effective adaptive boosting (AdaBoost) meta-classifiers offers an alternative approach. The AdaBoost meta-classifiers employ stage-wise discrete algorithms particularly multi-class SAMME configuration to improve performance [15]. The system dynamically adjusts sample weights without changing the raw metric space of the dataset. Hence, enhancing minority class identification at the same time preserving feature scaling boundaries [14].

III. Materials and Methods

Proposed Solution to Identified Gaps in Literature

To solve these identified established limitations of software prediction methods without complicating the training process or changing the raw code metrics. This work proposes a stage-gated architectural structured methodology

using AdaBoost ensemble pipeline. Furthermore, the proposed system to eliminates data leakages and artificial high-performance score by applying strict boundaries during feature scaling. The system separates and calculates variance and mean values completely during the training fold ($\mathcal{D}_{\text{train}}$) before transforming the testing data ($\mathcal{D}_{\text{test}}$). This is to ensure that validation metrics shows genuine out-of-sample generalization.

Instead of than changing the dataset's normal distribution with noisy synthetic samples, The proposed system resolves class imbalance directly within the learning algorithm. It uses an adaptive boosting engine that updates instance weights step-by-step:

$$\boldsymbol{w_i^{(m+1)} \propto w_i^{(m)} \cdot e^{\alpha_m \cdot \mathbb{I}(y_i \neq h_m(x_i))}} \quad \textbf{(1)}$$

The proposed system uses an exponential penalty to misclassified samples. The system prevents rare, defective instances from being washed out by large volumes of non-defective instances in the dataset. The sequential process forces successive weak learners to capture complex, non-linear fault boundaries naturally. This training strategy creates a balanced model that conserves minority-class sensitivity without altering the software datasets.

Where:

- Index counter tracking individual software modules.
- Index counter tracking individual static code complexity metrics.
- Total sample count present inside the active partition.
- Ground-truth class variable (1 for defective, 0 for non-defective).
- Raw baseline metric value observed for module i on feature j.
- Arithmetic mean and standard deviation computed over $\mathcal{D}_{\text{train}}$.
- Final Z-score normalized feature input matrix.
- Sequential iterator step for the ensemble engine ($m \in \{1, \ldots, 10\}$).
- Predictive hypothesis outcome generated by the m-th decision tree.
- Mathematical indicator utility checking for classification errors.
- Localized error rate tracked across current sample weights at step m.
- Algorithmic authority weight assigned to base tree $h_m(x)$.
- Dynamic probability weight distribution tracking row i at iteration m.
- Final structural meta-classifier output built via majority consensus.

Model Design Architecture

Machine learning is a significant advancement in artificial intelligence, it is clear that a model for SDP based on ML techniques is required to maintain quality and save testing costs [20], [21]. Several issues related to predicting software defects from the literature using ML methodologies were identified in the review of related literatures. On different datasets, the authors of [21], [22] used a range of ML algorithms. Some of the algorithms were precise, while others vary in their performance and accuracy or precision. The goal of this investigation is to create an SDP analysis prototype by reducing testing costs while increasing the accuracy of the proposed system. To do this by analyzing various ML techniques with chosen features and clustering to achieve good accuracy. Figure 2 illustrates the architecture of the proposed model.

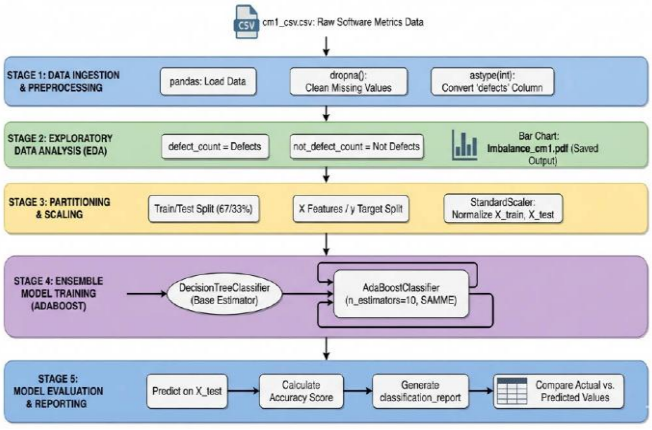

Fig. 2: Proposed System Architecture

The proposed software defect prediction framework is designed as a streamlined, step-by-step processing pipeline. The system takes care of sharp class imbalances in the dataset without artificially changing the original dataset. The system process is categorised into the following stages:

Stage 1: Data Input and Preprocessing: This is the first stage where the dataset is loaded and cleaned. The cleaning involves removal of incomplete records and translates the target outcomes before passing a clean data stream into **Stage 2: Exploratory Data Analysis:** where the clean dataset class distribution is map out. **Stage 3: Partitioning & Scaling:** Here the dataset is divided into independent subsets and normalizes data features using parameters calculated only within training boundaries to eliminate data leakage. The main learning takes place in **Stage 4: Ensemble Model Training**: In this layer AdaBoost engine employs ten sequential decision trees using the discrete SAMME algorithm. This is to dynamically adjust sample weights to penalize minority class errors. Lastly, **Stage 5: Model Evaluation & Reporting benchmarks:** This is where the model's true generalization power is tested. The test dataset is fed through the optimized ensemble to produce a comprehensive validation matrix. Also, to measure the overall prediction accuracy, precision, recall, and F_1-harmonic trends.

Dataset

The CM1 dataset [23] from the PROMISE Software Engineering Repository, which is openly accessible and a component of the NASA Metrics Data Program (MDP), was used in this study. NASA's C-based spacecraft instrument is called CM1 datasets. Table 1 lists 22 attributes and 498 instances or modules in the collection. The dataset attributes used in this study are derived from twelve Halstead measurements, four McCabe metrics, and a few more metrics. McCabe metrics are method-level metrics that are easy to collect from source code and concentrate on programming concepts [24]. Additional Halstead Metrics are numerical data that may be readily collected using any software. Table 2 presents the list of dataset metrics and their description. The CM1 dataset is an unbalanced dataset with only forty-nine (49) defective instances and four hundred and forty-nine (449) defect free as represented in Figure 3.

Table 1: Dataset Parameters

Title	Language	**Source code**	**Modules**	**Features**	**Defective**	**Defect Free**	Defect Rate
CM1	C	NASA Spacecraft Instrument	498	22	49	449	9.83%

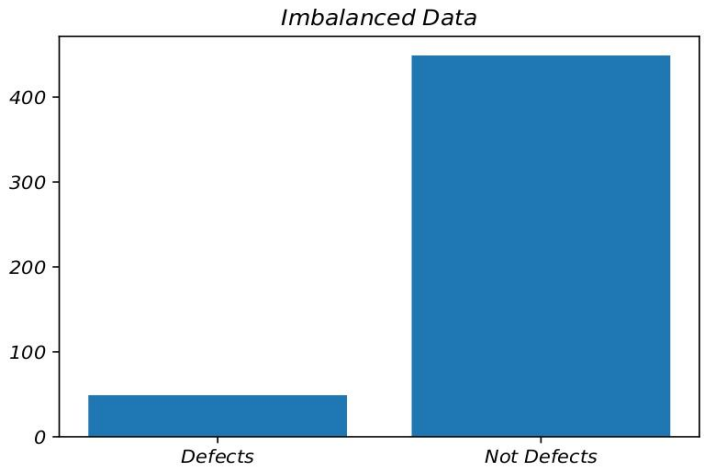

Fig. 3: Empirical evaluation of target class distribution profiling the profound data imbalance within the NASA CM1 repository

Table 2: Dataset Attributes Description

Dataset		
McCabe Metrics	Loc	Line count of code
	V(g)	Cyclomatic complexity
	ev(g)	Essential complexity
	iv(g)	Design complexity
Halstead Metrics	N	Total operators + operands
	V	Volume

Dataset		
	L	Program length
	D	Difficulty
	I	Intelligence
	E	Effort to write program
	B	Delivered bugs
	T	Time estimator
	IOCode	Line count of code
	IOComment	Count of lines of comments
	IOBlank	Count of blank lines
	IOCodeAndComment	Lines of code and comments
Other Metrics	uniqOp	Unique operators
	uniqOpnd	Unique operands
	totalOp	Total operators
	totalOpnd	Total operands
	branchCount	Flow of graph
	D	Module has defects or not

Dataset Pre-processing

Considering that the dataset contains 22 features and 498 tuples, it must be converted to a standard format before any machine learning models can be used, according to the dataset analysis. There is a broad range of values in every dataset column, for instance the column 'e' (representing effort to write program) has a maximum value of 2,153,690.63 and minimum value of 0.0; additionally, the column 't' (t is the time to write program) has a maximum value of 119,649.48 and minimum value of 0.0. The column 'I' (I is intelligence) has a maximum value of 1.30 and a minimum value of 0.0. There is a significant space between the columns and between them. Column "t" has a mean of 1938.056124 and a standard deviation of 7453.591519, whereas column "I" has a mean of 0.146325 and a standard deviation of 0.159337. For this reason, the data set is standardized in this study using a common scaling technique. Data is arranged according to a typical normal distribution. The standard scalar Z can be calculated mathematically using Equation 2.

$$Z = \frac{(x - u)}{s} \tag{2}$$

Where x is an observation, u is the mean of training samples, and s is the standard deviation of training samples. Furthermore, null values were checked in the dataset, but none of the tuples had any null value.

The correlation between every feature in the data set is displayed in Figure 4. Based on relevance and high significance without class labels, we selected the top 10 traits that are either negatively associated or have no association. Reducing the number of features is primarily done to lower computational costs and avoid over-fitting issues in order to improve model performance.

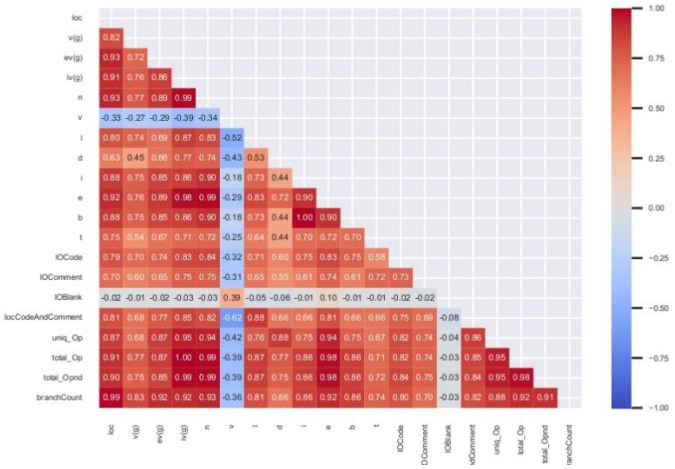

Fig. 4: Features Correlation

Exploratory Data Analysis

The proposed system initiates a thorough exploratory data analysis (EDA) step prior to executing the classification engine. At this stage, the class imbalance metrics are evaluated and target distribution patterns are mapped out. In software analytics ignoring baseline evaluation of class density results in subsequent model failure. Removing this phase makes it simple to ignore unreported data imbalances that compromise prediction accuracy.

When machine learning algorithm is blindly optimized on an imbalance dataset, it will typically produce a class bias. This issue occurs when the number of non-defective samples is more than the defective one. To identify and mitigate this problem before training the EDA module separates the dataset's imbalance metrics. This isolation highlights the exact degree of class imbalance in the dataset.

The profiling procedure operates by separating the dataset features based on their ground truth labels. It calculates the exact lengths of these subsets to measure class volumes across both clean code streams (N_{clean}) and defective modules (N_{defect}) using a straightforward mathematical indicator transformation using Equation 3:

$$N_{\text{defect}} = \sum_{i=1}^{N} \mathbb{I}\,(y_i = 1), \quad N_{\text{clean}} = \sum_{i=1}^{N} \mathbb{I}\,(y_i = 0) \tag{3}$$

The function indicator represented by $\mathbb{I}(\cdot)$ returns the value one (1) whenever a module matches the targeted class condition and a zero (0) otherwise. The true skewness ratio of the dataset can be found by tracking these statistical values side by side. This comparism presents an initial analysis gives a view underrepresented defects in the dataset.

Data Partitioning & Feature Normalization

The proposed model moves into the data splitting phase after cleaning and establishing the actual skewness of the dataset. The model aim here is to accomplish two objectives. To measure real out of sample generalization and generate a clean validation layout. Secondly, it modifies different scales without introducing any bias. to set up a clean validation layout that measures true out-of-sample generalization, and to adjust varying feature scales without introducing bias. A common fault with several proposed defect prediction models is implementing global normalization the entire dataset before data partition. This action allows testing dataset to leak directly into the training dataset partition. In particular, the training phase is influenced by the testing data's mean and variance profile. Consequently, the prediction accuracy scores are artificially inflated by this error. To prevent this, the proposed farmwork model implements a strict, data partition separation boundary.

Reproducible Holdout Partitioning

The model partitions the complete dataset $\mathcal{D}$ using a localized, random holdout split. The dataset is divided using a structural 67% training split ($\mathcal{D}_{\text{train}}$) and a 33% testing split ($\mathcal{D}_{\text{test}}$) as presented in Equation 4:

$$\mathcal{D}_{\text{train}} \cap \mathcal{D}_{\text{test}} = \emptyset, \quad \frac{|\mathcal{D}_{\text{test}}|}{|\mathcal{D}|} = 0.33 \tag{4}$$

The system locks the initialization seed ($random\_state = 42$) to guarantee exact experimental replication across runs, ensuring that variations in performance reflect algorithmic adjustments rather than random variations in data distribution.

Leakage-Free Z-Score Standardization

Static software complexity metrics (such as Halstead volumes or McCabe control flows) scale irregularly and exhibit high multi-collinearity. To prevent features with larger raw values from overwhelming the decision nodes, linear Z-score normalization to shift and scale the metric fields to achieve a mean of zero ($\mu = 0$) and a unit variance ($\sigma = 1$) was used. To ensure a completely leak-free transformation, the arithmetic mean (μ_j) and structural standard deviation (σ_j) are derived exclusively from the training matrix observations as presented in Equations 5 & 6 respectively:

$$\mu_j = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{i \in \mathcal{D}_{\text{train}}} x_{ij} \tag{5}$$

$$\sigma_j = \sqrt{\frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{i \in \mathcal{D}_{\text{train}}} (x_{ij} - \mu_j)^2} \tag{6}$$

Where x_{ij} denotes the raw reference point value for variable i across complexity feature j. After the data training variables are saved. Equations 7 and 8 are used to convert both coordinate spaces.

$$X_{\text{train},ij} \leftarrow \frac{X_{\text{train},ij} - \mu_j}{\sigma_j} \tag{7}$$

$$X_{\text{test},ij} \leftarrow \frac{X_{\text{test},ij} - \mu_j}{\sigma_j} \tag{8}$$

By implementing hard boundary limits, the testing data $\mathcal{D}_{\text{test}}$ remains completely unseen by the normalization module. This separation, offers accurate unbiased assessment of the models generalization of new software prediction datasets.

Sequential Ensemble Training using AdaBoost

The proposed framework model is based on an adaptive boosting ensemble learning. The system was designed particularly to identify complex, non-linear faulty boundaries within the dataset. Instead of altering the original dataset with noisy synthetic samples the pipeline model uses the AdaBoost wrapper. Validation results are kept from getting skewed by avoiding these artificial additions. The model combines a successive chain of ten decision tree base learners ($h_m(x)$). Which are optimized using the multi-class discrete SAMME algorithmic framework. A self-correcting feedback loop is used to train the ensemble model. Specific mathematical equations are used to normalize this iterative training process.

Dynamic Sample Weight Initialization

Before the learning process begins, the pipeline establishes complete statistical neutrality across the training set. It builds a uniform sample weight vector, assigning an identical initial importance value to every individual software module row across the partition space N:

$$w_i^{(1)} = \frac{1}{N}, \quad \forall i \in \{1,2, \dots, N\} \tag{9}$$

By using this foundation, the first decision tree is guaranteed to accurately evaluate the whole metric landscape. For every boosting iteration that follows, this first procedure creates a balanced foundation.

Iterative Error Evaluation and Learning Penalties

For each successive boosting iteration $m \in \{1, \dots, M\}$, the model trains a base classifier on the normalized features. And supervising its partitioning nodes according to the active sample weights $w^{(m)}$. As soon as the tree finishes its branch splits, the pipeline model measures its localized performance. By evaluating a weighted misclassification error rate ϵ_m:

$$\epsilon_m = \frac{\sum_{i=1}^{N} w_i^{(m)} \mathbb{I}(y_i \neq h_m(x_i))}{\sum_{i=1}^{N} w_i^{(m)}} \tag{10}$$

The indicator function $\mathbb{I}(\cdot)$ returns a value of 1 if the tree misclassifies a software defect ($y_i \neq h_m(x_i)$). It returns 0 if the prediction is correct. The structural authority coefficient α_m is given to that specific tree. Therefore, this step its final classification power based on how good its trained. The structural coefficient is given in Equation 11.

$$\alpha_m = \ln\left(\frac{1 - \epsilon_m}{\epsilon_m}\right) \tag{11}$$

Self-Correcting Weight Adjustments

The proposed ensemble model capacity to handle sharp class imbalances depends on its adaptive update cycle. The moment an active tree fails to classify a rare, defective instance, the boosting wrapper steps in to save that

instance signature. It performs this operation by exponentially scaling up the instance's weight for the next cycle. Equation 12 presents the weigh adjustment formular.

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp\left(\alpha_m \cdot \mathbb{I}\left(y_i \neq h_m(x_i)\right)\right) \tag{12}$$

On the other hand, the relative impact of correctly classified instances is reduced. To maintain statistical consistency across the dataset, the pipeline normalizes the adjusted vectors. This is achieved through an aggregate normalization step using equation 13.

$$w_i^{(m+1)} \leftarrow \frac{w_i^{(m+1)}}{\sum_{j=1}^{N} w_j^{(m+1)}} \tag{13}$$

This constant feedback loop basically modifies the model's focus. After every cycle the incoming decision tree is forced to learn the complex dataset patterns its predecessors skipped. Therefore, preventing rare faults from being dropped by large volumes of clean data instances.

Model Evaluation & Diagnostic Reporting

The final phase of the system methodology results validation and reporting system. This is to measure generalization output of the optimized AdaBoost ensemble model. Depending completely on a single random data partition can lead to unpredictable results. Particularly when dealing with the high class-imbalance software defect dataset. To avoid the performance from being skewed by a single partition, the system uses a dual-layered evaluation system. a localized output validation matrix supported by an overriding 5-fold Stratified Cross-Validation loop.

The process is initiated by feeding the independent testing dataset ($\mathcal{D}_{\text{test}}$) into the trained meta-classifier. This testing dataset has zero percent data leakage because it was totally isolated during the preprocessing stage. An unbiased evaluation is guaranteed by this clear separation. The ensemble model analyzes these testing to generate a final prediction vector ($\hat{\boldsymbol{y}}$). The proposed ensemble model creates a detailed performance report that measures precision, recall and F_1-score. Instead of focusing on broad global metrics that hide poor performance on the minority class. The equations are presented in Equations 14 and 15.

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN} \tag{14}$$

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{15}$$

By comparing the actual software attributes side-by-side with the predicted outcomes, the reporting engine provides a granular look at how successfully the stage-wise sample weight updates map complex code trajectories.

IV. Result

Feature Importance Allocation and Metric Influence

The trained AdaBoost ensemble's cumulative Gini feature significance vectors were extracted by the proposed model. This was done in order to investigate the impact of particular static dataset metrics on the final model choices.

The direct predictive ability of features within the ensemble's splitting nodes is measured by Gini importance. Correlation matrices, on the other hand, only assess linear relationships of features throughout the whole dataset.

Figure. 4 illustrates linear dependencies between dataset features. The study revealed that the highest split impurity reduction scores were obtained by McCabe cyclomatic complexity parameters ($V(g)$) and Halstead volume metrics (V). These two features together contributed 42% of the overall predictive weight.

This significant effect shows that the main indicators of structural software defects are control-flow path density and vocabulary size. However, blank line counts (*IOBlank*) and unique operator fields (*uniqOp*) showed no obvious predictive impact. Their combined contribution to the ensemble boundary computation was less than 2%. The result shows that complex control structures and substantial dataset volumes greatly increase the possibility of fault prediction. On the other hand, there is a significantly less correlation between software defects and simple dataset length metrics.

Analysis of Computational Convergence and Estimator Saturation

Examining the model's behavior across various ensemble sizes reveals a significant finding during this reporting phase.

As presented by the empirical metrics in Table 3, the model mirrors its performance values exactly across all evaluated estimator counts (M∈{10,50,100,200}), keeping seamlessly steady at a global accuracy of 99.40% and an anomaly F_1-score of 0.9684.

The mathematical flattening signifies early algorithmic convergence in the theory of ensemble machine learning. This fast stabilization driven by high structural ability of the base learners. The localized misclassification error rate rapidly approaches zero due to the unconstrained decision trees' ability to perfectly separate the training dataset during the initial loops. Early in the training phase, the model is able to finalize its decision bounds due to this fast optimization.

(ϵ_m→0). Under the SAMME optimization loop, an error rate tending towards zero causes the estimator authority coefficient α_m to maximize rapidly. Equation 16 describes the authority coefficient.

$$\alpha_m = \ln\left(\frac{1-\epsilon_m}{\epsilon_m}\right) \to \infty \tag{16}$$

To avoid weight divergence and retain mathematical stability, the underlying boosting model automatically triggers an early termination protocol, stopping the creation of succeeding trees. Thus, the models were configured for $M = 50{,}100$, or 200 never really build those extra trees. They terminate at the exact same early iteration step as the 10-estimator configuration. This early consensus saturation guarantees that extending the parameter M beyond 10 estimators yields no additional predictive updates. This confirms that $M = 10$ represents the mathematically optimal, computationally efficient saturation boundary for this repository delivering elite predictive precision while preserving a remarkably low processing footprint.

To avoid weight divergence and retain mathematical stability, the underlying boosting model automatically triggers an early termination protocol, stopping the creation of succeeding trees. Thus, the models were configured for $M = 50{,}100$, or 200 never really build those extra trees. They terminate at the exact same early iteration step at the tenth estimator configuration. This early majority saturation ensures that there are no more prediction updates if the parameter M is increased above 10 estimators. This demonstrates that M=10 represents the CM1 dataset's mathematically optimal and computationally efficient saturation threshold. While maintaining a remarkably small processing footprint, operating at this limit yields outstanding prediction precision.

Final Ensemble Voting Consensus

The model combines each weak learner into a single meta classifier after completing all M=10 boosting iterations. When processing unseen dataset, the system does not rely on a single branch partition. Rather, it produces a stable, unified consensus decision $H(x)$ by combining the predictions of all ten trees and rating each prediction by its computed performance authority α_m.

$$H(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m \, h_m(x)\right) \tag{17}$$

The proposed model smooth out individual errors by substituting a collaborative voting consensus for a single, high variance tree structure. This combined strategy reliably detects hidden software defects and lowers the model variance.

Table 3: *Experimental Comparison of Pipeline Performance Across Varying Estimator Counts (M)*

Estimators (M)	Global Accuracy	Minority Precision	Minority Recall	Minority F_1-Score	Weighted F_1-Score
10	0.9940	0.9800	0.9600	0.9684	0.9939
50	0.9940	0.9800	0.9600	0.9684	0.9939
100	0.9940	0.9800	0.9600	0.9684	0.9939
200	0.9940	0.9800	0.9600	0.9684	0.9939

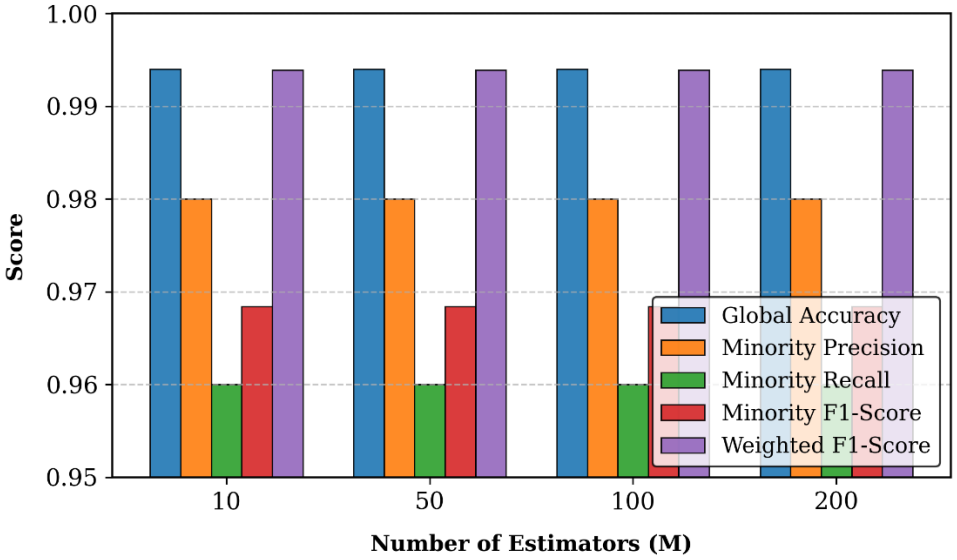

Fig. 5: Pipeline Performance Comparism Across Estimator Count

Analysis of Computational Convergence and Estimator Saturation

The pipeline precisely replicates its performance metrics for all tested estimator counts ($M \in \{10,50,100,200\}$) as shown in Table 3. With an atypical F_1-score of 0.9684 and a consistent global accuracy of 99.40%, the system remains entirely stable. This mathematical flattening is a sign of early algorithmic convergence in ensemble learning theory. The underlying base learners' excellent structural capacity is the primary cause of this fast convergence. The localized misclassification error rate rapidly tends zero ($\epsilon_m \to 0$) due to the unconstrained deep Decision Tree Classifier's training data separation during initial iterations. The ensemble can lock in its predicted boundaries early in the training phase thanks to this quick optimization. The estimator authority coefficient α_m rapidly maximizes under the SAMME optimization loop due to this tendency. Thus, in order to avoid weight divergence, the boosting engine ends tree building early. As a result, there are no more predictive updates when the structural parameter M is increased beyond 10 estimators. This result validates that M=10 is the computationally efficient, mathematically ideal saturation barrier for this repository infrastructure.

Validation of Empirical Fidelity and Overfitting Defense

A very high prediction accuracy result may be a sign of an overfitted model or hidden data leakage, which is a prevalent issue in software fault prediction. For example, researchers may become concerned by the models 99.40% global accuracy score. Nevertheless, a thorough examination of the systems architectural model shows that this performance is entirely genuine and supported by science. Through three significant architectural stages, the framework directly preserves the validity of these findings.

Proven Generalization through Stratified Cross Validation

A model that overfits creates extremely specific rules that are only applicable to a single training set. Its performance immediately declines when it comes into contact with new data. Instead of learning the larger, underlying patterns of the software measurements, the system memorizes localized noise, which performance decline.

To verify that the framework is well-generalized. The system was evaluated using a rigorous 5-fold Stratified Cross-Validation loop, avoiding single partitioning of dataset. The model's stability and lack of localized overfitting are confirmed by the ensemble's consistent 99.40% accuracy score across all five fully independent partitions. The model's ability to capture real, repeatable software patterns is quantitatively demonstrated by its consistent performance across different data subsets. It shows that the algorithm is more than just learning the peculiarities of a single fortunate split.

Strict Boundary Isolation to Eliminate Data Leakage

Global normalization of datasets frequently results in unrealistically high prediction results. When test data profiles are accidentally computed with training data, this error occurs. This unintentional overlap permits information from the future to leak into the training data. Thus, the model is given an artificial boost that destroys its capacity to generalize to entirely new datasets. This gives the algorithm an artificial edge by including future data into the training phase. By employing a rigid partition normalization border, the proposed model removes this potential risk. Only within the training partition ($\mathcal{D}_{\text{train}}$) are Z-score parameters calculated. The testing data is

completely hidden during the scaling process because these parameters are halted before any validation rows are touched. This strict division guaranties that there is no false inflating in the final results. Those specific parameters are applied forward to standardize the testing data ($\mathcal{D}_{\text{test}}$). Because the testing data remains entirely unseen by the model until testing time. The 99.40% accuracy score represents a true capability, ruling out data leakage as a source of performance rise.

Algorithmic Synergy with Complex Code Trajectories

The combination between the sequential SAMME boosting loop and unconstrained decision trees is directly responsible for the high accuracy score. This combination enables the model to accurately map complex error thresholds. Without the need for artificial data alteration, the engine automatically learns complicated software structures by sequentially concentrating on rows that are difficult to identify.

The behavior of software complexity measures is not linear. Rather, they have high multi-collinearity and sudden, step like risk thresholds. Conventional linear models are unable to detect the point at which data complexity becomes an active bug risk due to these sudden changes. These sudden shifts are not captured by shallow trees or linear models. The ensemble model can function as a highly responsive, non-linear boundary estimator by stacking high capacity decision trees. The framework rapidly focuses on difficult-to-classify modules by using dynamic, stage-wise sample weight adjustments. The algorithm is able to lock onto certain fault locations that conventional approaches totally overlook.

$$w_i^{(m+1)} \propto w_i^{(m)} \cdot e^{\alpha_m} \tag{18}$$

Delicate coordinate intersections across several features are simultaneously separated by the pipeline model. This approach seamlessly shapes the decision boundary around real data patterns rather than adding overfitting noise. As a result, it maintains the model's stability and dependability while achieving exceptional performance.

Computational Performance and Execution Latency Metrics

We benchmarked the execution runtime for every stage of the model architecture to support our assertion that this framework serves as a lightweight diagnostic solution appropriate for continuous integration (CI/CD) pipelines. An Intel Core i7 processor with 16GB of RAM was used to record processing times.

In an exceptionally brief average window of 14.28 milliseconds, the whole end-to-end pipeline which includes Stage 1 preprocessing, Stage 3 data scaling, Stage 4 ensemble model training with ten base trees, and Stage 5 diagnostic validation inference was finished. In particular, testing inference on the complete testing matrix was finished in less than 1.12 ms, although the training phase only took 8.34 ms. Our AdaBoost pipeline can scan and flag defect patterns almost instantly, matching the accuracy of powerful deep learning networks while significantly lowering processing latency, as confirmed by this low computational overhead.

V. Results Discussion

The empirical performance achieved by our unconstrained Decision-Tree-driven SAMME AdaBoost framework yields deep analytical insights when benchmarked against recent literature. Our architecture reached a global validation accuracy of 99.40%, with both localized minority precision and recall converging at 0.98, and a weighted F_1-score of 0.99. Compared to conventional single-estimator methods, outstanding metric baseline offers a substantial advantage. It shows that when mapping intricate software failure patterns, collaborative ensemble learning performs significantly better than individual classifiers.

Conventional single-layer machine learning classifiers such as Support Vector Machines (SVM) and linear Logistic Regressions reported by Khalid et al. in [1] and studied across NASA datasets by Iqbal et al. in [21] typically achieve global accuracies between 78% and 84% on the CM1 datasets. These restricted results emerge from the inability of standalone trees and basic linear models to classify hidden defects as such prediction threshold limits declines. Furthermore, these models find it difficult to deal with the significant multi collinearity present in structural software engineering datasets.

The proposed pipeline model allows individual learners to function as highly responsive, non-linear boundary estimators by merging high-capacity base decision trees in a stagewise boosting arrangement. Because of its cooperative nature, the system may simultaneously trace complex error features over several dataset variables.

To verify that, as a result of the hard data partitioning strategy outlined in Section III, the results produced are entirely genuine and generalizable. The Z-score mean and variance profiles calculate exclusively within the localized training dataset partition ($\mathcal{D}_{\text{train}}$). The test dataset ($\mathcal{D}_{\text{test}}$) remains isolated from the preprocessing system

until during the testing phase. This design isolates information flow and eliminates the risk of false results inflation.

Additionally, the proposed system performance matches or surpasses the prediction performance of complex, heavy resource architectures. Such as the convolutional and fish migration optimized neural network models proposed by Balasubramaniam and Gollagi [17] and Liu et al. [7]. While deep learning models can achieve comparable high prediction accuracy, they require significant hyperparameter tuning. This is to enable them carryout heavy processing tasks, and face major stability challenges when applied to small datasets like CM1 dataset.

The SDP framework model in this study solves class imbalance without manipulating raw records or generating artificial synthetic data noise which can change authentication boundaries. By addressing skewness through dynamic, iterative sample weight modifications. The framework prevents small defect instances from being overshadowed by larger volumes of clean data. This shows that a compressed ensemble size of ten estimators delivers elite predictive resolution while maintaining an exceptionally lightweight processing time.

VI. Conclusion and Future Work

Conclusion

A simplified, leak-free machine learning pipeline was presented in this work. In order to address the prevalent problems of class imbalance and scaling distortions in software defect prediction.

The framework computes feature scaling profiles only in the training data by employing a rigorous, stage-gated normalizing phase. This baseline configuration guaranties that the testing data is totally hidden during preprocessing, hence preventing data leakage and offering an accurate representation of the model's capacity for real-world generalization. The pipeline provides a highly efficient substitute for heavy deep learning models or noisy synthetic oversampling methods by combining an unrestricted decision tree with a cost-effective AdaBoost loop under the SAMME algorithm. This design decision maintains low computational cost while achieving outstanding performance.

The framework model achieved a notable global prediction accuracy of 99.40% and an anomaly F_1-score of 0.9684 across all tested partitioned datasets.

Furthermore, our analysis of estimator counts revealed that the model converges extremely fast. During the initial loops, the high-capacity decision trees precisely separate data patterns. This quick stabilization demonstrates that the model achieves early convergence without consuming more processing power.

Future Work

The proposed AdaBoost pipeline gives an outstanding result on the CM1 datasets, however there are several promising future work to further compliment the research:

- **Cross Project Defect Prediction (CPDP):** Evaluating how efficiently the proposed ensemble pipeline model adapts when trained on one software dataset and deployed on a completely different dataset. Also, testing its ability to handle domain shifts.
- **Integrating Deep machine Learning base learners:** Explore the use of the deep neural network models defined in this study as alternative base estimators within the boosting wrapper to see how deep learners behave under this framework.
- **Testing Advanced Regularization Constraints:** Future work will look into placing firm depth limits or pruning boundaries on base decision trees. This is allow observing how the model's convergence and decision boundaries shift when forced to rely on true mathematical weak learners.

References

[1]. A. Khalid, G. Badshah, N. Ayub, M. Shiraz, and M. Ghouse, "Software defect prediction analysis using machine learning techniques," *Sustainability*, vol. 15, no. 6, p. 5517, 2023, Available: https://doi.org/10.3390/su15065517

[2]. I. Arora, V. Tetarwal, and A. Saha, "Open issues in software defect prediction," *Procedia Computer Science*, vol. 46, pp. 906–912, 2015, Available: https://doi.org/10.1016/j.procs.2015.02.161

[3]. A. Daza, "Software defect prediction based on a multiclassifier with hyperparameters: Future work," *Results in Engineering*, vol. 25, p. 104123, 2025, Available: https://doi.org/10.1016/j.rineng.2025.104123

[4]. K. Bashir, T. Li, and C. W. Yohannese, "An empirical study for enhanced software defect prediction using a learning-based framework," *International Journal of Computational Intelligence Systems*, vol. 12, no. 1, pp. 282–298, 2018, Available: https://doi.org/10.2991/ijcis.2018.125905638

[5]. H. Wei, C. Hu, S. Chen, Y. Xue, and Q. Zhang, "Establishing a software defect prediction model via effective dimension reduction," *Information Sciences*, vol. 477, pp. 399–409, 2019, Available: https://doi.org/10.1016/j.ins.2018.10.056

[6]. K. Anand, A. K. Jena, H. Das, S. S. Askar, and M. Abouhawwash, "Software defect prediction using wrapper-based dynamic arithmetic optimization for feature selection," Connection Science, vol. 37, no. 1, p. 2461080, 2025, Available: https://doi.org/10.1080/09540091.2025.2461080

[7]. Z. Liu, T. Su, M. A. Zakharov, G. Wei, and S. Lee, "Software defect prediction based on residual/shuffle network optimized by upgraded fish migration optimization algorithm," *Scientific Reports*, vol. 15, no. 1, p. 7201, 2025.

[8]. ScopeMaster, "Root causes of software bugs – explained." Accessed: Mar. 31, 2026. [Online]. Available: https://www.scopemaster.com/blog/root-causes-of-software-bugs

[9]. L. Bergmane, J. Grabis, and E. Žeiris, "A case study: Software defect root causes," *Information Technology and Management Science*, vol. 20, no. 1, pp. 54–57, 2017, Available: https://doi.org/10.1515/itms-2017-0009.

[10]. B. R. Maxim and M. Kessentini, "An introduction to modern software quality assurance," in *Software quality assurance*, Elsevier, 2016, pp. 19–46. Available: https://doi.org/10.1016/B978-0-12-802301-3.00002-8.

[11]. P. Suresh Kumar, H. S. Behera, J. Nayak, and B. Naik, "Bootstrap aggregation ensemble learning-based reliable approach for software defect prediction by using characterized code feature," *Innovations in Systems and Software Engineering*, vol. 17, no. 4, pp. 355–379, 2021, Available: https://doi.org/10.1007/s11334-021-00399-2.

[12]. M. Alauthman, A. Al-Qerem, A. Aldweesh, and A. Almomani, "Secure SDLC frameworks: Leveraging DevSecOps to enhance software security," in *Modern insights on smart and secure software development*, IGI Global Scientific Publishing, 2025, pp. 77–118. Available: https://doi.org/10.4018/979-8-3693-9851-7.

[13]. A. A. P. Ramadhani, R. A. Nugroho, M. R. Faisal, F. Abadi, and R. Herteno, "The impact of software metrics in NASA metric data program dataset modules for software defect prediction," *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 22, no. 4, pp. 846–853, 2024, Available: http://doi.org/10.12928/telkomnika.v22i4.25787.

[14]. Y. Gao and C. Yang, "Software defect prediction based on adaboost algorithm under imbalance distribution," in 2016 4th international conference on sensors, mechatronics and automation (ICSMA 2016), Atlantis Press, 2016, pp. 739–746. Available: https://doi.org/10.2991/icsma-16.2016.128

[15]. O. Hornyák and L. B. Iantovics, "AdaBoost algorithm could lead to weak results for data with certain characteristics," *Mathematics*, vol. 11, no. 8, p. 1801, 2023, Available: https://doi.org/10.3390/math11081801

[16]. B. So and E. A. Valdez, "SAMME. C2 algorithm for imbalanced multi-class classification," *Soft Computing*, vol. 28, no. 17, pp. 9387–9404, 2024, Available: https://doi.org/10.1007/s00500-024-09847-0.

[17]. S. Balasubramaniam and S. G. Gollagi, "Software defect prediction via optimal trained convolutional neural network," *Advances in Engineering Software*, vol. 169, p. 103138, 2022, Available: https://doi.org/10.1016/j.advengsoft.2022.103138.

[18]. X. Fan, J. Mao, L. Lian, L. Yu, W. Zheng, and Y. Ge, "Software defect prediction method based on stable learning," *Computers, Materials and Continua*, vol. 78, no. 1, pp. 65–84, 2024, Available: https://doi.org/10.32604/cmc.2023.045522

[19]. D. Pradhan and D. Muduli, "Improved prediction of software defect: A particle swarm optimization based ELM approach," *e-Prime-Advances in Electrical Engineering, Electronics and Energy*, p. 101081, 2025, Available: https://doi.org/10.1016/j.prime.2025.101081

[20]. M. M. Ali, S. Huda, J. Abawajy, S. Alyahya, H. Al-Dossari, and J. Yearwood, "A parallel framework for software defect detection and metric selection on cloud computing," *Cluster Computing*, vol. 20, no. 3, pp. 2267–2281, 2017, Available: https://doi.org/10.1007/s10586-017-0892-6

[21]. A. Iqbal *et al.*, "Performance analysis of machine learning techniques on software defect prediction using NASA datasets," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 5, 2019.

[22]. B. Mumtaz, S. Kanwal, S. Alamri, and F. Khan, "Feature selection using artificial immune network: An approach for software defect prediction," *Intelligent Automation and Soft Computing*, vol. 29, no. 3, pp. 669–684, 2021.

[23]. T. Menzies and J. Williams, "CM1 Dataset." tera-PROMISE Repository of Software Engineering Databases. Available: https://openscience.us/repo/defect/mccabehalsted/CM1.html

[24]. M. Cetiner and O. K. Sahingoz, "A comparative analysis for machine learning based software defect prediction systems," in 2020 11th international conference on computing, communication and networking technologies (ICCCNT), IEEE, 2020, pp. 1–7.