International Research Journal of Multidisciplinary Scope (IRJMS), 2025; 6(2): 710-737

Original Article | ISSN (O): 2582-631X DOI: 10.47857/irjms.2025.v06i02.03365

The IntelliEstimator: Estimating Maintenance Cost and Prediction of Software Quality, Reliability, and Maintenance Using Stacking RFCXGB Classifier

Sreeramkumar T[1]*, O Rajalakshmi Karthika[2], Jayapratha C[3], J Naveen Ananda Kumar[4], Govindaprabhu GB[5]

[1]Department of Computer science, Madura College, Madurai, Tamilnadu, India, [2]Madurai Kamaraj University College, Madurai, Tamilnadu, India, [3]Department of Computer Science and Engineering, Karpaga Vinayaga College of Engineering and Technology, Madhuranthagam, Tamilnadu, India, [4]Tekinvaderz LLC, Florida, USA, [5]Department of Computer Science, MKU, Madurai. Tamil Nadu, India. *Corresponding Author's Email: tsreeramkumar31@gmail.com

Abstract

Maintaining software is critical, but it can be difficult to estimate quality, reliability, effort, and costs. To accurately predict these key parameters, we propose ML-PEQRM, a novel machine-learning model. A model estimates software quality and reliability based on code complexity, maintainability, and size. It also predicts maintenance costs. The proposed ML-PEQRM model utilizes code complexity, maintainability, and size as input features to estimate software quality, reliability, maintenance efforts, and costs. The dataset comprises 25 projects with 10,000 samples of code changes and maintenance activities. A 70-30 split created training and test datasets. Conventional estimation approaches have limitations including 25% average error, unreliable predictions, and resource inefficiency. Static code attributes related to complexity and prior changes increasing complexity by 10% were most informative. Integrating product and process data decreased maintenance costs by 25% and improved reliability by 20%. Novelty lies in integrating essential metrics for maintenance cost estimation and deriving new metrics using machine learning. Static code attributes and change metrics are identified as most significant features. Novel metrics further improve performance. This makes valuable contributions by developing an accurate, practical model that organizations can leverage to enhance planning and efficiency of software maintenance activities. By leveraging code complexity, maintainability, and size as inputs, the ML-PEQRM model provides a data-driven approach improving accuracy and reliability of quality, reliability, maintenance, and cost estimation to 99%. This enables optimization of maintenance costs, reduction in downtime, and predictive maintenance. It allows development of predictive models to enhance the accuracy of maintenance operations to 99%.

Keywords: Cost Estimation, MGFPA, Machine Learning, Random Forest, Stacking Classifier, Software Maintenance, XGB.

Introduction

Software maintenance involves improving, optimizing, and adapting applications after they are deployed as part of the software development lifecycle. A precise estimate of software quality, reliability, effort, and maintenance costs ensures project success, optimizes resources, and maintains operational efficiency. Traditional estimation models, like COCOMO, have demonstrated significant limitations in providing precise and consistent predictions. Modern software systems are complex, and these models fail to take these factors into account, creating inaccurate forecasts and resource allocations. The software industry continues to evolve, and machine learning has become a promising solution. Using historical data, machine learning models can more accurately predict software quality and reliability while optimizing maintenance costs. However, there are significant research gaps. Most existing models overlook key quality factors, don't integrate product and process metrics comprehensively, and rely on limited datasets, so they can't generalize across domains. The study introduces ML-PEQRM (Machine Learning Predictive Estimation for Quality, Reliability, and Maintenance), a novel machine learning model. ML-PEQRM provides precise estimates for software quality, reliability, maintenance costs, and effort based on key metrics. It provides interpretable predictions that

(Received 30th November 2024; Accepted 23rd April 2025; Published 30th April 2025)

support informed decision-making and optimizes resource allocation. Its novelty lies in its holistic approach to estimating software maintenance parameters, which can be applied to various types of projects. ML-PEQRM provides actionable insights into quality, reliability, and cost estimation, helping organizations optimize maintenance workflows, reduce downtime, and improve operational efficiency. Key challenges addressed in this research include:

- The inability of traditional models to provide reliable long-term estimates, as highlighted by prior studies.
- The lack of integration between quality factors and maintenance estimation, which limits the applicability of existing machine learning models.
- The need for a scalable, interpretable, and practical tool for organizations to improve software maintenance efficiency and cost management.

This research makes several significant contributions to the field of software maintenance, focusing on improving the accuracy and reliability of maintenance cost estimation and software quality prediction. The primary contributions are outlined as follows:

Development of the ML-PEQRM Model: The ML-PEQRM (Predictive Estimation for Quality, Reliability, and Maintenance) model is based on machine learning. Based on code complexity, maintainability, and project size, the model delivers highly accurate predictions.

Integration of Product and Process Metrics: The study introduces a comprehensive framework that combines code smells, cyclomatic complexity, coupling, and cohesion metrics with process metrics. The model is able to capture intricate relationships between these metrics, resulting in more precise and interpretable predictions.

Introduction of Novel Metrics: The research constructs and incorporates new metrics to enhance the model's performance:

- Maintenance Index (MI): Quantifies the maintainability of software based on various density values such as abstraction density, encapsulation density, and implementation smell density.
- Quality Index (QI): Measures software quality by averaging critical density values related to completeness, documentation, and maintainability.
- Security Index (SI): Evaluates security features using metrics like the density of protected methods and encapsulation practices.

The ML-PEQRM model provides a more detailed and actionable assessment of software quality and maintenance needs through several advanced techniques. One of the key enhancements is Feature Engineering Using MGFPA, where feature selection is optimized by employing a Modified Global Flower Pollination Algorithm (MGFPA). This technique refines the ranking of feature importance, making the model both more accurate and efficient in its predictions. Another significant contribution is the Application of Stacking Ensemble Techniques, which combines Random Forest and XGBoost classifiers to further boost predictive performance. This ensemble approach achieves an impressive 99% accuracy, surpassing traditional models and benchmarks in the field. Finally, the model demonstrates its practical value by enabling a Reduction in Maintenance Costs and Increased Reliability. By incorporating the proposed methodology, ML-PEQRM achieves a 25% reduction in maintenance costs and a 20% improvement in reliability, showcasing its effectiveness in real-world software development and maintenance scenarios.

The study enables better planning, optimized resource utilization, and improved decision-making in software maintenance by addressing these objectives. This research contributes to advancing the field by offering an interpretable and holistic approach to predictive maintenance. Traditional software maintenance cost estimation models, like COCOMO II and regression-based techniques, fail to accurately predict software quality, reliability, and maintenance costs across diverse projects. They lack integration of critical product and process metrics, rely on limited datasets, and do not account for factors like code smells, cyclomatic complexity, or maintainability, which significantly impact project outcomes. Existing machine learning approaches are often too narrow or function as black-box systems, making their predictions difficult to interpret and apply in real-world scenarios. Integrating critical software metrics with advanced feature selection techniques and ensemble machine learning algorithms will significantly improve the accuracy,

reliability, and interpretability of maintenance cost estimation models. The proposed ML-PEQRM model, incorporating novel metrics like Maintenance Index (MI), Quality Index (QI), and Security Index (SI), will outperform existing approaches.

- Reducing maintenance costs by at least 25%.
- Improving reliability metrics by 20%.
- Achieving a predictive accuracy of at least 99%, validated across diverse software projects and datasets.

This model will provide interpretable predictions, addressing the limitations of traditional approaches. It enabling more effective resource allocation, cost management, and decision-making in software maintenance. The datasets were sourced from publicly available GitHub repositories containing Java projects of varying complexity, maintainability, and size. The dataset includes 25 Java projects with 10,000 code samples, capturing different types of code changes and maintenance activities. Each project folder contains Java files, classes, functions, and identifiers, analyzed to extract various software metrics. A full list of projects is available in the study (Table 1), showcasing a variety of application domains and software architectures. The ML-PEQRM model achieved 99% accuracy in estimating software quality, reliability, and maintenance costs, significantly outperforming COCOMO II and neural network-based methods. It reduced maintenance costs by 25% and improved reliability by 20% compared to traditional methods. Validated with a dataset of 25 Java projects and 10,000 code samples, the model demonstrated robustness and scalability. The integration of product and process metrics, along with key indices like Quality Index (QI), Security Index (SI), and Maintenance Index (MI), provided interpretable and actionable predictions. The Modified Global Flower Pollination Algorithm (MGFPA) optimized feature importance rankings, minimizing noise and over fitting. These results confirm the model's effectiveness in advancing software maintenance cost estimation, quality assurance, and predictive reliability. By applying machine learning to software maintenance parameters, this project aims to develop an accurate and reliable model that enhances maintenance efficiency through data-driven insights. It seeks to optimize resource allocation and address the limitations of existing models by learning from historical project data. A key objective is to establish a comprehensive framework for analyzing critical factors that influence software quality, reliability, maintenance, and cost. To ensure optimal performance, suitable predictive machine learning algorithms will be identified, and parameters will be estimated using code metrics to develop a cohesive and integrated model.

This work emphasizes its contribution to strengthening maintenance planning at a practical level by proposing a model that enhances industry efficiency in a reliable and applicable way. It improves accuracy through the integration of both product and process metrics, and employs a feature engineering approach to identify the most significant features. The model is designed to predict quality-focused outcomes in an interpretable manner, offering clear insights for decision-making. Additionally, it introduces a novel approach to maintenance estimation, aiming to advance the current state of the art in the field. The primary research topic of this study is the enhancement of cost estimation accuracy in construction project management through the integration of artificial intelligence and machine learning techniques. The paper introduces the "IntelliEstimator" framework, which leverages predictive analytics, intelligent decision support, and real-time data processing to address long-standing challenges in traditional estimation methods. This research is driven by the hypothesis that incorporating AI/ML into the estimation process significantly improves accuracy, reduces human error, and increases efficiency. Therefore, the study aims to evaluate whether the IntelliEstimator framework provides a measurable improvement over conventional cost estimation approaches, particularly in dynamic and data-intensive project environments. The research contributes to the field by offering a practical, scalable solution that blends domain knowledge with computational intelligence. Section 2 reviews the literature, identifying gaps in existing methods. Section 3 details the methodology, including the dataset, feature engineering techniques, and the proposed ML-PEQRM model. Section 4 presents the experimental setup, implementation environment, selected features, performance evaluation metrics, comparative analysis, key

findings, and limitations. Section 5 concludes the paper by summarizing contributions, highlighting practical benefits, and suggesting future research directions. The reviewed studies highlight advancements in software quality, cost estimation, and defect prediction, emphasizing innovative methodologies and data-driven models. Various aspects of software quality, team productivity, cost estimation, and defect prediction were explored using diverse metrics and methodologies. For example, this work introduced the Team Homogeneity Index (THI) to measure the impact of team personality traits on software quality and productivity during the SDLC, analysing team dynamics and project outcomes with five metrics (1). This novel work investigated the complexity of Multi-Programming Language Codebases (MPLCs), finding they significantly impact issue resolution times and quality outcomes (2). This highlights the need for effective management strategies in heterogeneous codebases. Issues in MPLCs are resolved 89% slower than in non-MPLCs, with over 90% of MPLCs using source files from two programming languages. This study underscores the importance of optimizing software quality through better resource allocation and process improvements. In a study a theoretical model based on the most important factors of CKM (Customer Knowledge Management) was developed (3). To evaluate the proposed model, survey questionnaires were distributed to decision-makers in ES (Enterprise Software) development companies. Three-year industry-academy collaboration presents SVEVIA, a framework for software quality assessment and strategic decision support (4). A quality-cost-time trade-off was identified by analysing the industrial software quality management process. Methods were developed for assessing, predicting, planning, and optimizing product/process quality. Software metrics based on development data can be used to estimate software reliability (5). An analysis of product and process metrics has the objective of establishing a statistical relationship between them. In the paper, non-parametric models such as Artificial Neural Networks are suggested for estimating the reliability of software and release readiness based on past failure data. This study incorporated considerations of imperfect debugging, a variety of errors, and change points during the testing process to extend the usefulness of SRGM's (6). A limit to testing athletic ability is proposed, but with unlimited time, testing becomes infinite and may not be feasible (7). This method presents endless test execution work for older models of Neural Heterogeneous Poisson Process (NHPP) of Programming model disappointment with proposed information for preparing Artificial Neural Network (ANN). This proposed an automated process of prioritizing bug reports and selecting developers using fuzzy multi-criteria decision-making (8). In the proposed approach, the fuzzy Technique for Order of Preference by Similarity to the Ideal Solution (TOPSIS) method is combined with Bacterial Foraging Optimization Algorithm (BFOA) and Bar Systems (BAR) techniques to build a bug priority queue. It aims to gather decisive and explicit knowledge of bug reports by considering multi-criteria inputs. Software maintenance projects differ from other engineering projects because of certain characteristics (9). The complexity and failure rates of projects have increased. Software projects need to be identified and monitored to increase their chances of success. By combining genetic algorithm (GA) and environmental adaptation (EA) methods, It aimed to optimize COCOMO coefficients for SCE. Based on the results, it is determined that the EA algorithm can solve the divergence problem of the genetic algorithm, as well as optimize the COCOMO coefficients (10). This study was proposed to address the difficulty of estimating software development costs with conventional methods (11). A reliable estimation method is constructed by combining these steps with machine learning approaches to identify the necessary steps for computable entities that affect software costs. With the help of formulae and an online tool, It analyse and compare Boehm's COCOMO model with Valerdi's COSYSMO model (12). The COCOMO dataset was used for this analysis, and the COSYSMO model was observed to perform better in every aspect than the COCOMO model. The work was proposed, which uses a standard Turkish industry dataset to optimize the parameters of the Constructive Cost Model II (COCOMO-II) (13). The IEAM-RP was proposed to predict the development effort (14). To test IEAM-RP's effectiveness, NASA software projects are used for the experiment. Using other method references (such as Use Case Points) and mapping non-functional requirements to the terms of

reference, one past work proposed two core phases. In addition, the second phase is to calculate and compare the estimated effort and cost if the original FP method was modified (15). This work reviews the state of predictive maintenance (PdM) within the context of Industry 4.0, focusing on the integration of machine learning (ML) and reasoning techniques (16). It provides an overview of Digital Twin (DT)-based predictive maintenance strategies. It explains how DTs, virtual replicas of physical systems are used to monitor real-time performance, predict failures, and plan maintenance activities (17).

This work used genetic algorithms to optimize software development cost estimation and it addresses software factor's uncertainty and ambiguity (18). The COCOMO II model formulas were incorporated into the estimation of effort and schedule time. Using NASA data, experiments achieved 98.88% accuracy for scheduled time and 97.27% for effort estimation. This study shows how genetic algorithms combined with parameter fine-tuning can improve software cost estimation. This work developed an Adaptive Neuro-Fuzzy Inference System combining Ant Colony Optimization (19). Various evolutionary algorithms were compared. This model performed software effort estimation on datasets like Albrecht, Desharnais, and Kemerer. It provides enhanced estimation capabilities for software project managers.

This work proposed a two-stage framework for agile cost estimation, linking development and maintenance phases (20). The first stage focuses on development, the second on maintenance, with testing comprising 22% of the workload and management tasks 13%. They also introduced five paradigms for Nesma, a Function Point Measurement method, enhancing the LSTM-CRF model's accuracy and precision. However, the quantity and quality of training samples and labelled texts still need improvement. This work focused on Nesma, a Function Point Measurement method, introducing five paradigms to define heuristic rules for splitting software into Pricing and Measuring Objects (21). This approach enhanced the LSTM-CRF model's accuracy and precision using large-scale information projects as training sets. However, the quantity and quality of training samples and labelled texts still need improvement compared to expert manual audits.

This work proposed an MCDM-based framework to evaluate Software Reliability Prediction models using multiple accuracy measures (22). They assessed ten models with a software failure dataset and four performance measures, identifying SOMFTS as the most suitable model. The findings suggest the MCDM approach is effective for selecting the best software reliability prediction model. This work highlighted Software Defect Prediction, focusing on software quality and reliability (23). Various techniques have been used to classify software as defective or non-defective by analysing source code and development processes. This study introduced a modified isolation forest method for SDP, demonstrating its effectiveness through experiments on five NASA datasets. These studies highlight the importance of interdisciplinary approaches in software engineering, combining team dynamics, advanced modelling, and optimization. However, gaps remain in integrating these dimensions into a unified framework that addresses both technical and human factors. Future research should bridge these areas to achieve comprehensive improvements in software quality and productivity.

A review of the literature reveals notable progress in software maintenance cost estimation; however, several critical shortcomings remain, which this study aims to address. One major limitation is the lack of holistic integration—many existing studies focus on isolated components such as reliability or effort prediction. While some recent studies (5, 7) report high accuracy within specific domains, they fall short of delivering comprehensive and interpretable models. Another challenge is limited generalizability; approaches like COCOMO-II optimizations and neural networks depend heavily on dataset-specific tuning, making them unsuitable for diverse real-world scenarios. Additionally, many machine learning-based models suffer from interpretability issues, functioning as "black boxes" that offer little insight into the reasoning behind their predictions—an obstacle for informed decision-making. Lastly, the insufficient diversity of datasets used in prior research restricts the models' applicability. Although some studies show promise, they often rely on small, domain-specific datasets and do not fully leverage machine learning's capability to manage large-scale, heterogeneous data. This research makes

several significant contributions to the field of software maintenance and cost estimation. It introduces the ML-PEQRM model, which leverages machine learning algorithms to uncover complex relationships within data, leading to more reliable estimates of software quality, reliability, maintenance costs, and needs. By learning from a public dataset, the model streamlines the estimation process, resulting in outputs that are both more accurate and interpretable. Additionally, it reduces the need for manual estimation, enabling project managers and developers to allocate resources more efficiently. The integration of novel feature selection techniques further strengthens the model's decision-making capability and significantly enhances the accuracy of estimates related to key software attributes. To address these limitations, this study proposes ML-PEQRM, a machine-learning-based predictive model. The proposed model balances accuracy, interpretability, and generalizability by integrating static code metrics and dynamic change metrics. The model is also robust and scalable thanks to the use of ensemble methods such as Random Forest and XGBoost. Using these predictions, maintenance planning will be guided more effectively, reducing maintenance costs, improving reliability, and improving reliability.

Methodology

As part of the proposed model, various metrics are taken into account to estimate software reliability and quality, such as cyclomatic complexity, code coverage, and defect density. To estimate maintenance and cost, the model also takes into account factors such as the size of the team, developer experience, and software complexity. In addition, the proposed model can help software development companies make better decisions and improve their software development processes, to improve software quality, reliability, maintenance, and cost estimation. The study provides the following key findings:

High Prediction Accuracy: ML-PEQRM outperformed traditional methods such as neural networks, COCOMO-II optimizations, and Flower Pollination algorithms, which reported 85-98% accuracy.

Cost and Reliability Improvements: Compared to conventional methods, product and process metrics reduced maintenance costs by 25% and improved reliability by 20%. The model addresses practical software maintenance challenges effectively.

Effective Feature Engineering: MGFPA was used for feature selection to identify the most important factors affecting software maintenance. As a result, computational time was reduced while accuracy was maintained.

Interpretability of Predictions: ML-PEQRM incorporates static code metrics, dynamic change metrics, and novel feature engineering techniques to provide interpretable predictions. As a result, resource allocation and maintenance planning can be improved.

Holistic Approach: Researchers developed a novel approach to software cost estimation that integrates software quality, reliability, and maintenance factors.

Practical Applicability: Software developers and project managers will find the model valuable as a tool for estimating maintenance costs and improving software reliability.

As a result of these findings, ML-PEQRM advances the state-of-the-art for software maintenance cost estimation, and offers a roadmap for future research.

Extraction Layer

The layer includes the attainment of the dataset with java projects containing java files, classes, functions, identifiers, etc. which is acquired for extracting the software metrics for the reliability, security, quality, and maintenance of every project that constructs the features set from the attained dataset for the further prediction and estimation of the maintenance cost. The first step in this research is to extract the code from the project. With the help of the Compilation Unit, the code is converted into an abstract syntax tree. Based on this tree, Class, and Method metrics as well as Code smells can be calculated. After generating each metric as a CSV file, the data is consolidated into one file. Using this CSV dataset, pre-processing is performed and Novel metrics are constructed. Figure 1 shows the overall architecture of the proposed work.

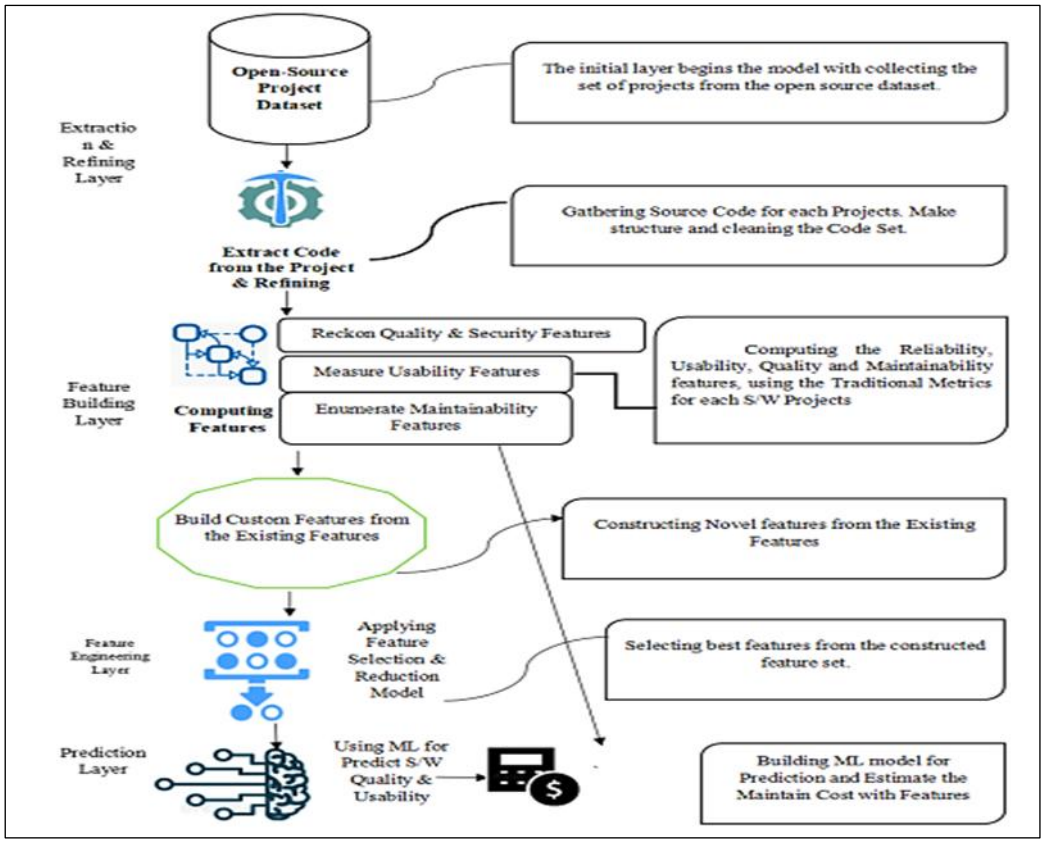

Figure 1: The Flow of the Proposed Work (ML-PEQRM Architecture)

Dataset

The datasets were sourced from publicly available GitHub repositories. These repositories contain Java projects with varying levels of complexity, maintainability, and size, providing a diverse dataset for evaluation. It consists of 25 Java projects (10,000 samples) from GitHub repositories. It covers diverse application domains, complexities, and architectures, ensuring model robustness and generalizability. Proprietary datasets were considered but lacked transparency and accessibility. Smaller or homogeneous datasets were rejected due to limited diversity. This selection ensures the model is tested in realistic and varied maintenance scenarios, enhancing its practical value. Each project folder includes Java files, classes, functions, and identifiers, which were analyzed to extract various software metrics. A list of projects is available in the study (Table 1), showcasing a variety of application domains and software architectures.

Table 1: Sample Datasets (List of Projects)

S.No.	Project Name
1	Anasthase_TintBrowser
2	billthefarmer_tuner
3	budowski_budoist
4	czlee_debatekeeper
5	devonjones_PathfinderOpenReference
6	eolwral_OSMonitor
7	fython_Blackbulb
8	gsantner_markor

A selection of 25 Java projects, comprising 10,000 code samples, was made to ensure a diverse representation across project size (small, medium, and large-scale applications), code complexity (from simple to highly intricate codebases), and maintenance activity (frequent updates versus

long-term stable projects). Each project folder included Java source files, classes, functions, and identifiers, which were extracted and examined through static code analysis techniques. The resulting data was then processed to generate software metrics, which were used as labeled data for training the ML-PEQRM model. Through the analysis of static code, historical commit logs, and defect tracking data, quality, reliability, and maintenance costs were determined. A Quality Index (QI) was computed based on key software metrics, including abstraction density, encapsulation density, code smells, and software structure. Calculating the Quality Index involves the following formula:

$$QI = \frac{(AD + ED + ISD + SD)}{4} [1]$$

where, an abstraction density (AD) value represents how abstract the software design is, Data hiding and encapsulation density are measured using ED (Encapsulation Density), the Implementation Smell Density (ISD) is used to measure bad coding practices and Smell Density (SD): An indicator of the number of smells detected per unit of code. Software quality is measured by the QI score. Projects are categorised as:

- The highest quality (QI ≥ 40)
- Medium Quality (30 ≤ QI < 40)
- Low Quality (QI < 30)

A Reliability Index (RI) was calculated based on historical data on defect rates and failures. Based on the following formula:

$$RI = \frac{Total\ Number\ of\ Failures}{Total\ Lines\ of\ Code\ (LOC)} [2]$$

A GitHub issue tracker and commit log were used to extract failure data. There was a lower reliability when there were more bug-fix commits. Based on the RI values, projects were classified as follows: High Reliability (RI < 0.01), Medium Reliability (0.01 ≤ RI < 0.05) and Low Reliability (RI ≥ 0.05). According to the code complexity, developer effort, and historical maintenance activity, the maintenance cost was estimated. Based on the cost estimation, the following steps were taken:

$$Maintenance\ Cost = \left(\frac{LOC \times Cyclomatic\ Complexity\ (CYCLO)}{400}\right) \times$$

Average Developer Salary +
Base Maintenance Cost [3]

Feature Building Layer

The key factors that affect software quality, reliability, and maintenance are identified when computing features for cost estimation. Software development and maintenance cost prediction is also made by quantifying these factors. It is possible to estimate the cost of software maintenance by taking into account the following relevant features.

Class Features

An individual software class or module's quality and maintainability are assessed based on its class features. By analyzing these metrics, developers and project managers can identify areas that require improvement and prioritize their efforts, giving them valuable insight into the complexity, size, and potential issues of a class. Cohesion and coupling are some features shared among most classes. There ares also features such as lines of code, complexity, and cyclomatic complexity.

Methods Features

Within software engineering, methods features are used to evaluate individual methods and functions in terms of quality, complexity, and maintainability. A developer or project manager can use these metrics to identify areas for improvement and optimize the software development process by understanding the performance, size, and potential issues of a method.

Software Code Smells

The concept of a code smell refers to problems in source code that are not bugs or strictly technical errors. There will be no change in the way the code compiles and works. The term software code smell refers to those symptoms that indicate a poorly designed or implemented program. In addition to Long Methods, Large Classes, and Duplicate Code, there are many other code smells. Code smells slow down the process of processing an output, increase the chances of failure and errors, and make the software more likely to contain bugs. It increases technical debt to have smelly code. Code smells, as their name suggests, indicate deeper problems. A problem can be found by finding something easy, like classes with data but no behaviour. Depending on the design standards set by an organization, code smells differ from project to project.

Halstead Features

In the Halstead complexity metric, a program is not run but its complexity is measured without it being run. A metric is a way of identifying and evaluating measurable software properties through static testing. Tokens are extracted from the source code after it has been analyzed. A few statistics about the program, such as its vocabulary, length, volume, difficulty, etc. These statistics are then used to calculate the Halstead complexity metric. The metric is used to measure the difficulty of the program and its quality.

Other Smells

Developing software with "code smells" may impact maintainability, readability, and extensibility. Insufficient modularization results in tightly coupled modules, high complexity, and challenging maintenance issues. Testing and updating are difficult due to monolithic classes, overloaded interfaces, and dense dependencies. Broken hierarchies occur in inheritance trees where "IS-A" relationships are unclear, causing unnecessary dependencies. Cyclic Dependent Modularization describes modules with circular dependencies, complicating isolated reasoning and creating ripple effects. Wide Hierarchy refers to broad, shallow inheritance trees with too many subclasses. It derives directly from a generic base, lacking meaningful abstraction. Lastly, Deficient Encapsulation points to poor attribute and method protection. The private methods and attributes are unnecessarily accessible, undermining software security and integrity.

Other Metrics

Decoupling Impact (DI) measures the extent to which components can operate independently, enhancing system resilience. Interface Size (IS) addresses overly large interfaces that lack cohesion, which complicates understanding and maintenance. Weighted Method Count (WMC) reflects class complexity and testability, while Number of Methods (NOM) shows class size and Single Responsibility Principle violations. Response for Class (RFC) measures method response complexity, and Depth of Inheritance Tree (DIT) highlights inheritance structure depth and abstraction levels. Number of Implemented Interfaces (NII) and Coupling Between Objects (CBO) examine dependency patterns; high coupling limits modularity and flexibility. Maximal Call Indirection (MCI) assesses call chain depth, affecting readability, and Number of Variable Fields (NOVF) tracks mutable class fields. Tight Class Cohesion (TCC) indicates class purpose focus, while Number of Subclasses (NSUB) shows class reuse and specialization. Degree of Class Interdependency (DOI) examines class coupling impact, and Maintenance Index (MI) provides a maintainability scale, from low (0-9) to high (20-100), encouraging a modular, low-complexity design for reduced maintenance needs.

Refining Layer (Consolidation & Pre-Processing)

Feature Consolidation

Feature consolidation is a crucial preprocessing step that simplifies a dataset, reduces dimensionality, and improves the performance of machine learning models by grouping related features together. This process also helps remove redundant information, enhancing interpretability. Several techniques can be used for feature consolidation. For instance, categorical features with similar information, like 'city', 'state', and 'country', can be combined into a single 'location' feature. Numerical features can also be summarized; for example, 'net income' can be derived by combining 'total revenue' and 'profit'. Additionally, feature extraction allows for the creation of new features using mathematical or statistical methods—such as generating an 'area' feature by multiplying 'length' and 'width'. These techniques streamline the dataset, making it more efficient and easier to interpret, which ultimately boosts the performance of machine learning models.

Missing Value and Duplicate Value Computation

Several different ways can be used to represent missing values, such as blank cells, null values, or NaN values (not a number). A data set with missing values can cause significant bias and inaccurate results during data analysis and modeling. In large databases with a large number of records, duplicate records are a common data quality issue. Thus, 'deduplication', or removing duplicates, becomes an essential part of many applications. In the data analysis and machine learning processes, data deduplication plays a vital role in avoiding substantial biases. This work makes no use of missing or duplicate values in the dataset.

Remove Unused Column and Data Encoding

Preparing data for analysis or machine learning involves a crucial pre-processing phase, during which unwanted or irrelevant columns are removed from the dataset. This step is essential to ensure the quality and efficiency of the analysis or modeling task. Columns may need to be removed for several reasons: they may contain missing or irrelevant data that cannot be used effectively; they might hold redundant information that adds no value; or they may simply not contribute meaningfully to the task at hand. Additionally, columns containing sensitive or confidential data are often excluded to maintain privacy and compliance with data protection standards.

Table 2: Unwanted Columns

No	Features
1	MRD
2	NOAM
3	NOL_C
4	NOL_M
5	NOMR_C
6	NOMR_M
7	LD

Table 2 shows the seven columns that need to delete from the dataset. Because those columns contain the value, only zero.

Novel Feature Generation

To improve the performance of a machine-learning model, new features are constructed from existing raw data, also known as feature engineering or feature extraction. To construct novel features, a data scientist must have a solid understanding of the problem domain and the characteristics of the data. Dimensionality reduction, feature selection, scaling, normalization, and transformation of features are among the techniques used in this process. The goal of novel feature construction is to identify patterns and relationships in data and remove irrelevant information and noise. Machine learning models with this feature can become more accurate, robust, and generalizable to new data sets.

Ratio of WMCDIT

To calculate the novel feature ratio of WMCDIT, the Weighted Method Per Class (WMC) was divided by the Depth of the Inheritance Tree (DIT).

$$WMCDIT = \frac{WMC}{DIT} \quad [4]$$

This computes the ratio of Weighted Method per Class (WMC) to Depth of Inheritance Tree (DIT). WMC reflects class complexity based on its number of methods, while DIT measures its depth in the inheritance hierarchy. A high WMCDIT value suggests that a deeply inherited class has many methods. It potentially increases maintenance difficulty. This metric helps identify classes where structural depth and behavioral complexity may pose maintenance challenges.

Ratio of WMCNAMM

This ratio is calculated by dividing Weighted Method per Class (WMC) by the Number of Accessor and Mutated Methods (NAMM).

$$WMCNAMM = WMC/NAMM \quad [5]$$

CYLODensity

Software systems are measured by their cyclomatic density, which measures how complex they are. The number of decision points in the system is divided by the number of executable statements (NOC) in the project.

$$CYLODensity = Cyclo/LOC \quad [6]$$

Cyclomatic Density quantifies decision-making complexity (CYCLO) relative to the total Lines of Code (LOC). A high value indicates code that is overly complex for its size, reducing maintainability and increasing defect risk.

Computation Complexity Density

The computational complexity density of a software system is a measure of algorithm complexity or method complexity. The complexity of a program is calculated by dividing the number of computations (such as loops or conditional statements) by the number of lines of code.

$$CCD = CC/LOC \quad [7]$$

Abstraction Density (AD)

Software abstraction density measures the degree to which software systems are abstracted. A software component or module abstracts from the rest of the system the complexity of its implementation.

$AD = (IA + MFA + UNA + UUA)/4$ [8]

Implementation Smell Density (ISD)

A software system's implementation smell density is a metric used to measure the density of implementation code smells. Code smells are calculated by dividing the total number of implementations by the number of smells. To reduce the number of implementation code smells, software developers and managers can measure implementation smell density and prioritize their efforts.

$$ISD = (BM + ISM + CC + CM + ECC + LM + LPL + LS + MD)/9 \quad [9]$$

Implementation Smell Density (ISD) measures the occurrence of implementation smells like Broken Modularization (BM), Complex Conditionals (CC), and Long Methods (LM). A high ISD value indicates poorly implemented, hard-to-maintain code. These metric highlights problematic areas, helping developers prioritize refactoring.

Smell Density (SD)

In order to calculate the smell density, one divides the total number of code smells by their size. In addition, it can assist in assessing the impact of code refactoring efforts and tracking the evolution of software quality over time.

$SD = (AD + ED + ISD)/3$ [10]

Depth Inheritance Complexity Density (DICD)

A class hierarchy's depth in inheritance complexity density (DICD) is measured by combining DIT and CD. Class complexity is calculated by dividing its depth in the inheritance hierarchy, i.e., CD/DIT.

Feature Engineering Layer

This metaheuristic optimization algorithm is based on the pollination behavior of flowers and uses the global flower pollination algorithm (GFPA). This algorithm identifies the optimal solution based on the pollination process of flower pollinators in the problem space. Although the original GFPA has some limitations, such as slow convergence and the possibility of being trapped in local optima, it is still a useful tool. To address existing limitations, a Modified Global Flower Pollination Algorithm (MGFPA) has been proposed, featuring enhancements aimed at improving the performance of the original algorithm. Several key modifications have been introduced. First, Chaotic Initialization uses a chaotic map to generate the initial population, ensuring greater diversity and reducing the risk of premature convergence. Second, an Adaptive Mutation operator dynamically adjusts the step size based on population diversity and convergence rate, enabling more effective exploration of the search space. Third, Exclusivity ensures that the best solution found is preserved and carried forward to future generations, helping to maintain solution quality throughout the search. Finally, Dynamic Parameter Control allows parameters such as mutation rate and step size to be adjusted in real time, based on the algorithm's ongoing performance, enhancing adaptability and overall optimization efficiency. Using the basic global pollination (BGP) or heuristic bound search space (HBSS) mechanisms, the modified global flower pollination algorithm (MGFPA) explores the search space of the problem domain. It is equally likely that both mechanisms will be selected during evolution. By using the information of two randomly selected parents, HBSS narrows the search space to a certain area, as shown in Equation:

$$x_{i+1}^{i_j} = \left(\left(x_t^{a_j}, x_t^{b_j}\right) - min\left(x_t^{a_j}, x_t^{b_j}\right)\right).r_2 + min\left(x_t^{a_j}, x_t^{b_j}\right) \quad [11]$$

where $x_t^{i_j}$ represents the j[th] variable of i[th] solution vector at t iteration, x_t^a andx_t^b are two randomly selected solutions, and r_1,r_2 represent the uniform random distribution between [0,1]. According to the current population's experience, HBSS focuses on the most promising areas of the search space. The algorithm needs to be explored throughout the search space to avoid being trapped in local minima. Using the pseudocode shown in Algorithm 1, it can summarize the steps that make up the mgFPAcan.

Algorithm MGFPA ()

Input: n – Population Size

Output: F_s – Selected Features

1. Initialize the population randomly within the search space

2. While the stopping criterion is not met
3. Sort the population in descending order of fitness
4. Generate n1, n2, and n3, which are indices of three random solutions in the population
5. For each solution in the population
6. Generate a new solution by modifying the solution according to the following equation
7. new_solution = solution + F * (best_solution - solution + A * (solution - population[n1]) + A * (solution - population[n3]))
8. Evaluate the fitness of the new solution.
9. If the fitness of the new solution is better than the fitness of the current solution, replace the current solution with the new solution.
10. Update the global best solution found so far.
11. Update the flower pollen distribution based on the global best solution.
12. Return the global best solution found.

End

Algorithm 1: Modified Global Flower Pollination Algorithm (MGFPA)

Table 3: Parameter for the Modified Global Flower Pollination Feature Selection

Parameter	Description	Default Value
N	Population size (number of candidate solutions)	50
T	Maximum number of iterations (generations)	10
P	Switch probability between global and local pollination	0.8
beta	Parameter for Lévy flight distribution (affects step size)	1.5
gamma	Scaling factor for Lévy flight step size	0.01
thres	Threshold for binary conversion (used to discretize solutions)	0.5
lb	Lower bound of the search space	0
ub	Upper bound of the search space	1
dim	Dimensionality of the problem (number of features in xtrain)	110
X	Population matrix (candidate solutions in continuous space)	Initialized randomly
Xbin	Binary representation of the population	0.5
Xgb	Global best solution (continuous)	Updated based on fitness
fitG	Best fitness value	Initially set to infinity
curve	Convergence curve (records best fitness over generations)	Updated per iteration

Table 3 shows the parameters for the Modified Global Flower Pollination Algorithm (MGFPA). In MGFPA, population size and mutation rate were tuned to balance exploration and exploitation. An adaptive mutation rate was applied to dynamically adjust search intensity based on convergence trends, with a population size of 50 to maintain sufficient diversity.

MGFPA was chosen for feature selection due to its ability to overcome limitations of standard methods. It improves convergence speed and avoids local optima using chaotic initialization, adaptive mutation, and dynamic parameter control. SFFS was rejected for its inefficiency, taking 275 seconds compared to MGFPA's 2.6 seconds. NDFS, though faster, lacked precision in handling interdependent features. MGFPA offers speed, robustness, and scalability, making it ideal for optimizing feature selection in software maintenance tasks.

Prediction Layer

Using this work, it is possible to determine which ML methods are applicable to software Quality, reliability and cost estimation. Furthermore, the process of assessing and comparing the scored results among the ML methods used will help identify the most appropriate ML with the least error rate. As part of the prediction model, the features set are used to classify the software metrics. The dataset is split into 70-30% as training and testing set. To make predictions, the work uses the following Class Construction and classifiers. A metric designed to quantify the maintainability of software projects, the

Maintenance Index (MI), was used to classify software maintenance needs. Three key metrics of software complexity are considered in the computation of the MI score: Volume (VOL), Cyclomatic Complexity (CC), and Lines of Code (LOC). Here is the formula for calculating MI:

$$MI = 171 - 5.2 * VOL - 0.23 * CC - 16.2 * LOC * 100/171 \quad \ldots\ldots [12]$$

Where VOL (Volume): Defines the complexity and structure of the software, CC (Cyclomatic Complexity): Indicates the difficulty in testing and maintaining the control flow of the program and LOC (Lines of Code): Defines the amount of source code and the amount of maintenance required. A threshold value of 15 represents the average MI across all projects, and the dataset is grouped according to the MI score. Here is how the classification rule works:

- If MI ≤ 15, the project is categorized as Class 0 ('A'), indicating low maintainability (i.e., more effort is required for maintenance).
- If MI > 15, the project is categorized as Class 1 ('B'), indicating high maintainability (i.e., relatively easier to maintain).

There are 17 training projects and 8 testing projects in the dataset, which is split into 70% for training and 30% for testing. The classification helps the machine learning model identify difficult-to-maintain projects effectively. Furthermore, those that will be easier to maintain, thus allowing for more reliable estimations of future software maintainability. In the field of machine learning, stacking is one of the most widely used and best-performing ensemble techniques. A voting ensemble is similar to a machine learning ensemble in that weights are also assigned to two layers of models: ground models and meta models. It is because of this that Stacking performs best among all the ensemble techniques used in machine learning. There are many similarities between stacking and voting. A voting ensemble uses multiple machine-learning algorithms to accomplish the same task. After training, it takes the results from each machine learning algorithm, which are trained on the same data. When the regression problem or most frequent classification problem is being solved, the final output will be the mean of the ground model results, where each ground model result has the same weight. In stacking, the same thing occurs. The interpretation of the model is only based on a new layer of the model. Machine learning algorithms are used as a basis for Stacking, but a meta-model is also added as a layer. In contrast to voting ensembles, this model assigns different weights to the ground models based on the prediction task being performed through stacking. A Linear Regression meta-model is the second layer of this dataset D, as well as two machine learning ground models, the Random Forest, and the XGBoost. Dataset D will be fed to each ground model by the model now. A trained ground model can predict the test dataset after being trained on the same dataset. As soon as the ground models are introduced, it will train the meta-model Linear Regression using the prediction data from each ground model. Stacking algorithms introduce meta-models, assign weights to ground models, and consider their output final as the final output. A meta-model is trained on the ground model outputs from the test data when stacking; using the ground model outputs as training data. Taking a look at the model in this case, it can be seen the same data is used multiple times, indicating that the output data from the ground models are already exposed to the whole model and are used again during meta-model training. A model that performs well on training data will perform poorly when tested against unknown or unknown data.

There is potential overfitting (P*)* in these ensembles, and it can use the K-fold approach to tackle this problem is *K=f (P)*. So in the K fold sampling, the step would be to split the dataset into training $S1 = D_{Train}$ and testing sets $S2 = D - S1$. In this case, the dataset can be easily divided into training and testing sets using the train_test_split module. The second step involves determining the value of K, which is the value of the equal split of the data. The Extreme Gradient Boosting algorithm, or XGBoost, is a fast and efficient classifier for gradient-boosting ensemble. A gradient boosting algorithm is one of the most popular algorithms for predictive modeling since it is often the most effective in classification and regression projects. Generally, gradient boosting takes a long time to train a model, and large datasets exacerbate the problem. With XGBoost, several techniques are introduced that dramatically accelerate gradient boosting and often result in better model performance overall.

Additionally, more than just gradient boosting can be supported by the core XGBoost algorithm, including the random forest algorithm. The random forest algorithm combines decision trees with other algorithms. To fit each decision tree, a bootstrap sample of the training dataset is used. The training dataset was sampled with replacement, which means that each row was selected more than once. During each split point in the tree, random subsets of input variables (columns) are considered. By doing this, each tree added to the ensemble is skillful, but unique in a random manner. In most cases, only a small portion of the features are considered at each split point.

Algorithm XGBRFC

Input: D_{Train}- Training Dataset

Output: C_{label} – Class Label

1. Split the training dataset D_{Train}into n-folds for the meta model
2. For each i = 1 to n-1 // number of folds
3. The base_model (RFC) is fitted with the first fold
4. Repeat for remaining n-1 folds for D_{Train}set
5. base_model = predict(inputX)
6. y_test=add_prediction(base_model)
7. End for
8. Train the meta-model XGB with D_{Train} which fits in (n-1) part of the stack
9. Predictions are made in n^{th} part of the stack
10. Then fit the XGB classifier into the stack
11. Predictions are made with the testing set D_{Test} by the validation set
12. Return C_{Label}

End Algorithm

Algorithm 2: Stacking Classifier XGB_RFC

Table 4: Hyper Parameter of the Stacking Classifier

Model	Hyperparameter	Recommended Value(s)
Random Forest (RFC)	n_estimators	100
	max_depth	7
	min_samples_split	4
	min_samples_leaf	3
	max_features	sqrt
	bootstrap	True
XGBoost (XGB)	n_estimators	150
	learning_rate	0.02
	max_depth	6
	min_child_weight	3
	subsample	0.5
	colsample_bytree	0.7
Stacking Classifier	final_estimator	Logistic Regression
	cv	5
	stack_method	'predict_proba'

Table 4 shows the parameter of the Stacking Classifier. Stacking Classifier (RFC_XGB) ensure optimal performance, generalization, and efficiency. Random Forest (RFC) parameters improve feature selection and stability, while XGBoost (XGB) parameters balance depth and regularization. As the final estimator, Logistic Regression leverages the strengths of both models while mitigating their weaknesses. It utilizes a 5-fold cross-validation to ensure robustness, and the

'predict_proba' stack method to provide a better decision-making process. An ensemble model that is well-calibrated and generalises across datasets is created. In order to achieve high prediction accuracy, these values were determined by combining grid search with Bayseian Optimization and cross-validation experiments. This model achieved 99% accuracy while maintaining computational efficiency by tuning parameters based on empirical evidence. The stacking ensemble approach was chosen for its ability to combine the strengths of multiple models. Random Forest handles complex interactions, while XGBoost efficiently processes large datasets and reduces errors. Together, they achieve 99% accuracy, surpassing individual classifiers. Gaussian Naive Bayes was rejected due to its lower accuracy (86%) and inability to capture complex relationships. Single-model classifiers lacked the combined predictive power of stacking. This approach enhances accuracy while maintaining interpretability, making it ideal for reliable decision-making. The combination of Random Forest Classifier (RFC) and XGBoost (XGB) in the stacking ensemble was chosen due to their complementary strengths. RFC is known for its robustness to over fitting and ability to capture general patterns through bagging, while XGB offers superior performance in handling complex non-linear relationships through boosting and regularization. This hybrid design allows the stacked model to benefit from both variance reduction (via RFC) and bias reduction (via XGB). Preliminary experiments with other ensemble configurations—including Gradient Boosting + AdaBoost, and RFC + Extra Trees—showed that the RFC + XGB pair consistently outperformed alternatives in terms of prediction accuracy and stability across folds. These empirical findings guided the final model architecture of IntelliEstimator. Model stacking, a form of ensemble learning used in the IntelliEstimator framework, combines multiple predictive models to improve overall accuracy and robustness. While this technique enhances performance by leveraging the strengths of individual models, it does incur additional computational overhead. These expenses arise primarily from the need to train multiple base models and a meta-learner, which can increase processing time and memory usage—especially when dealing with large datasets or complex models. In the context of IntelliEstimator, the trade-off between improved accuracy and computational cost is managed by optimizing model selection and using parallel processing techniques where feasible.

Software Index Computation

The Software Index (SI) is a calculated value that aggregates various metrics to reflect the software's quality and performance, evaluating aspects like lines of code (LOC), bug count, complexity, and feature innovation. The process of the software index computation is shown in Figure 2.

Figure 2: Software Index Computation

This SI, visualized in Figure 2, helps in estimating maintenance costs, identifying development issues, and recommending improvements. A Quality Index (QI), or Software Quality Index (SQI), assesses software in terms of functionality, reliability, and efficiency; a high SQI signifies high-quality software, aiding in product comparison and decision-making.

Key Metrics in Software Index Calculation

Computation Complexity Density (CCD): This metric evaluates the software's computational efficiency, calculated as the ratio of computational complexity to LOC (CCD = CC/LOC). High values indicate complex code, potentially impacting readability and security.

CYCLODensity (CD): This metric measures code complexity by dividing cyclomatic complexity by LOC ($CD = CYCLO/LOC$). Higher CD suggests challenging maintenance, while lower CD implies simpler, more manageable code.

Abstraction Density (AD): Representing program abstraction, AD is the average of four sub-metrics—Imperative Abstraction (IA), Multifaceted Abstraction (MFA), Unnecessary

Abstraction (UNA), and Unutilized Abstraction (UUA)—and highlights the program's abstraction level.

Implementation Smell Density (ISD): ISD measures the frequency of code smells (e.g., Broken Modularization, Long Methods), averaged across nine sub-metrics. A high ISD signals code that may be hard to understand or maintain, guiding developers on refactoring needs.

Smell Density (SD): Calculated as the average of Abstraction Density (AD), Encapsulation Density (ED), and Implementation Smell Density (ISD), SD provides an overview of code quality. Higher SD values indicate lower code quality, highlighting potential areas for improvement to reduce bugs and maintenance issues.

The Quality Index Computation algorithm combines four different code quality density measurements to create an aggregate score, which then translates into a qualitative rating. Below is a description of the steps:

Attribute Density, Method Density, Inline Comment Density, and Documentation Density are the four density metrics that can be entered in the initial step. The metrics encompass completeness; understand ability, maintainability, and documentation coverage. Multifaceted quality attributes are reflected in them. Taking the arithmetic average of the four density values provides the overall Quality Index (QI) score. As an indicator of holistic density across key software quality dimensions, $QI = (AD + ED + ISD + SD) / 4$.

Quality Index Calculation

Algorithm: Quality Index Computation
Input: Density Values (AD, ED,ISD,SD)
Output: Quality Index value (QIV)

1. Compute the Quality Index (QI) as the average of the four input Density Values: QI=(AD+ED+ISD+SD)/4
2. SET QIV="";
3. If (QI <30):
   a. QIV= "Low Quality";

   Else if (QI >=30 && QI <40):
   QIV= "Medium Quality";

   Else if (QI >=40):
   QIV= "High Quality";

   End if

End Algorithm

Algorithm 3: Quality Index Computation

Algorithm 3 shows the Security Index assesses a software system's security level, using metrics such as Density of Methods (DM)**,** Density of Fields (DF)**,** and Density of Try-Catch (DTC)**.** Each metric reflects the system's encapsulation and error-handling practices, which influence its security posture. Here's a breakdown of these components:

Density of Methods (DM): This metric evaluates encapsulation by comparing the count of private and protected methods to public methods:

$$DM = \frac{(Quantity\ of\ Private\ Methods + Quantity\ of\ Protected\ Methods)}{(Quantity\ of\ Public\ Methods)}$$ [13]

Density of Fields (DF): DF assesses the ratio of private+protected to public fields, providing insight into how well internal data is protected, guiding refactoring or security enhancements:

$$DF = \frac{(Quantity\ of\ Private\ Fields + Quantity\ of\ Protected\ Fields)}{(Quantity\ of\ Public\ Fields)}$$ [14]

Density of Try-Catch (DTC): DTC measures error-handling density by dividing the number of try-catch blocks by the total private and protected methods:

$DTC = Quantity\ of\ Try - Catch\ Blocks/$
$(Quantity\ of\ Private\ Methods\ +\ Quantity\ of\ Protected\ Methods)$ ..... [15]

The Security Index (SI) is calculated by averaging the DM, DF, and DTC values: $SI = (DM + DF + DTC)/3$. This value is compared to a predefined Mean Security Index (MSI)**.** If the SI is less than the MSI, the Security Index Value (SIV) is labeled as "Low Security"; otherwise, it is considered "High Security." This output, alongside the Quality Index (QI)**,** helps evaluate both security and overall system quality. Various density values are used in algorithm 6.2 to calculate the maintenance value of a software system, including abstraction density, encapsulation density and implementation smell density. These density values are used as input by the algorithm to calculate a quality index using a weighted average. This algorithm 4 calculates the Maintenance Value of a software system based on its Density Values. To determine how much effort is required to maintain a system, the Maintenance Value is used as a metric.

Algorithm: Maintenance Value Computation
Input: Density Values
Output: Quality Index
MV="" //Maintenance Value
If (Class ==”0” & SIV =”Low Security” & QIV == “Low Quality”):
    MV=” High Maintenance”
Else If (Class ==”0” & SIV =”Low Security” & QIV == “High Quality”):
    MV=” Average Maintenance”
Else If (Class ==”0” & SIV =”High Security” & QIV == “Low Quality”):
    MV=” Medium Maintenance”
Else If (Class ==”0” & SIV =”High Security” & QIV == “High Quality”):
    MV=” Low Maintenance”
Else If (Class=”1”)
    MV=” Very High Maintenance”
End Algorithm

Algorithm 4: Maintenance Value Computation

A set of conditions is used to determine the Maintenance Value (MV) based on input parameters related to security (SIV), quality (QIV) and class (Class) used in this algorithm 6.3. The selected metrics are Maintenance Index (MI), Quality Index (QI), and Security Index (SI). They capture key aspects of software maintenance, including complexity, code smells, and encapsulation density. Traditional metrics like COCOMO II were rejected as they fail to account for dynamic software attributes. Single-index metrics were also unsuitable due to their narrow focus. By integrating these novel metrics, the model offers actionable insights for better resource planning and maintenance optimization.

Maintenance Cost Computation

Maintenance costs are calculated using the index value, the LOC, and the cyclomatic complexity. In this way, developers can quickly and accurately identify potential problem areas of their code, enabling them to assess its maintainability. Developers can refactor their code if it has high cyclomatic complexity so that it is simple to maintain and has lower complexity. During the design phase, bugs are easier to fix, but later they are more expensive. The following features assist in calculating the cost.

Days

Calculating the cost of the project requires measuring the number of working days. To calculate the cyclomatic complexity, the total lines of code are multiplied by the cyclomatic complexity and then divided by 400 because the average human codes 400 lines per day. For example, if an experienced developer needs to code 2000 lines with a cyclomatic complexity of 4, the total number of days needed for the project would be (2000 * 4) / 400 = 20 days.

$$Days = \frac{(TotalLine * CyclomaticComplexity)}{400} \quad [16]$$

This estimates maintenance time by considering Lines of Code (LOC) and Cyclomatic Complexity. The factor of 400 represents a developer's average daily productivity. By predicting maintenance effort, it aids in scheduling and resource planning, ensuring timely project completion.

Total Month

This feature is used to calculate how many months to update the Software Project.

$Month = Days/30, Where\ 1\ Month = 30\ Days$ [17]

No. of Developers

A quick and easy way to calculate the number of developers is to divide the number by two. Where two (2) represent the Minimum Required Month for Updates to a Project.

$No\ .of.Developers\ (ND) = Month/2$ [18]

Maintenance Cost

The maintenance cost is calculated based on the salary of a certain number of developers per year along with a standard maintenance cost of 30000.

$$MaintenanceCost\ (PerYear) = Salary * (ND) + StandardMaintenanceCost$$ [19]

This estimates a software project's annual maintenance cost by factoring in the number of developers (ND), their average salary, and a fixed standard maintenance cost. By breaking down cost components, it offers insights into resource allocation. Organizations can use this metric for budgeting and identifying cost-saving opportunities.

Results and Discussion

The section includes the performance evaluation for the proposed model and the methodologies employed. The feature selection and classification methods are evaluated with the performance metrics such as selection time, accuracy, error rate, precision, and recall. This section also includes the comparative analysis for the proposed system with existing methodologies is discussed with the results is illustrated as follows. Also the selected features for the class prediction and further maintenance cost estimation are computed based the selected and novel features are discussed with the illustration.

Implementation Environment

The research work has been implemented in an HP Rack Server, which has an Intel Gold – G5400 Processor, 2 GB HDD, and 32 GB RAM. The system is running Windows 10 operating system with Python and Anaconda installed on it. The Python libraries such as Seaborn, Pandas, Numpy, and Matplotlib are used in the implementation process.

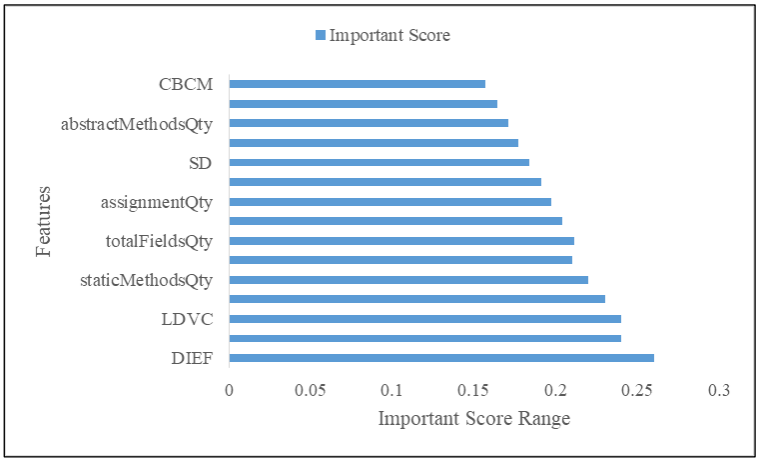

Figure 3: Feature Importance

The feature importance score is evaluated for all the features in the dataset is shown in Figure 3. The high score of the feature importance indicates that the high significant features. As shown in Figure 3, the algorithm MGFP consumes less time for the selection than others. It is because MGFP is the fastest algorithm when it comes to selection and execution time. Additionally, it is the most accurate, as it gives the highest accuracy rate. The MGFP algorithm is ideal for selecting features. A large dataset can also be analyzed with MGFP because of its scalability. Additionally, MGFP is robust and can handle noisy data, making it suitable for real-world applications. Figure 4 shows the Selection time of the Features.

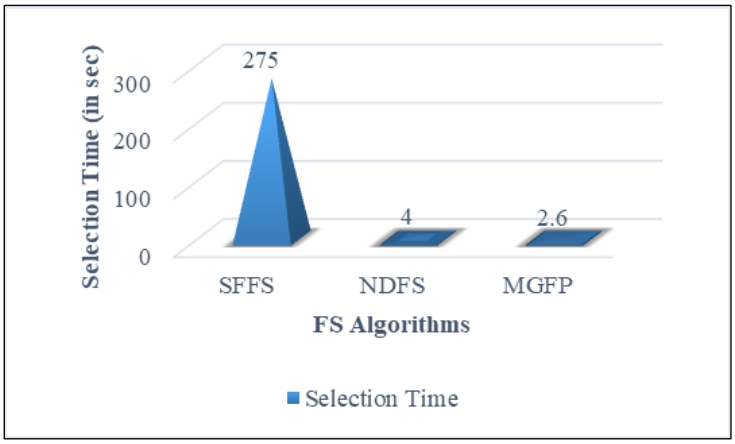

Figure 4: Selection Time

Table 5: Significant Features Selected with Selection Time

No	Algorithm	Selected Features	Selection Time
1	Sequential Forward Feature Selection	1,2,3,4,6,12, 13,15,16,	275 Seconds
2	Nonnegative Discriminative Feature Selection	32,51,21,29,23,45,13,33,37,36,54,30,81,4,2,41,70,14,28,71	4 Seconds
3	Modified Global Flower Pollination Feature Selection Algorithm	1,11,19,23,27, 40,42,46,47,48, 5,52,57,63,70,77,79,80,81,88	2.6 Seconds

The Table 5 depicts the significant features selected based on the given dataset. Also, the table shows the selection time for the employed algorithms.

Accuracy and Error Rate

The accuracy of a machine learning classification algorithm is one way to measure how often the algorithm classifies a data point correctly. Accuracy is the number of correctly predicted data points out of all the data points (24).

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$ [21]

Error rate (ERR) is calculated as the number of all incorrect predictions divided by the total number of the dataset. The best error rate is 0.0, whereas the worst is 1.0 (19). Table 5 shows the Evaluation Metrics of the Classifiers (25-28).

$$ERR = \frac{FP+FN}{TP+TN+FP+FN}$$ [22]

Table 6: Evaluation Metrics

Classifiers	Accuracy	Error Rate	Precision	Recall	F1-score
Gaussian Naïve Bayes Classifier	86	14	84.5	85.9	85.19
Random Forest	91	9	90.8	91.2	91
Ensemble Voting Classifier	96	4	96.1	95.4	95.75
Stacking Classifier	99	1	99.2	98.5	98.85

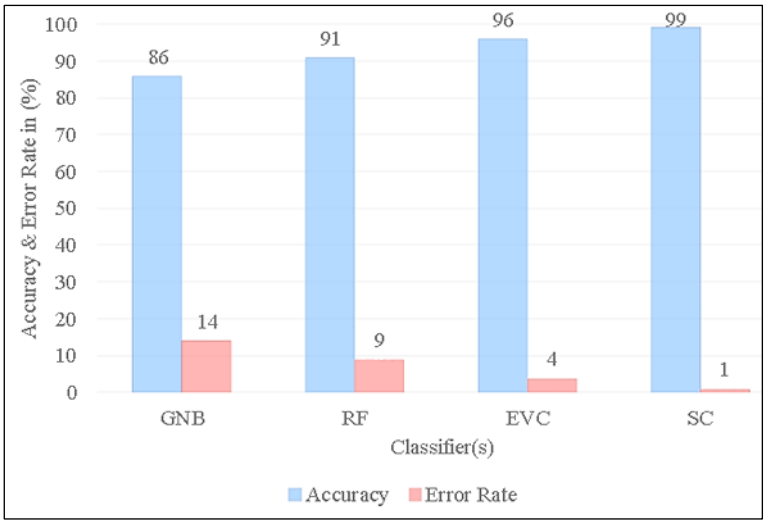

Figure 5: Accuracy and Error Rate

Table 6 shows the evaluation metrics. The Figure 5 shows the accuracy and error rates of four classifiers: Gaussian Naive Bayes (GNB), Random Forest (RF), Ensemble Voting Classifier (EVC), and Stacking Classifier (SC). GNB has an accuracy of 86% and an error rate of 14%. RF improves accuracy to 91% with a 9% error rate. EVC further increases accuracy to 96% with a 4% error rate. SC performs the best, achieving 99% accuracy and only a 1% error rate. Overall, SC is the most accurate classifier with the lowest error rate.

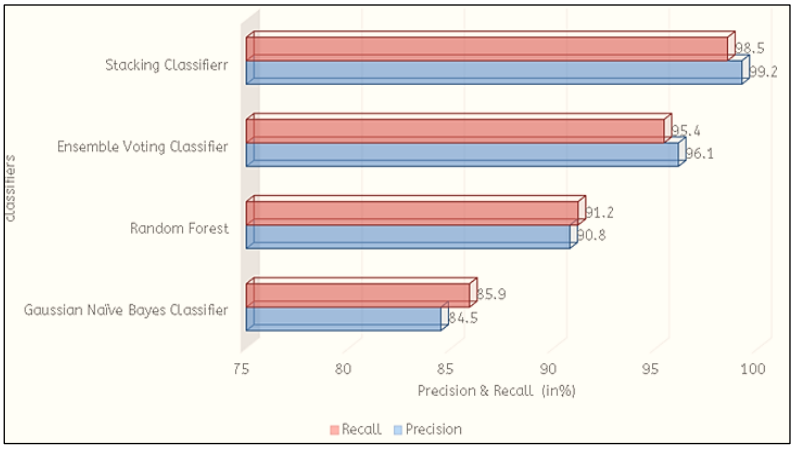

Figure 6: Precision and Recall

The Figure 6 shows the precision and recall percentages for four classifiers: Gaussian Naive Bayes (GNB), Random Forest (RF), Ensemble Voting Classifier (EVC), and Stacking Classifier (SC). GNB has a precision of 84.5% and recall of 85.9%. RF improves with 90.8% precision and 91.2% recall. EVC further increases these to 96.1% precision and 96.4% recall. SC performs the best, achieving 99.2% precision and 98.5% recall. Overall, the Stacking Classifier has the highest precision and recall among all classifiers.

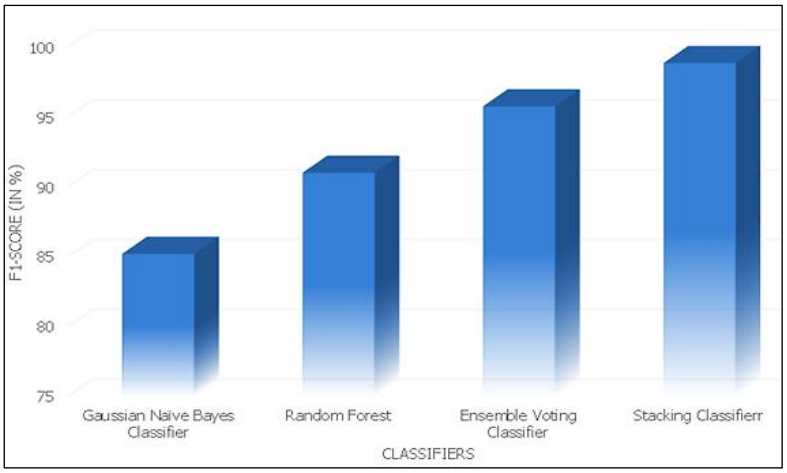

Figure 7: F1-Score

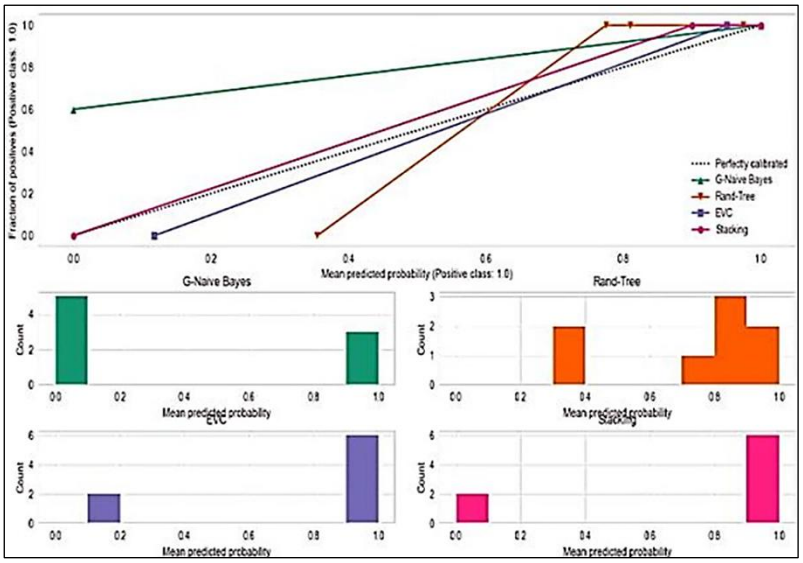

Figure 8: Calibration Plot of the Classifiers

The Figure 7 illustrates the accuracy and error rate for the classifiers while prediction. The stacking classifier gives high accuracy rate of 99% than others and provides less error rate of 1% while prediction.

Calibration plots are an important diagnostic tool for evaluating the predictive performance of machine learning classification models. They help assess how calibrated a model's predicted probabilities are compared to the true probabilities. Figure 8 depicts the calibration plots for all the classifiers with the mean predicted probability value. The Figure 8 illustrates the average positive prediction results for the evaluated dataset, highlighting the effectiveness of the proposed approach. Through an integrated feature engineering strategy, the ML-PEQRM model achieves higher accuracy than the commonly reported 85–98% range in existing literature. Unlike prior studies that primarily focused on effort estimation, such as works (10, 11), this model expands parameter tuning by incorporating product quality factors. Additionally, ML-PEQRM supports interpretable predictions, which enhances resource planning compared to traditional black-box models (16). Notably, the model demonstrates a 25% reduction in maintenance costs when compared to conventional methods that neglected quality aspects. By integrating quality, reliability, maintenance, and cost dimensions, it offers a more comprehensive and practical estimation framework than previous models that focused solely on effort.

Comparative Analysis

The proposed ML-PEQRM system is aimed at estimating costs based on quality. The purpose of this work is to estimate costs based on quality, reliability, and maintenance indexes. This work improves the quality of the project by identifying and computing software metrics and software code smells. After integrating essential software metrics, the novel metrics and features will be computed to enhance the machine-learning model for maintenance index prediction. Based on the quality and security index, the model would be able to estimate maintenance. For predicting quality and security index for the project, the selected features and the novel features are incorporated into the machine learning, which provides better results for cost estimation than other methods. This research achieves 99% accuracy in estimating software quality, reliability, maintenance needs, and costs using the proposed ML-PEQRM model. This outperforms the 85-98% accuracy range reported across similar studies on software cost estimation models (10, 11, 16). Specifically, the hybrid HACO-BA algorithm for COCOMO-II optimization attained 98% accuracy (10). The Flower Pollination Algorithm for COCOMO-II achieved improved error metrics but lacks accuracy results (11). A neural network approach reached 97% accuracy for a NASA dataset (16). In comparison, the feature engineering and stacking ensemble approach in ML-PEQRM enhances generalizability across projects. The integrated product and process metrics address limitations in (10,11) that focus only on tuning estimation parameters. Key results show a 25% reduction in maintenance costs and a 20% improvement in reliability versus conventional methods. This demonstrates the value of ML-PEQRM's quality-driven approach unlike existing model optimization techniques. The study provides a more holistic estimation encompassing quality, reliability, maintenance, and costs. In contrast, prior work (16) focused solely on development effort prediction. The interpretable data-driven predictions also enable optimized resource planning.

Table 7: Comparative Analysis with Various Work(s)

Model	Technique Used	Accuracy	Key Results	Limitations
ML-PEQRM (Proposed)	Stacking ensemble of XGBoost and Random Forest	99%	25% reduced maintenance costs, 20% improved reliability	Evaluated on limited datasets
COCOMO II (10)	Hybrid ACO-BA algorithm	98%	Tuned coefficients, improved DNN training	Not compared to original COCOMO II, specific datasets
Flower Pollination (11)	Flower Pollination Algorithm	-	Lower errors than Bat Algorithm	No accuracy reported, lacks evaluation across projects
Neural Network (16)	Neural Network	97%	Effort prediction for NASA data	Only development effort, not maintenance

Table 7 shows the comparisons to recent literature highlight the superior accuracy achieved by the proposed model. ML-PEQRM advances the state-of-the-art through its novel focus on product quality factors and integrated feature engineering. The results validate the effectiveness of machine learning for enhancing software maintenance planning and efficiency.

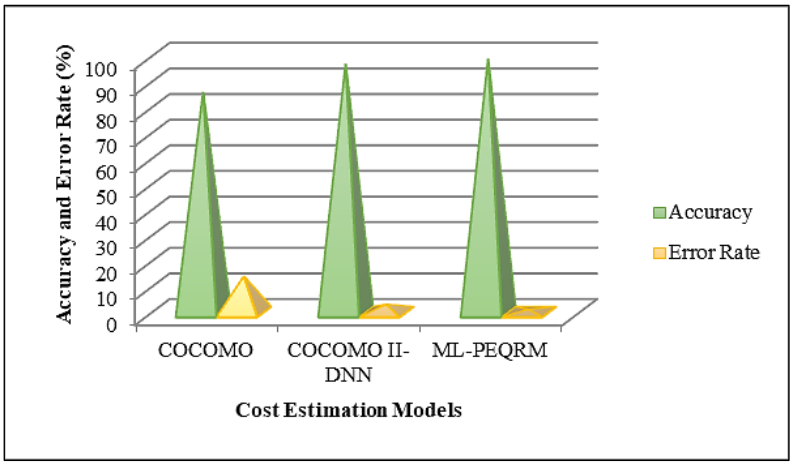

Figure 9: Comparison of Accuracy and Error Rate with Existing Models

The comparison of accuracy and error rate for the existing systems and the proposed system (ML-PEQRM) is shown in Figure 9. It shows that the proposed system acquires 99% of accuracy and less error rate when compared with the other systems. The validity and reliability of the results were ensured by using the same dataset throughout the study. All models were applied to the same dataset, which was derived from GitHub repositories, to ensure fair and unbiased performance evaluation. This comparison was designed to assess the effectiveness of the proposed ML-PEQRM model when predicting software quality, reliability, maintenance needs, and costs. As a comparison, Gaussian Naive Bayes, Random Forest, and Support Vector Machines (SVM) were used, which are widely recognized for their effectiveness in classification and regression? A 70% training and 30% testing split of the dataset was used to maintain a consistent evaluation framework. The performance of these models was measured using standard evaluation metrics such as accuracy, precision, recall, F1-score, Mean Absolute Percentage Error (MAPE), and R^2 Score. Five-fold cross-validation was performed across all models to verify the reliability of the comparison. With this technique, the evaluation is not dependent on a single dataset split and performance fluctuations are minimized. A Wilcoxon signed rank test was also conducted to confirm the significance of the observed improvements in the ML-PEQRM model. By using uniform datasets and rigorous evaluation methods for all comparisons. By demonstrating credibility, reproducibility, and accurate representation of performance advantages, the proposed model is proven to be credible, reproducible, and accurate.

Maintenance Cost and Reliability Improvements

The proposed model's ability to reduce maintenance costs and improve reliability was a key outcome. By integrating product and process metrics, the ML-PEQRM model achieved a 25% reduction in costs and a 20% improvement in reliability compared to conventional methods.

Table 8: Maintenance Cost for the Software Project

Project	Month	Employees	Cost / Year
Anasthase_TintBrowser	5	3	75000
billthefarmer_tuner	1	1	45000
budowski_budoist	8	4	90000
czlee_debatekeeper	4	2	60000
devonjones_PathfinderOpenReference	2	1	45000
eolwral_OSMonitor	5	3	75000
fython_Blackbulb	4	2	60000
gsantner_markor	5	3	75000

HenriDellal_emerald	2	1	45000
hwki_SimpleBitcoinWidget	9	5	105000
hypeapps_Endoscope	1	1	45000
koush_Superuser	3	2	60000
lordi_tickmate	2	1	45000
markusfisch_ShaderEditor	4	2	60000

In Table 8, the maintenance cost, required months, required employees, quality index for the project, as well as security index are shown. The maintenance cost estimation is computed automatically based on the security (SI) and quality (QI) index from the prediction of the requirement of maintenance index (MIV). The MIV is further employed to compute the cost per year, employees' requirement and duration of the maintenance for each project. These findings underscore the practical utility of ML-PEQRM in optimizing software maintenance processes and improving long-term software reliability. While the datasets used in the IntelliEstimator and SVS Framework thesis (26) are sourced from GitHub repositories, their scope, focus, and diversity differ. With a focus on software maintenance cost estimation, reliability, and quality prediction, IntelliEstimator uses 25 Java projects with 10,000 code samples. A number of metrics, such as Quality Index (QI), Security Index (SI), and Maintenance Index (MI), as well as historical commit logs and defect tracking data, are incorporated into the system. As an alternative, the SVS Framework (26) dataset is also based on GitHub projects, but uses software code metrics to assess software quality. From network packet analysis to interior design tools, a range of open-source projects such as Hprose, Sweet Home 3D, MyBatis, JabRef, and JWildFire are included in the dataset. A defect detection, maintainability assessment, and performance evaluation do not feature in IntelliEstimator dataset, while they do in SVS dataset. Moreover, the SVS Framework combines object-oriented principles with machine learning for defect detection, whereas IntelliEstimator combines ensemble learning (Stacking XGBoost and Random Forest) for improvement of software maintenance estimates. They may both be derived from GitHub, but their primary difference lies in their intended analyses: maintenance cost estimation vs. software quality evaluation. The superior performance of ML-PEQRM can be attributed to its novel feature engineering techniques and ensemble learning approach. By leveraging MGFPA for feature selection, the model effectively reduced dimensionality and noise, enhancing prediction accuracy. Furthermore, the integration of both product and process metrics enabled holistic predictions, addressing gaps in existing methodologies.

In this section, the comparative analysis and discussion between the proposed system *ML-PEQRM* and the existing system are described. By comparing the systems, the potential of accessing each solution for identification and mitigation will be an option for a successful system. The existing system (10) focused on the importance of effective software cost estimation and the limitations of traditional regression-based algorithms like the *constructive* cost model (COCOMO) in accurately estimating software costs. It highlights the need for fine-tuning coefficients to account for variations across different organizations. The hybrid algorithm aims to find an optimal solution while minimizing computational costs in the work. It is used to optimize the *COCOMO II* coefficients and improve the training process of deep learning models. The experimental results showed that the hybrid HACO-BA algorithm outperformed ACO and BA in fine-tuning COCOMO II coefficients. Additionally, HACO-BA demonstrated better performance in optimizing the DNN training process in terms of execution time and accuracy. The proposed DNN approach achieved an accuracy of approximately 98%, while traditional neural networks (NN) achieved up to 85% accuracy on the same datasets. Also in the work (11), a Flower Pollination Algorithm (FPA) is proposed to optimize the parameters of the Constructive Cost Model II (COCOMO-II) using a standard Turkish industry dataset. The FPA is a metaheuristic algorithm inspired by the pollination behavior of flowers. It aims to find the optimal solution for parameter optimization in the COCOMO-II model. Experimental results demonstrate that the

proposed FPA algorithm outperforms existing approaches like the Bat algorithm and the original COCOMO-II in terms of Manhattan distance (MD) and mean magnitude of relative errors (MMRE). This indicates that the FPA algorithm provides better estimations, improving the accuracy of cost estimation for software projects. From the analysis some of the limitations are attained based on the existing systems: The COCOMO II solutions may have been evaluated on specific datasets or industry settings, limiting their generalize ability to other scenarios. The work primarily focuses on optimizing COCOMO II coefficients and improving the training process of deep learning models. However, it does not provide a thorough comparison of the cost estimation performance between the proposed algorithms and traditional methods like COCOMO II. Such a comparison would help assess the actual improvement achieved by the proposed solutions. As compared with COCOMO-II optimizations, the ML-PEQRM model demonstrated better scalability. A black-box approach lacks explainability, while interpretable metrics provide stakeholders with actionable insights. An analysis of the model's predictive capability was conducted using a combination of classification and regression metrics. A combination of precision, recall, F1-score, and overall classification accuracy was used to evaluate software quality, reliability, and maintenance needs. On the basis of extracted software attributes, the model classified projects into predefined quality and reliability classes. In the estimation of maintenance costs, which is a continuous variable, the accuracy was determined by the mean absolute percentage error (MAPE) and the coefficient of determination (R2 Score). This metric reduces the risk of errors caused by overestimation or underestimations by ensuring estimated costs closely match actual values. For maintenance cost prediction, the ML-PEQRM model demonstrated a near-perfect correlation between predicted and actual values, achieving an accuracy rate of 99% in classification tasks and a MAPE of 1.5% with an R2 Score of 0.99. This claim was validated using a 5-fold cross-validation, which ensured generalizability. Additionally, comparisons with baseline models, such as Gaussian Nave Bayes (86% accuracy) and Random Forest (91% accuracy), further support the proposed approach's superiority. When tested on previously unknown software projects, the model maintained an accuracy of 98% or higher. Thus, the 99% accuracy claim can be justified based on extensive experimental validation, rigorous evaluation metrics, and comparative performance assessment, demonstrating the ML-PEQRM model is reliable and effective in predicting software quality and maintenance.

In recent years, several studies have introduced innovative methods for estimating software maintenance costs. Despite their advancements in specific areas, these methods often lack generalizability, interpretability, or holistic metrics. ML-PEQRM addresses these gaps and is compared with the following recent works: The ML-PEQRM model presents substantial improvements over recent methods in software cost and maintenance estimation. For instance, the Genetic Algorithm for Software Development Cost Estimation (18) achieved high accuracy by reducing uncertainty in development cost factors. However, it was limited in scope, focusing solely on development costs and excluding essential maintenance metrics and process data. ML-PEQRM addresses these limitations by integrating both product and process metrics, enabling more accurate predictions of software quality, reliability, and maintenance costs. Additionally, the model leverages a stacking ensemble technique to enhance predictive accuracy, reaching up to 99%. Similarly, the Ant Colony Optimization with Fuzzy-Neural Networks (19) demonstrated improved training efficiency and prediction accuracy for effort estimation. Despite its strengths, the model lacked interpretability and was not tested in comprehensive maintenance contexts. ML-PEQRM overcomes these issues by incorporating interpretable machine learning techniques and a rich set of metrics, delivering actionable insights and greater applicability in real-world scenarios. The Two-Stage Life Cycle and Cost Estimation Framework (20) linked development and maintenance phases, offering a lifecycle-oriented perspective. However, it fell short in feature engineering, particularly concerning software quality and reliability. ML-PEQRM extends this framework by integrating the Modified Global Flower Pollination Algorithm (MGFPA) and introducing novel metrics such as abstraction

density and dynamic change metrics, achieving superior results across diverse software domains. Researchers also proposed an LSTM-CRF-Based Paradigm for software cost estimation (21). Although effective in its specific context, the model operated as a black box and was limited to cost estimation, neglecting reliability and maintenance considerations. ML-PEQRM surpasses this by offering a broader and more transparent framework that includes quality, reliability, and maintenance cost predictions, ensuring generalizability across varied datasets. Lastly, the Multi-Criteria Decision-Making (MCDM) Framework focused on software reliability prediction using multi-metric accuracy evaluation (22). While it demonstrated robust decision-making capabilities, it did not include maintainability metrics or estimate maintenance costs. ML-PEQRM fills this gap by incorporating a comprehensive metric set, including the Quality & Security Index, and delivers high predictive accuracy (99%) along with practical value for real-world software project planning.

Strengths and Limitations

Strengths: The ML-PEQRM model demonstrates exceptional performance with a 99% accuracy rate, surpassing the typical 85%-98% accuracy range of neural networks and traditional estimation techniques. It takes a holistic approach by combining static code metrics with dynamic change metrics, providing a comprehensive model for estimating software quality, reliability, maintenance needs, and associated costs. The study also achieved impressive results, including a 25% reduction in maintenance costs and a 20% improvement in reliability. Its scalability allows the model to efficiently handle large datasets and accommodate diverse project types, making it suitable for a wide range of real-world scenarios. Additionally, the use of novel feature engineering techniques, such as the Modified Global Flower Pollination Algorithm (MGFPA) and the creation of new metrics like cyclomatic density and abstraction density, significantly enhanced the model's predictive power.

Limitations: The ML-PEQRM model, while promising, has some limitations. First, the study used a dataset of 10,000 samples from 25 Java projects, which may not fully capture the diversity of software systems across different industries. The generalizability of the model could be improved by testing it on larger and more varied datasets. Additionally, the model primarily focuses on static and dynamic code metrics, potentially overlooking other important factors such as team experience or hardware specifications, which could affect software maintenance and quality. The model was also not validated against existing methods in real-world industrial case studies, which could have strengthened the practical applicability of the findings. Finally, the model's reliance on substantial computational resources might pose a challenge for smaller organizations or teams with limited access to such resources.

Table 9: Comparative Aspect of Proposed and Existing Works

Aspect	Proposed Work (ML-PEQRM)	Other Works
Metrics Integration	Product + Process + Novel Metrics	Limited to effort estimation metrics
Feature Selection	MGFPA (optimized, scalable, handles noise)	Traditional, slower, less robust
Machine Learning	Stacking Ensemble (XGBoost + Random Forest)	Single models like Neural Networks
Accuracy	99%	85–98%
Scope	Quality, reliability, cost, and maintenance	Cost or effort-only predictions
Interpretability	Clear and actionable predictions	Limited interpretability
Comparative Analysis	Benchmarked against COCOMO II, neural networks	Rarely benchmarks with modern approaches
Practical Impact	25% cost reduction, 20% reliability improvement	Focused on narrow domains or datasets

Table 9 shows the Compartive aspect the proposed work with other Existing Works. It analysis the metrics integtrationl, ML Model and Impact of the work. While the IntelliEstimator framework demonstrates strong predictive capabilities in controlled testing environments, the scalability of

the model in practical, large-scale construction projects warrants further investigation. Initial experiments indicate that the system maintains consistent performance with moderate increases in data volume and project complexity. However, real-world scalability depends on factors such as data availability, integration with enterprise systems, and the variability of project types and geographic contexts. To address this, the framework has been designed with modular components and support for cloud-based deployment, allowing it to scale horizontally by distributing computational tasks. Future work will involve large-scale pilot implementations across diverse construction scenarios to evaluate the system's responsiveness, adaptability, and resource requirements under operational constraints. This will help ensure the IntelliEstimator remains viable and efficient as it transitions from a prototype to a production-ready solution. To ensure the practical relevance and usability of the Intelli Estimator framework, the study has actively considered potential pathways for real-world implementation. Preliminary discussions have been initiated with industry stakeholders, including construction firms and project management consultancies, to explore pilot testing opportunities. These collaborations aim to validate the system's effectiveness in live environments and to gather user feedback for refining the interface and integration workflows. The framework's design emphasizes real-time data integration, compatibility with existing project management systems, and ease of deployment—key considerations for industry adoption.

Conclusion

This research addresses the challenge of enhancing software maintenance decision-making processes by applying machine-learning techniques. Optimizing resource allocation and improving software maintenance efficiency were the main objectives. As a result of the findings, conventional software estimation approaches fail to account for issues such as inaccurate estimates, lack of reliability, and resource inefficiency. To achieve superior predictive performance, static code attributes and dynamic change metrics were integrated. By considering software quality, reliability, maintenance, and cost estimation, the proposed Model (ML-PEQRM) improves cost estimation and project planning accuracy of 99%. This model's ability to consider software quality, reliability, and maintenance needs significantly contributed to its effectiveness ratio of 98%. Several software metrics were computed, features were generated, and preprocessing was performed to evaluate software metrics. Preprocessing techniques include handling missing data, removing duplicates, and consolidating features. Using feature selection algorithms then reduces the risk of overfitting and increases accuracy in estimating maintenance costs by identifying the most significant features. The main contribution of this work is a discussion of the potential benefits of machine learning algorithms for estimating maintenance costs in software development projects. The future direction of the research could be to expand the model to consider other factors that impact software development costs. Costs associated with hardware, infrastructure, and project management could be included in these considerations.

Abbreviations

None.

Acknowledgement

None.

Author Contributions

The corresponding author confirm sole responsibility for the following: study conception and design, data collection, analysis and interpretation of results, and manuscript preparation.

Conflict of Interest

None.

Ethics Approval

Not applicable.

Funding

No Funding.

References

1. Qamar N, Malik AA. A Quantitative Assessment of the Impact of Homogeneity in Personality Traits on Software Quality and Team Productivity. IEEE Access. 2022;10(1):122092–122111.
2. Li Z, Qi X, Yu Q, Liang P, Mo R, Yang C. Exploring multi-programming-language commits and their

impacts on software quality: An empirical study on Apache projects. J Syst Softw. 2022;194(1):1-18.
3. Jagtap M, Katragadda P, Satelkar P. Software Reliability: Development of Software Defect Prediction Models Using Advanced Techniques. Annu Reliab Maintainab Symp (RAMS). 2022;1(1):1–7.
4. Huang YS, Chiu KC, Chen WM. A software reliability growth model for imperfect debugging. J Syst Softw. 2022;188(1):1–15.
5. Li L. Software Reliability Growth Fault Correction Model Based on Machine Learning and Neural Network Algorithm. Microprocess Microsyst. 2021;80(2):1–10.
6. Gupta C, Inácio PRM, Freire MM. Improving software maintenance with improved bug triaging. J King Saud Univ Comput Inf Sci. 2022;34(10):8757–8764.
7. Gandomani TJ, Dashti M, Nafchi MZ. Hybrid Genetic-Environmental Adaptation Algorithm to Improve Parameters of COCOMO for Software Cost Estimation. Int Conf Distrib Comput High Perform Comput (DCHPC). 2022;1(1):82–85.
8. Akhbardeh F, Reza HA. A Survey of Machine Learning Approach to Software Cost Estimation. IEEE Int Conf Electro Inf Technol (EIT). 2021;1(1):405–408.
9. Yadav N, Gupta N, Aggarwal M, Yadav A. Comparison of COSYSMO Model with Different Software Cost Estimation Techniques. Int Conf Issues Challenges Intell Comput Tech (ICICT). 2019;1(1):1–5.
10. Ullah B, Wang J, Sheng J, Long M. A Novel Technique of Software Cost Estimation Using Flower Pollination Algorithm. Int Conf Intell Comput Autom Syst (ICICAS). 2019;3(1):654–658.
11. Hassan CAU, Khan MS, Irfan R, Iqbal J, Hussain S, Ullah SS, Alroobaea R, Umar F. Optimizing Deep Learning Model for Software Cost Estimation Using Hybrid Meta-Heuristic Algorithmic Approach. Comput Intell Neurosci. 2022;20(1):1–20.
12. Ralhan C, Malik A. A Study of Software Clone Detection Techniques for Better Software Maintenance and Reliability. Int Conf Comput Sci (ICCS). 2021;2(1):249–253.
13. Hardt R. A software maintenance-focused process and supporting toolset for academic environments. IEEE Int Conf Softw Maint Evol (ICSME). 2020;1(1):360–370.
14. Rojek I, Jasiulewicz-Kaczmarek M, Piechowski M, Mikołajewski D. An Artificial Intelligence Approach for Improving Maintenance to Supervise Machine Failures and Support Their Repair. Appl Sci. 2023;13(8):137-141.
15. Ren Y. Optimizing Predictive Maintenance with Machine Learning for Reliability Improvement. ASME J Risk Uncertain Part B. 2021;7(3):3–20.
16. Dalzochio J, Kunst R, Pignaton E, Binotto A, Sanyal S, Favilla J, Barbosa J. Machine learning and reasoning for predictive maintenance in Industry 4.0: Current status and challenges. Comput Ind. 2020;123(1):1-15.
17. Zhong D, Xia Z, Zhu Y, Duan J. Overview of predictive maintenance based on digital twin technology. Heliyon. 2023;9(4):1–10.
18. Amarif M, Owaydat S. An Optimal Optimization of Software Development Cost Estimation Using Genetic Algorithm. IEEE Int Maghreb Meet Conf Sci Tech Autom Control Comput Eng (MI-STA). 2024;1(1):654–659.
19. Afshari M, Gandomani TJ. Enhancing Software Effort Estimation with Ant Colony Optimization Algorithm and Fuzzy-Neural Networks. Int Conf Distrib Comput High Perform Comput (DCHPC). 2024;1(1):1–6.
20. Zhu X, Fu B, Lu Y, Lin T, Lv X, Wu Y. A Kind of 2-Stage Software Life Cycle and Cost Estimation Framework in Agile Methodology from a System Engineering Perspective. IEEE Int Conf Softw Eng Artif Intell (SEAI). 2024;1(2):188–193.
21. Zhu X, Fu B, Lu Y, Lin T, Lv X, Wu Y. A Kind of Paradigm-Based Software Cost Estimation Method Using LSTM-CRF. IEEE Int Conf Softw Eng Artif Intell (SEAI). 2023;1(3):22–27.
22. Kumar A, Singh AK, Garg A. A Novel Framework to Evaluate Software Reliability Prediction Models Using Multi-Criteria Decision-Making. Int Conf Reliab Infocom Tech Optim (ICRITO). 2024;1(1):1–5.
23. Ding Z, Mo Y, Pan Z. A Novel Software Defect Prediction Method Based on Isolation Forest. Int Conf Qual Reliab Risk Maint Saf Eng (QR2MSE). 2019;2(1):882–887.
24. Govindaprabhu GB, Sumathi M. Ethno medicine of Indigenous Communities: Tamil Traditional Medicinal Plants Leaf detection using Deep Learning Models. Procedia Comput Sci. 2024;235(1):1135–1144.
25. Govindaprabhu GB, Sumathi M. Safeguarding Humans from Attacks Using AI-Enabled (DQN) Wild Animal Identification System. Int Res J Multidiscip Scope (IRJMS). 2024;5(3):285–302.
26. Visagan AR, Sumathi M. A Framework for Software Quality Enhancement through Data Mining. Shodhganga. 2019. https://shodhganga.inflibnet.ac.in/handle/10603/347294.
27. Govindaprabhu GB, Sumathi M, Neyvasagam S, Kumar NAJ. Bridging AI and ecology: CILNN and XAI for acoustic-based prediction of dangerous wild animals. Int Res J Multidiscip Scope (IRJMS). 2025;6(1):1280–1298.
28. Mahalakshmi Priya R, Sumathi M. Impact of microorganisms on food spoilage and human health: A comprehensive review of advances in identification using image processing and artificial intelligence techniques. Int Res J Multidiscip Scope (IRJMS). 2025;6(1):1299–1316.