



International Journal of Artificial Intelligence and Machine Learning

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



Research Paper

Open Access

Artificial Intelligence-Based Software Engineering Framework for Automated Program Quality Assessment

Dr. Resmi G Nair¹, Dr Sangeetha Shibu², Mr Jinu Raj R³, Mrs Divya GS⁴, Mrs Divya V B⁵, Mrs Jincy Jesudasan⁶

¹Dean Academics & HOD of Department of Artificial Intelligence and Data Science, Holy Grace Academy of Engineering, Thrissur, Kerala, India. Email ID: reshmignair82@gmail.com, Orcid ID: 0000-0002-4409-522X.

²Professor & HOD, Department of Computer Science and Engineering, Rajadhani Institute of Engineering and Technology, Trivandrum, Kerala, India. Email ID: sangeethas@rietedu.in, Orcid ID: 0009-0000-0606-8776

³Assistant Professor, Department of Computer Science and Engineering, Rajadhani Institute of Engineering and Technology, Trivandrum, Kerala, India. Email ID: jinuajr@rietedu.in, Orcid ID: 0009-0004-7346-2991

⁴Assistant Professor, Department of Computer Science and Engineering, Rajadhani Institute of Engineering and Technology, Trivandrum, Kerala, India. Email: divyags@rietedu.in, Orcid ID: 0009-0003-0169-9394

⁵Assistant Professor Department of Computer Science & Engineering, Rajadhani Institute of Engineering and Technology, Trivandrum, Kerala, India. Email ID: divyavb@rietedu.in

⁶Assistant Professor, Dept of Computer Science & Engineering, Rajadhani Institute of Engineering and Technology, Trivandrum, Kerala, India. Email ID: jincyjesudasan@rietedu.in

ABSTRACT

The evaluation of the quality of software products plays an important role in software engineering, enabling the creation of robust, maintainable, and secure software applications with great performance levels. With the growth in size and complexity of modern software projects, traditional methods for quality assessment using hand-crafted software metrics and supervised machine learning need much feature engineering and labeled defect data. In order to overcome difficulties outlined above, in this research, framework for automated program quality assessment based on Artificial Intelligence-Based Software Engineering (AISE) is suggested. This framework encompasses use of AI-SEPQA dataset, data preprocessing, feature extraction through CodeBERT (Bidirectional Encoder Representations from Transformers), quality prediction using hybrid approach Dynamic Tunicate Swarm Algorithm-fused Least Square Support Vector Machine (DTSA-LS-SVM). AI-Based Software Engineering Program Quality Assessment (AI-SEPQA) dataset consisting of 12,000 software modules is used and preprocessed via comment removal, lexical normalization, and tokenization. Machine Learning-based DTSA-LS-SVM quality prediction for evaluation of various software quality metrics such as defect proneness, code complexity, maintainability, reliability, code standards conformance. DTSA fine-tunes LS-SVM parameters to ensure convergence and generalization, while LS-SVM helps in predicting software quality metrics with better classification accuracy and prediction. Experimental evaluation is done using Python, and proposed framework is evaluated on the basis of various performance measures, including accuracy of 98.25%. The results obtained by applying the AISE framework have been found to be statistically significantly better than those obtained through the use of traditional approaches ($p = 0.003$). The framework offers a smart way of assessing program quality automatically.

Keywords: Artificial Intelligence, Software Engineering, Source Code Representation, Machine Learning, Software Defect Prediction, Software Quality Analysis

1. Introduction

Software engineering has gone through tremendous changes with the increase in complexity of software systems and thus created a need for effective ways of assessment of program quality [1]. A Software Engineering Framework for Automated Program Quality Assessment enables a systematic evaluation of software products during the entire software life cycle [2]. The integration of software metrics, coding rules, testing process, defects detection and quality evaluation techniques to evaluate critical characteristics such as correctness, maintainability, reliability, readability, efficiency, security and adherence to the programming principles [3]. The process of program quality evaluation is automated in order to reduce the reliance on manual code evaluation and ensure objective assessment of the software quality. Automated evaluation of software quality is an important part of modern software engineering, since it helps to develop reliable software systems [4-5].

AI is indispensable for Software Quality Assurance through its ability to predict defects, perform automated assessment of quality and data-based decisions. Figure 1 shows the source code, test results and the behavior of software to make testing more efficient, reliable software, increase productivity, and improve the overall quality of software [6-7]. The extensive application of automated quality assessment tools for the programs has transformed software engineering in varied industrial and academic settings. They are commonly applied in CI/CD pipeline, software testing, education for programming, development of enterprise applications, cloud software applications, embedded software, and open source software [8-9]. Moreover, automated quality assessment is also helpful in software maintenance because it assists in early detection of defects, debugging,

improves documentation and teamwork [10]. The incorporation of automated quality assessment in the process of software engineering leads to faster delivery of software, greater reliability of the product and lower costs of maintenance [11].

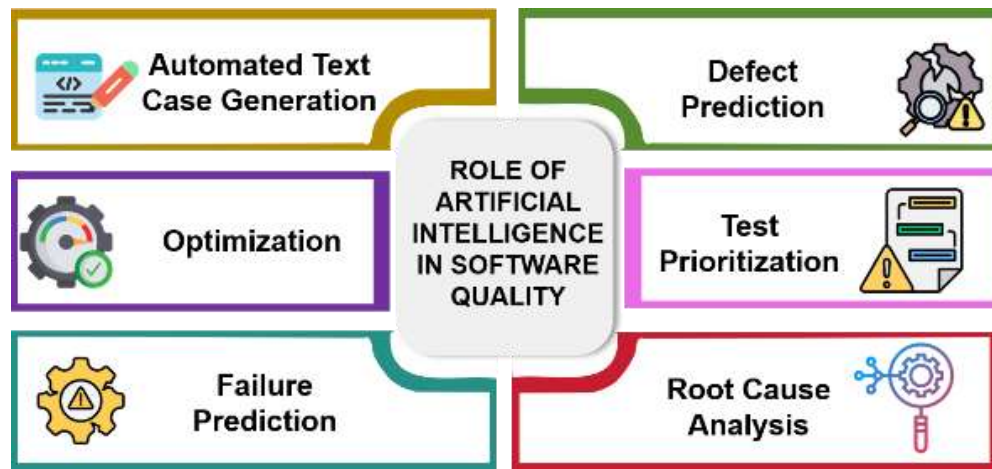


Figure 1: Role in Artificial Intelligence (AI) Software Quality Assessment

Many existing systems are highly dependent on rule-based evaluation procedures, quality metrics that were built up beforehand, and static analysis methods that are often unable to detect the semantics and contextually correct behavior of software programs [12]. The diversity of programming languages used today, the architecture of software, programming styles, and the rapidly changing development environment make the problem of software quality assessment even harder. The tools of traditional assessment give many false alerts, which complicates the work of the developer in identifying defects and just warnings in the software [13]. Moreover, the size of modern software databases, distributed software development, and the frequency of code change become another obstacle to solving this problem [14].

Automated frameworks for program quality assessment provide many benefits for software engineering firms. Firstly, automation facilitates reduction in manual inspection work, speeding up software validation processes, provides consistency in the quality measurement process, and allows continuous monitoring of software during its development [15-16]. In this way, it helps detect programming errors, non-compliance with coding standards, security weaknesses, and maintainability problems at an early stage, avoiding further propagation of the detected problems [17]. By enhancing programmers' productivity through immediate feedback, code refactoring, quality assurance activities, and ensuring reliability of software, these paradigms can also facilitate higher productivity of programmers. Hence, there are many disadvantages in traditional automated quality assessment frameworks [18]. For example, automation techniques rely on manual procedures and hence lack the capability to understand the semantic meaning of the programs, programmer's intent, and specific requirements of implementation in specific areas. This means that there are a number of research challenges that have been affecting the creation of automated methods of program quality evaluation in software engineering [19]. Some of these challenges include the issue of software quality evaluation in various programming languages, identification of complex logic errors, avoidance of false positives, among many others. Solving these challenges is critical in the creation of next-generation software engineering paradigms that will be able to provide efficient automation of program quality evaluation [20].

1.1 Problem statement: The conventional approach for evaluating software quality involves the process of manually identifying features and creating labeled data sets, which makes the approach less scalable, accurate, adaptive, and fails in early detection of defects. In order to address such shortcomings, the research suggests an Artificial Intelligence-Based Software Engineering (AISE) Framework for Automated Program Quality Assessment that uses a Dynamic Tunicate Swarm Algorithm-fused Least Square Support Vector Machine (DTSA-LS-SVM) method. This method is expected to automatically assess software program quality accurately.

1.2 Research Organization: The challenges of automated quality estimation of programs and research work on AISE are discussed in Sections 1 and 2, respectively. The description of the suggested AISE approach based on DTSA-LS-SVM algorithm is discussed in Section 3, while experimental analysis and its evaluation is discussed in Section 4. Discussion and results analysis is discussed in Section 5, and conclusions and future work is stated in Section 6.

2. Related work

The previous research has greatly contributed to the development of the software engineering paradigm, AI, software development, security, sustainability, and quality assurance. Table 1 depicts that the previous research has been based on specific domains and no model of AI that can assess the quality of programs intelligently has been developed.

Table 1: Literature work of Automated Program Quality Assessment

Ref.	Objective	Dataset	Method	Results	Limitations
[21]	A decision-making framework for addressing quantum software engineering challenges.	Quantum software engineering case studies	Utilized a multi-criteria decision-making framework for prioritizing software engineering issues.	Identified critical challenges and supported effective decision-making in quantum software projects.	Limited to quantum software engineering and lacked automated quality assessment mechanisms.
[22]	Analyzed software engineering research related to the Robot Operating System (ROS).	582 ROS-related research publications	Conducted systematic mapping research by reviewing and classifying published literature.	Identified research trends, challenges, and future directions in ROS software engineering.	Presented only a literature mapping without proposing an automated quality assessment.
[23]	Developed an information security engineering method for firewall modeling.	Firewall security modeling case studies	Employed Neutrosophic Petri Nets for secure packet-filtering firewall modeling.	Improved firewall security modeling and reliability analysis.	Focused on network security engineering instead of software code quality evaluation.
[24]	Established a framework for measuring open-source software innovation.	Open-source GitHub repositories	Integrated repository analytics with innovation measurement indicators.	Evaluated innovation across 98 repositories using repository metrics; no predictive accuracy reported.	Emphasized innovation measurement rather than automated program quality assessment.
[25]	Investigated the usage of mocking in Apache Software Foundation projects.	Apache Software Foundation repositories	Performed empirical analysis on large-scale open-source software repositories.	Analyzed 95.3% of test classes and mocking instances; no quality prediction model developed.	Focused on software testing model without evaluating overall program quality.
[26]	Evaluated a Business Process Model and Notation (BPMN) extension for security requirements engineering.	BPMN security modeling examples	Applied comparative evaluation techniques for validating BPMN security extensions.	Improved modeling and management of software security requirements.	Limited to security requirement engineering and excluded software quality evaluation.

[27]	Assessed the quality of language product lines using language decomposition.	Language Product Line (LPL) benchmark projects	Conducted empirical quality assessment of language product line architectures.	Enhanced modularity, maintainability, and quality of language product lines.	Restricted to language engineering and not applicable to general software quality assessment.
[28]	A model of edge computing and Machine Learning (ML) for software sensors.	Edge computing software sensor datasets	Combined edge computing architecture with ML-based software sensor development.	Achieved improved monitoring accuracy and computational efficiency.	Addressed software sensor development instead of automated program quality assessment.
[29]	Examined the relationship between computational thinking and program quality.	Student programming assignment datasets	Conducted empirical experiments and statistical analysis of programming performance.	Found a positive relationship between computational thinking skills and software quality.	Focused on educational assessment rather than automated software quality evaluation.
[30]	Compared structured and agile software development methodologies.	Industrial software development projects	Performed comparative analysis using software engineering performance metrics.	Identified strengths and weaknesses of structured and agile development approaches.	Did not include automated code quality assessment or AI-based evaluation techniques.
[31]	A quality assurance for logistics packaging.	Industrial logistics packaging datasets	Developed and validated an industrial quality assurance model.	Improved process consistency, packaging quality, and operational performance.	The designed for logistics applications rather than software engineering.
[32]	Synthesized recent research on AI in software engineering.	200+ AI software engineering studies	Conducted a synthetic knowledge synthesis of AI-based software engineering researches.	Demonstrated AI benefits in software development, testing, maintenance, and project management.	The existing researches without proposing an implementation.
[33]	Developed automated software quality assessment using self-supervised source code representations.	AI-SEPQA benchmark dataset	Self-Supervised Learning model extracted semantic source code representations for automated software quality assessment.	Achieved 94.18% accuracy for automated software quality assessment using learned semantic source code.	Required further optimization techniques for improved prediction accuracy, scalability, robustness.

2.1 Research gap: Existing research mainly focuses on specific areas such as security engineering, software testing, repository analysis, agile methodologies, or AI applications in software development [21-25]. However, to comprehensive automated framework for program quality assessment. Current approaches depend on manual metrics and feature engineering, limiting scalability, adaptability, and accurate early identification of software quality risks [26-33]. To overcome these limitations, this research proposes an AISE Framework for

Automated Program Quality Assessment using DTSA-LS-SVM prediction to automatically assess program quality indicators, reduce feature engineering dependency, and improve quality prediction accuracy.

3.Methodology

This section presents the proposed AISE Framework for automated program quality assessment. The overall architecture of the proposed framework is illustrated in Figure 2. It is designed to predict multiple software quality attributes from source code using an intelligent learning framework. The process consists of AI-SEPQA dataset acquisition, data preprocessing, feature extraction using CodeBERT, hyperparameters optimization using the DTSA-LS-SVM classifier.

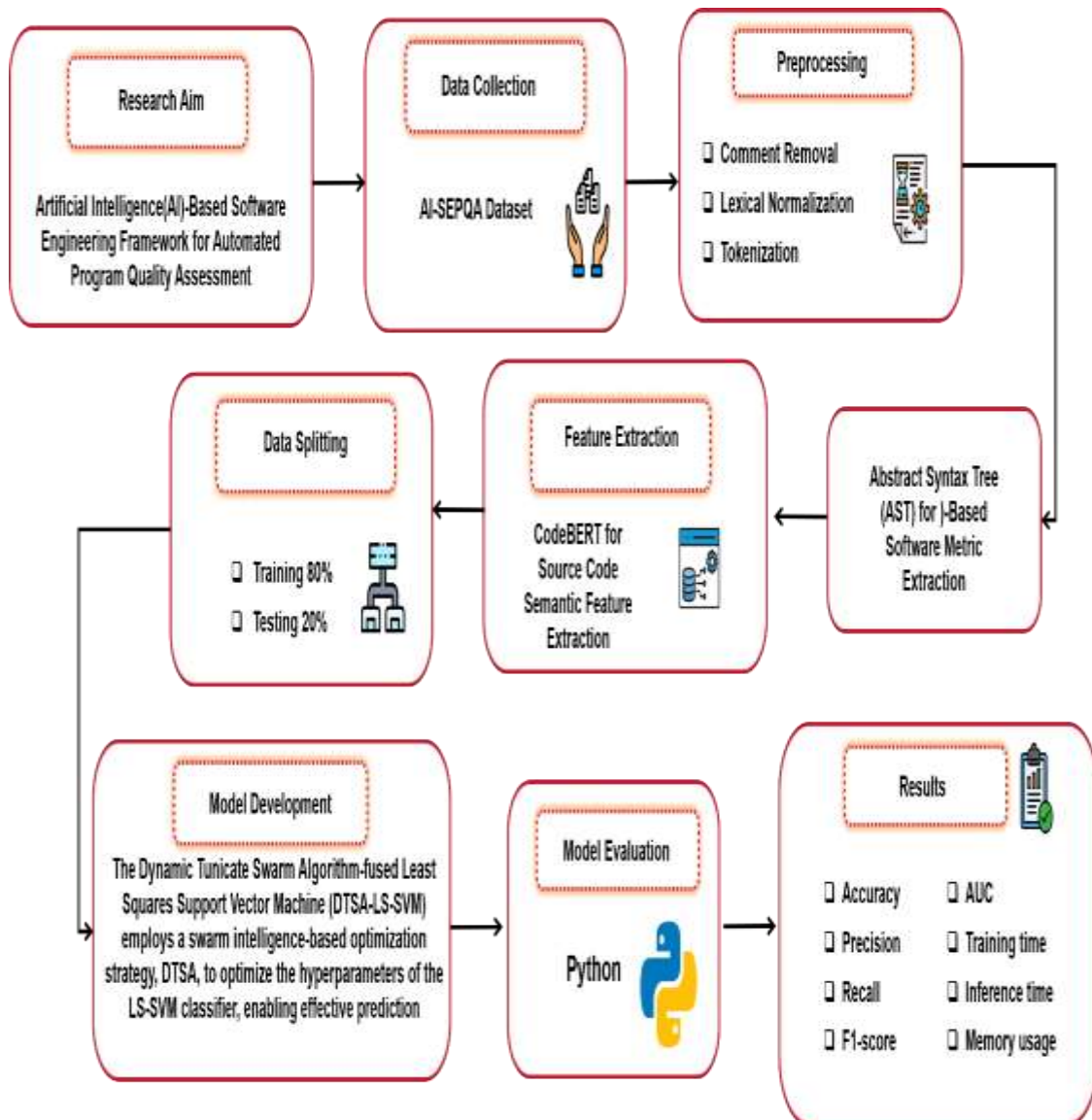


Figure 2: Overall Architecture of the Proposed Framework

3.1 Data Acquisition

The Artificial Intelligence-Based Software Engineering Program Quality Assessment (AI-SEPQA) Dataset is a structured benchmark dataset developed to support research on software quality evaluation and program analysis. It contains 12,000 software module instances, each described by 32 software engineering attributes representing various aspects of source code characteristics and software quality. The dataset was divided into 80% training and 20% testing subsets.

The dataset includes measurements related to code size, complexity, maintainability, object-oriented design, reliability, security, dependency structure, and coding standard compliance. These attributes provide a comprehensive representation of software modules, making the dataset suitable for analyzing overall program quality and identifying quality-related patterns. Each instance is assigned one of five software quality categories: 0 - Excellent, 1 - Good, 2 - Moderate, 3 - Poor, 4 - Critical. (Link: <https://www.kaggle.com/datasets/ziya07/ai-sepqa-dataset/data>)

3.2 Data preprocessing

The AI-SEPQA dataset consists of raw software source code along with comments, inconsistencies in formatting, duplicate symbols text. The process of preprocessing involves removing comments, normalizing the lexicon, and tokenization to create a standardized form of the lexicon, which facilitates the measurement of software metrics.

- **Comment Removal for Source Code Noise Elimination**

Raw software source code files containing executable statements, single-line comments, multi-line comments, documentation blocks, and unnecessary textual annotations. The elimination of comments removes textual information that is non-executable and does not add any functionality to the program, to eliminating extraneous information and saving processing time. This facilitates the further lexical normalization and tokenization process of extracting software metrics, since now all these processes to operate purely on the executable code of the program.

- **Lexical Normalization for Source Code Standardization**

Comment-free source code containing identifiers, keywords, operators, literals, delimiters, and formatting inconsistencies requiring standardized lexical representation. Lexical Normalization involves the standardization of source code through the standardization of format of identifiers, deletion of unnecessary delimiters and addressing formatting differences. Lexical Normalization makes the process of lexical representation of the source code uniform, minimizes variation among code samples and improves accuracy of tokenization, parsing and software metrics computation. Lexically normalized source code with standard identifiers and formatting for tokenization and parsing.

- **Tokenization for Source Code Lexical Representation**

Lexically normalized source code containing standardized identifiers, keywords, operators, literals, and delimiters prepared for lexical token generation. Tokenization is the process through which normalized source code is divided into various lexical tokens such as keywords, identifiers, operators, literals, and separators. Tokenized code enables the effective analysis of programs, syntax analysis, and extraction of software metrics among other benefits. Structured lexical token sequence representing executable source code, enabling accurate syntax parsing, software metric extraction, and automated quality prediction.

3.3 Abstract Syntax Tree (AST)-Based Software Metric Extraction

Following source code preprocessing, the preprocessed source code files were extracted from public software repositories, containing classes, methods, statements, and software quality-related structural information. AST is constructed to model the syntactical structure of the source code and to retain the relations between programming language constructs. The AST captures the structure of program elements including classes, methods, variables, control structures, and expressions into an organized format that static code analysis. The suggested framework uses the AST to detect object-oriented features and calculate software engineering metrics that indicate software quality aspects. Such metrics cover size, complexity, coupling, cohesion, inheritance, and maintainability metrics that measure structural attributes of each software module. The computed metrics are then aggregated into a feature vector in multiple dimensions to represent the program. The feature vector is then provided to assess program quality in an automated manner. Through extracting structural attributes from parsed source code, AST-based feature extraction decreases the reliance on human analysis of the code. Therefore, automated software quality assessment predicts defect proneness, maintainability, reliability, code complexity, coding standard compliance, and overall quality classification.

3.4 CodeBERT for Source Code Semantic Feature Extraction

The processed source code tokens and AST-derived code sequences are provided as input to CodeBERT for semantic feature representation. CodeBERT is a pretrained transformer-based language model developed for programming languages and natural language. It extracts semantic and contextual representations from source code by processing tokenized inputs through embedding layers and transformer encoders with self-attention mechanisms. The resulting feature representations capture syntactic structure and code semantics, making them suitable for automated software quality assessment and defect prediction.

$$F(ts) = \{F[cls], F[t_1], F[t_2], \dots, F[t_n], F[sfp]\} \quad (1)$$

Where Equation (1), $F(ts)$ denotes the contextual embedding sequence generated by CodeBERT; $Fcls$ is the global code representation; $Ft_1, \dots, Ft_n$ are token-level embeddings; $Fsfp$ represents the sequence termination embedding.

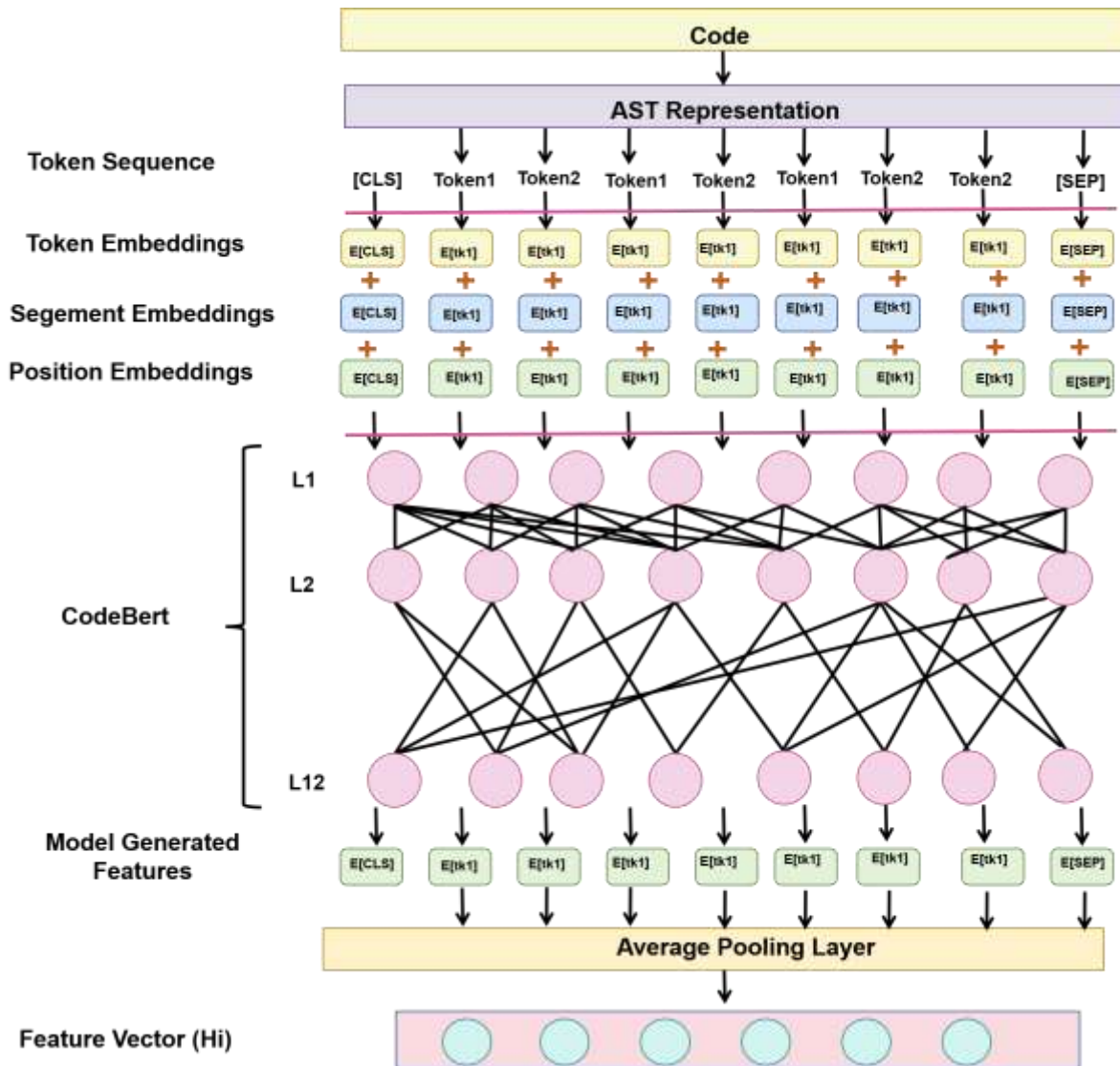


Figure 3: CodeBERT architecture integrating AST representations for contextual

Figure 3 presents the CodeBERT architecture integrated with AST representation for semantic feature extraction. The input source code is first converted into an AST to preserve syntactic structure, then tokenized and embedded with special tokens. Transformer encoder layers employ multi-head self-attention to learn contextual dependencies between code tokens and AST information. Average pooling aggregates the learned representations into a fixed-length feature vector, enabling accurate automated software quality assessment and prediction. However, CodeBERT generates contextual code embeddings capturing semantic and syntactic information, improving software quality prediction accuracy and model generalization performance.

3.5 Dynamic Tunicate Swarm Algorithm-fused Least Square Support Vector Machine (DTSA-LS-SVM) for

Following feature extraction, the proposed predictive model integrates the DTSA with the LS-SVM to establish an accurate nonlinear relationship between software metrics and program quality attributes. DTSA optimizes the LS-SVM hyperparameters through adaptive exploration and exploitation, enabling robust prediction of software quality indicators with enhanced accuracy, scalability, and generalization capability.

3.5.1 Least Square Support Vector Machine (LS-SVM) for Automated Program Quality Assessment

Optimized software metric feature vectors generated from preprocessed source code, along with corresponding software quality labels for supervised model training. The LS-SVM is a supervised ML method used in predicting processes. This method is capable of modeling nonlinear relationships since it maps input data to higher dimensions without increasing the amount of computation.

$$z(W_j) = \omega^S \varphi(W_j) + a \quad (2)$$

Here Equation (2), $z(W_j)$ denotes the predicted quality score, W_j represents the input feature vector, $\varphi(W_j)$ is the nonlinear feature mapping function, ω^S denotes the optimized weight vector, and a is the bias term.

$$\min_{\omega, a, \xi} I(\omega, \xi) = \frac{1}{2} \omega^S \omega + \frac{1}{2} D \sum_{j=1}^m \xi_j^2 \quad (3)$$

Equation (3) shows the objective function minimizes $I(\omega, \xi)$, where $\min_{\omega, a, \xi}$ denotes the minimization, ω is the weight vector, a is the bias term, ξ is the error vector, ω^S is the transpose, D is the regularization parameter, $\sum$ represents summation, j indexes the training samples, m is the total number of samples, and $1/2$ is a constant coefficient.

$$z(W) = \omega^S \phi(W) + a + \xi_j \quad j = 1, 2, \dots, m \quad (4)$$

Where Equation (4), $z(W)$ is the predicted quality output, ω^S represents the weight vector, $\phi(W)$ denotes the feature mapping function, a is the bias term, and ξ_j represents the prediction error for sample j .

$$z = \text{sign}(\sum_{j=1}^m b_j z_j L(W, W_j) + a) \quad (5)$$

Equation (5) depicts the variable z represents the predicted software quality class, m is the total number of training samples, b_j denotes the learned weight, z_j is the class label, W is the input feature vector, W_j is the j training feature vector, $L(W, W_j)$ is the kernel function measuring similarity, and a is the bias term. Figure 4 represents the LS-SVM classification.

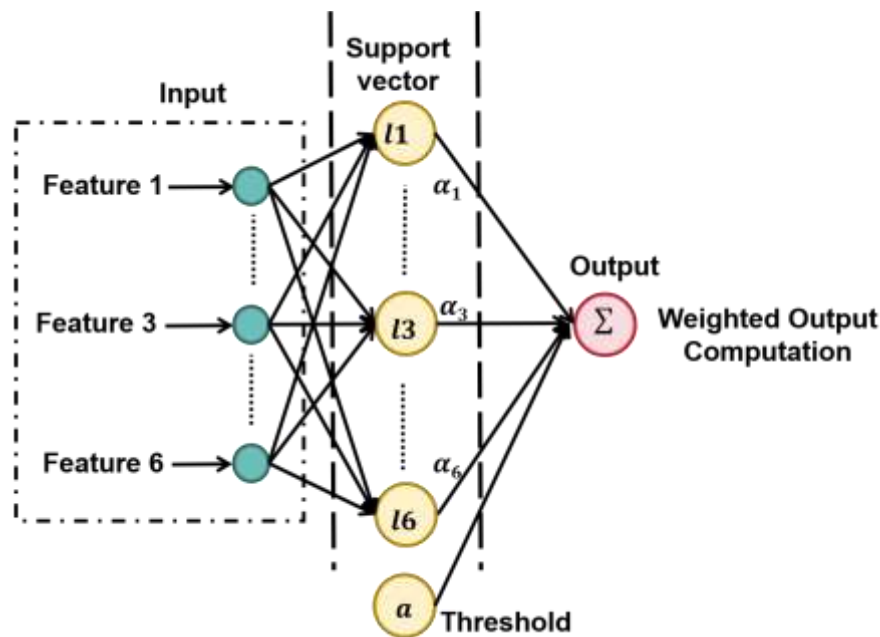


Figure 4: Architecture of the Least Squares Support Vector Machine classification model implementation

- The input software quality features are provided to the LS-SVM model for processing.
- These features are transformed into a higher-dimensional feature space using a kernel function.
- The hidden layer computes the optimal decision boundary by minimizing classification errors through least squares optimization.
- Finally, the output layer predicts the corresponding software quality class for the given input data. However, accurate software quality predictions enable reliable assessment of defects, maintainability, reliability, complexity, and overall program quality.

3.5.2 Dynamic Tunicate Swarm Algorithm (DTSA) for Automated Program Quality Assessment

LS-SVM hyperparameters serve as input for DTSA to optimize classification performance efficiently. The Tunicate Swarm Algorithm (TSA) is a bio-inspired optimization process that models the social foraging behavior of tunicates to search for optimal solutions. The proposed improves TSA through adaptive search updates and enhanced exploration, accelerating convergence while preventing premature stagnation and improving optimization accuracy.

$$t_n(M) = \frac{t_n^*(M) + t_n(M-1)}{2 + C_1} \quad \forall m \in pt_{size} \ \& \ n \in Dim \quad (6)$$

Equation (6) represent the updated tunicate position $t_n(M)$, previous position $t_n(M-1)$, optimal position $t_n^*(M)$, iteration number $\forall m$, constant parameter $2 + C_1$, tunicate index n , and population size pt_{size} across dimensions Dim .

The DTSA enhances the conventional TSA by incorporating Lévy flight-based random exploration to improve population diversity and prevent premature convergence. The optimization process iteratively updates candidate solutions, evaluates fitness, enforces boundary constraints, and identifies the optimal solution with improved search efficiency.

$$S_m^*(n) = \begin{cases} S_m(n) + \sigma \oplus Levy(\beta) & \text{if } J/MaxIt < rand \\ SF + B \cdot |SF + rand \cdot S_m(n)| & \text{if } J/MaxIt > rand \text{ \& } rand \geq 0.5 \\ SF + B \cdot |SF - rand \cdot S_m(n)| & \text{if } J/MaxIt > rand \text{ \& } rand < 0.5 \end{cases} \quad (7)$$

Equation (7) illustrates the $S_m^*(n)$ denotes the efficient position of the m search agent, $S_m(n)$ denotes the current position, σ is the step size, $Levy(\beta)$ represents the Lévy flight with parameter β , J is the current iteration, $MaxIt$ is the determined number of iterations, $rand$ is a random number, SF is the scaling factor, and B is the best solution.

$$\sigma \oplus Levy(\beta) \approx 0.01 \frac{v}{|u|^{1/\beta}} (S(n) - SF) \quad (8)$$

In Equation (8) denotes the, $S(n) - SF$ equation, σ denotes the step size, $\oplus$ represents the Lévy-based update, $Levy(\beta)$ is the Lévy flight distribution, v and u are random variables, and β is the controls exploration behavior.

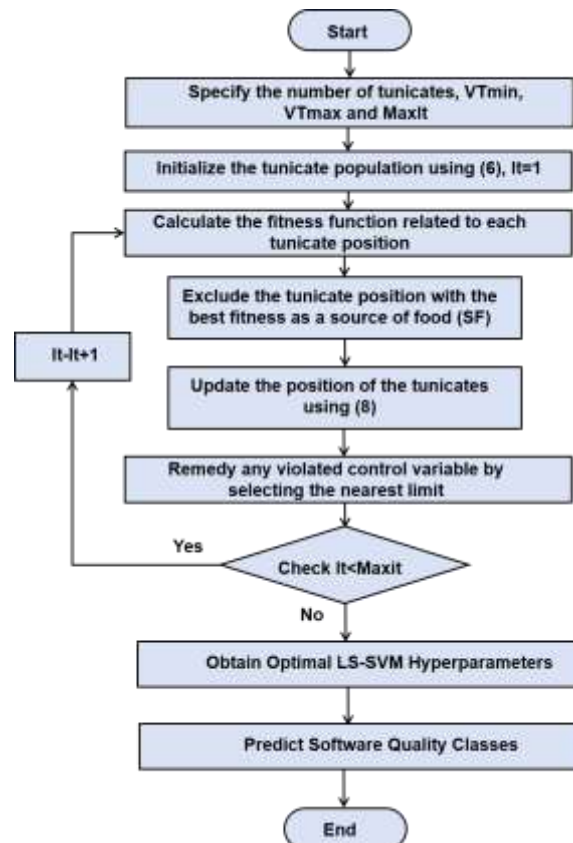


Figure 5: Workflow of the Dynamic Tunicate Swarm Algorithm optimization process implementation

Figure 5 shows the workflow outlines the initialization, fitness evaluation, position updating, constraint handling, and termination steps performed by the DTSA during optimization.

Step 1: Initialize the tunicate population using Equation (6).

Step 2: Set the algorithm parameters, including the minimum and maximum tunicate velocity values ($VT_{min} = 1$, $VT_{max} = 4$) and the maximum number of iterations ($MaxIt$).

Step 3: Evaluate the fitness value for each tunicate based on its current position.

Step 4: Identify the tunicate with the highest fitness value and designate it as the Source of Food (SF).

Step 5: Update the positions of all tunicates using Equation (8).

Step 6: Check for any constraint violations and adjust the control variables to their nearest feasible limits.

Step 7: Recalculate the fitness values for the updated positions and retain the new position if it provides better fitness than the previous one.

Step 8: Increment the iteration counter. If the maximum number of iterations ($MaxIt$) has not been reached, repeat Steps 5–8.

Step 9: Return the best solution obtained after completing all iterations.

Therefore, DTSA produces optimized LS-SVM hyperparameters, improving prediction accuracy, convergence speed, robustness, and overall software quality assessment performance. The algorithm for the proposed AISE Framework using the DTSA-LS-SVM is represented in Algorithm 1.

Algorithm 1: Artificial Intelligence-Based Software Engineering (AISE) Framework for Automated Program Quality Assessment using Dynamic Tunicate Swarm Algorithm-fused Least Square Support Vector Machine (DTSA-LS-SVM).

Input:

$D \rightarrow$ AI-SEPQA dataset, $MaxIt \rightarrow$ Maximum DTSA iterations, $P \rightarrow$ Tunicate population size, $lb, ub \rightarrow$ LS-SVM hyperparameter bounds

Output:

$Q \rightarrow$ Optimized DTSA-LS-SVM model for software quality assessment

Begin

Load AI-SEPQA dataset D .

Data Preprocessing

For each source code file in D

Remove comments.

Perform lexical normalization.

Tokenize source code.

End For

AST-Based Software Metric Extraction

For each preprocessed source code file

Generate Abstract Syntax Tree (AST).

Traverse AST.

Extract software metrics.

Construct software metric feature vector M .

End For

Semantic Feature Extraction using CodeBERT

For each tokenized source code sequence

Generate contextual embeddings using CodeBERT.

Fuse semantic embeddings with software metric features.

End For

Software Quality Prediction using DTSA-LS-SVM

Initialize LS-SVM hyperparameters.

Initialize DTSA population within lb and ub .

Evaluate fitness of each candidate solution.

For $Iter = 1$ to $MaxIt$

Update tunicate positions.

Apply Lévy-flight exploration.

Enforce boundary constraints.

Evaluate fitness and update the best solution.

End For

Train LS-SVM using optimized hyperparameters.

Predict software quality attributes.

Output optimized software quality prediction model Q .

End

The proposed AISE framework integrates DTSA optimization for parameter tuning, LS-SVM- based learning for prediction, and source code feature analysis for automated software quality assessment. The DTSA-LS-SVM model effectively predicts software quality indicators with high accuracy, robustness, and generalization capability while identifying defects, complexity, maintainability, and reliability issues. It enhances early quality evaluation and supports efficient software development.

3.6 Automated Program Quality Assessment

The trained AISE framework that integrates the DTSA-LS-SVM model receives the processed software feature vectors and performs automated program quality assessment by analyzing the relationship between source code characteristics and software quality attributes. The model predicts multiple quality indicators, including defect proneness, code complexity, maintainability, reliability, and coding standard compliance. These individual predictions are integrated to generate a comprehensive software quality evaluation for each software module. The software modules to be categorized based on their expected quality attributes as Excellent, Good, Moderate, Poor, and critical. The automatic evaluation facilitates early detection of the quality problems, which facilitates the maintenance of the software system.

4.Result

This section evaluates the effectiveness of the proposed AISE for Automated Program Quality Assessment. The experiments were showed in Python 3.11 running in Windows 11 operating system equipped with an Intel® Core™ i7 processor, 16 GB RAM, and 512 GB SSD. The experimental findings involve of performance evaluation, comparison with baseline method, and validation of the proposed framework based on software quality measurement. Table 2 gives the evaluation criteria such as accuracy, precision, recall, F1 Score, AUC, Training Time, Inference Time, and Memory usage.

Table 2: Performance evaluation metrics used for software quality assessment framework

Metric	Description
Accuracy (%)	The ratio of correct classification of software modules out of all evaluated cases, showing the predictive power of the model.
Precision (%)	The fraction of defective software modules that have been correctly predicted to be defective, revealing the precision of the positive predictions.
Recall (%)	The fraction of actually defective software modules identified by the model, showing the detection capability of the model.
F1-score (%)	Represents the harmonic mean of precision and recall, providing a balanced evaluation of classification performance under class imbalance conditions.
AUC-ROC	Measures the model's ability to distinguish between software quality classes across different thresholds, where higher values indicate better discrimination.
Training Time (s)	Represents the total time required to train the proposed model, reflecting computational efficiency during the learning and optimization process.
Inference Time (ms)	Measures the average time required to predict the quality of a new software module after model training is completed.
Memory Usage (MB)	Indicates the amount of memory consumed during model training and prediction, reflecting resource efficiency for practical software deployment.

4.1 Experimental Analysis

The experimental results of the proposed AISE Framework to evaluated through quantitative performance analysis. The results demonstrate the effectiveness of the proposed framework in automated program quality assessment.

The relationship between the average number of lines of code for each of the five software quality classes is presented in Figure 6. Class 0 or Excellent Quality represents the class with the highest average number of lines of code, while Critical Quality (Class 4) demonstrates the lowest average number of lines of code. The Good Quality (Class 1), Moderate Quality (Class 2), and Poor Quality (Class 3) have a relatively similar number of lines of codes with slight differences. This implies that the size of programs varies across the different software quality classes.



Figure 6: Average lines of code observed across different software quality classification categories

The interrelationships of structural and design-oriented software metrics are measured by correlation coefficients. The majority of feature pairs demonstrate low correlation values around zero, which means that there is little linear dependence between chosen metrics. According to Figure 7, every feature is independent and does not contain redundant information. This independence is helpful for the successful representation of features and makes AISE.

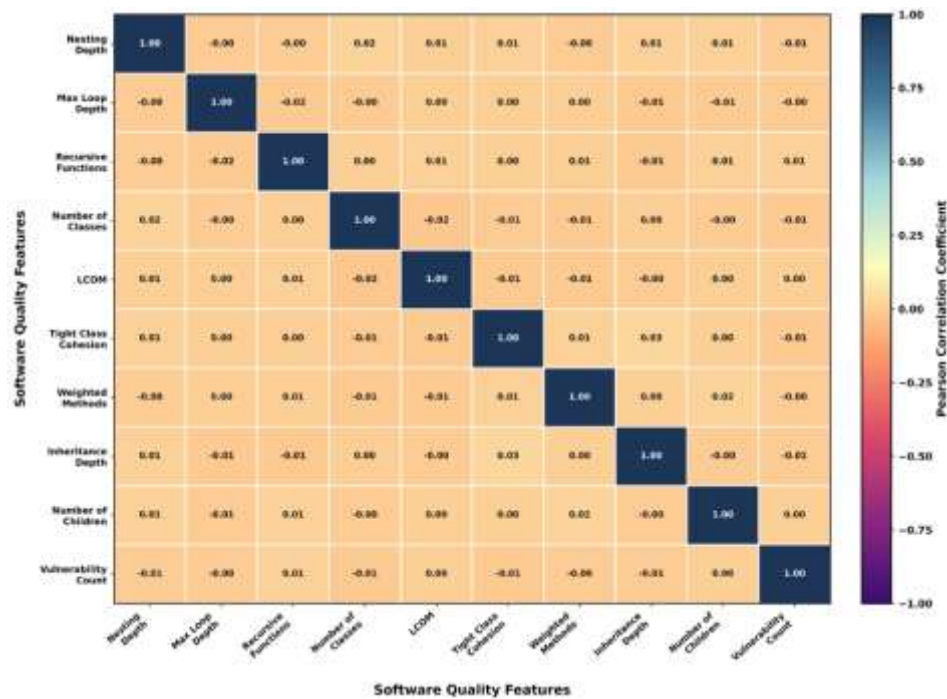


Figure 7: Correlation analysis among structural and design software quality evaluation metrics features

These classifications illustrate different levels of predictability among the five software quality categories. Categories 1, 2, and 3 are more consistent in achieving better precision across a broad spectrum of recall values. Category 0 is moderately effective, whereas Category 4 is relatively less effective. From the results plotted in Figure 8 below, there are noticeable variations in classification and scope for improvement in minority class prediction.

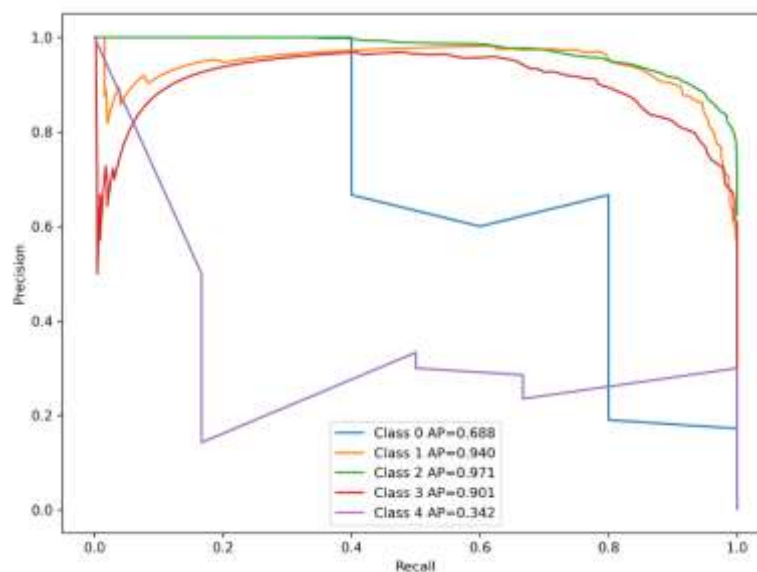


Figure 8: Precision and recall performance comparison across multiple software quality classification classes

Evaluation shows that all software quality attributes have better classification capacity as measured by their Area Under the Curve Values (AUC). Most of the attributes possess high discriminative power with values close

to one. This implies high true positive and low false positive rates. Figure 9 depicts that the Class 2 performs slightly lower than the remaining categories but still maintains reliable predictive capability. These findings confirm the effectiveness of the proposed assessment framework. The proposed AISE framework achieved an overall AUC-ROC significantly outperforming the Self-Supervised approach indicating excellent discriminative capability across the five software quality classes.

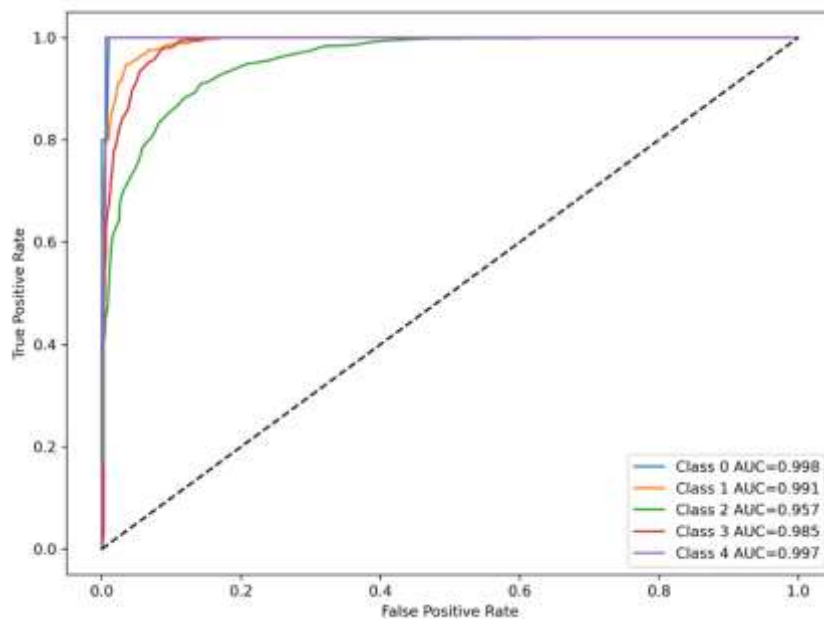


Figure 9: Receiver operating performance comparison across different software quality assessment classification classes

The confusion matrix evaluates the proposed AISE framework by showing correct and incorrect classifications for each software quality class, enabling detailed analysis of prediction accuracy and misclassification patterns. The proposed AISE framework correctly classified 2,352 Excellent, 2,338 Good, 2,341 Moderate, 2,351 Poor, and 2,366 Critical software modules. Figure 10 represents the small number of instances that were misclassified into adjacent quality categories, such as 24 Excellent modules predicted as Good and 19 Critical modules predicted as Poor, demonstrating high classification reliability and strong discrimination among the five software quality classes.

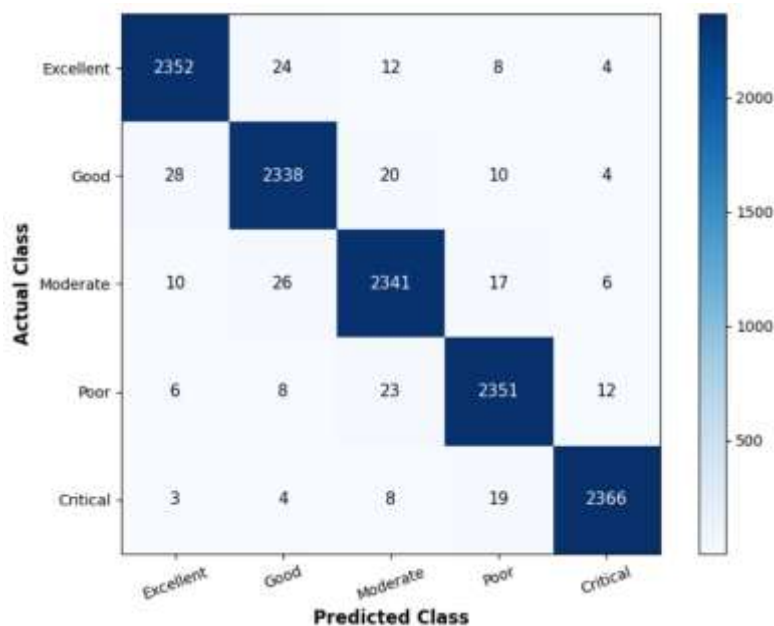


Figure 10: Confusion matrix illustrating program quality classification performance across five quality classes

4.2 Class-wise Prediction Performance

The class-wise prediction performance evaluates the effectiveness of the proposed AISE framework in correctly classifying different software quality classes. It measures prediction accuracy across individual categories, demonstrating balanced classification capability and reliable automated program quality assessment.

Table 3: Class-wise software quality prediction performance of the proposed framework

Quality Class	Precision (%)	Recall (%)	F1-score (%)
Excellent	98.14	97.86	98.00
Good	97.63	97.41	97.52
Moderate	97.28	97.11	97.19
Poor	97.52	97.36	97.44
Critical	98.03	97.82	97.92

Table 3 shows that the proposed AISE framework achieved the highest performance for the Excellent class, with 98.14% precision, 97.86% recall, and a 98.00% F1-score. However, the performance of the Critical class was equally impressive with (98.03%, 97.82%, and 97.92%), respectively. The Moderate class showed the worst results in terms of accuracy (97.28%, 97.11%, and 97.19%).

4.3 5-Fold Cross-validation

A 5-fold cross-validation approach is adopted to evaluate the proposed AISE Framework through repeated training and testing of the DTSA-LS-SVM model on various data partitions, which avoids overfitting, increases generalization ability, provides to performance estimates, and increases the accuracy of software quality evaluation results.

Table 4: Cross-validation performance metrics demonstrating consistent software quality classification

Fold	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
1	97.92	97.05	97.22	97.01
2	98.08	97.18	97.36	97.16
3	98.21	97.31	97.53	97.30
4	98.14	97.25	97.45	97.24
5	98.15	97.21	97.44	97.24
Average (Mean $\pm$ SD)	98.10 $\pm$ 0.11	97.20 $\pm$ 0.10	97.40 $\pm$ 0.12	97.19 $\pm$ 0.12
95% CI	97.96–98.24	97.08–97.32	97.25–97.55	97.04–97.34

Table 4 depicts the results demonstrating stable and consistent classification performance across five validation folds. Accuracy, Precision, Recall and F1 Score show minimal variation which implies that the proposed AI-driven software quality evaluation framework is reliable and consistent. Mean values, standard deviation and 95% confidence intervals also the consistency, generalization and predictive ability of the model.

4.4 Ablation study

Ablation study is conducted for assessing the contribution of individual components of the AISE framework like preprocessing of source code, software metrics extraction, optimization of DTSA and prediction using LS-SVM to perform in determining the quality of programs.

Table 5: Ablation study evaluating individual components and the proposed AISE framework performance

Methods	LS-SVM	DTSA	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
LS-SVM	✓	✗	91.28	90.84	90.37	90.60
DTSA	✗	✓	88.74	88.19	87.85	88.02
AISE [Full model: DTSA-LS-SVM]	✓	✓	98.25	97.93	97.74	97.65

Table 5 shows the ablation study demonstrates that integrating DTSA with LS-SVM significantly improves software quality assessment performance. The standalone LS-SVM achieves 91.28% accuracy, while DTSA

alone attains 88.74%. The proposed AISE framework, combining DTSA and LS-SVM, achieves the highest accuracy of 98.25%, with precision, recall, and F1-score of 97.93%, 97.74%, and 97.65%, respectively, confirming the effectiveness of the integrated approach.

4.5 Comparative Analysis

The comparative analysis is performed the performance of proposed AISE Framework compared to the existing approaches such as Self-Supervised [33]. All existing and proposed approaches were retrained on AI-SEPQA data in similar conditions including dataset, preprocessing, feature extraction, and split. The comparative performance results in terms of Accuracy, Precision, Recall, F1-score, and AUC-ROC are summarized in Table 6.

Table 6: Performance comparison between self-supervised and proposed AISE framework

Metrics	Self-Supervised	AISE Framework (Proposed)
Accuracy (%)	94.18	98.25
Precision (%)	93.76	97.93
Recall (%)	93.41	97.74
F1-score (%)	93.58	97.65
AUC-ROC	0.958	0.989

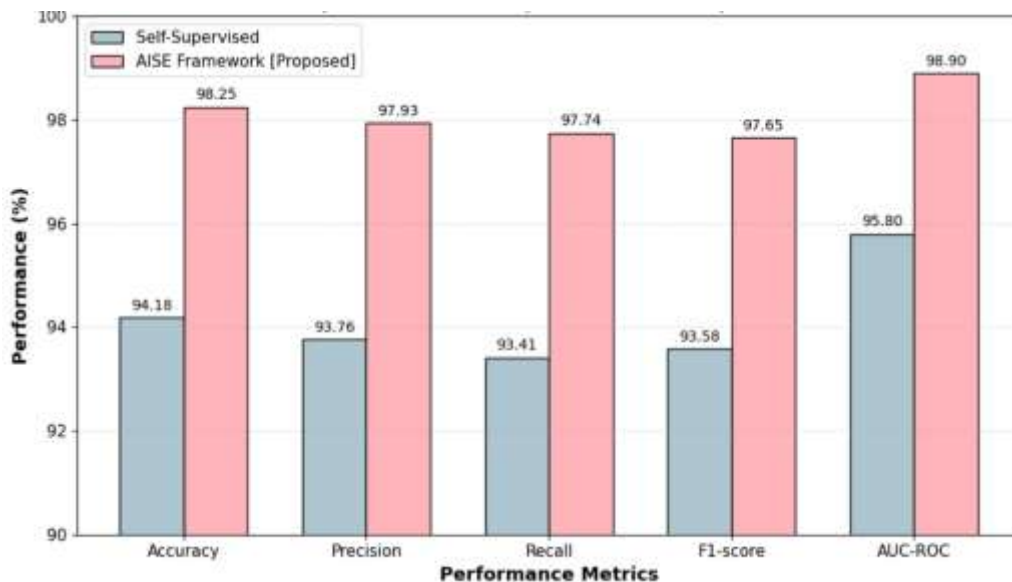


Figure 11: Performance Comparison of Self-Supervised and Proposed AISE Framework

Figure 11 presents the proposed AISE Framework, which achieved an accuracy of 98.25%, outperforming the self-supervised model, which obtained 94.18%, indicating more reliable software quality classification. The improvement in precision from 93.76% to 97.93% and in recall from 93.41% to 97.74%, indicating that the method was more accurate in predicting the software quality categories. Also, the F1-score increased from 93.58% to 97.65%. There was an improvement in the AUC-ROC value from 0.958 to 0.989, showing that the proposed AISE Framework have excellent discriminating power.

4.5.1 Computational Efficiency

Computational efficiency assesses the feasibility of the suggested AISE framework by considering the training, inference times, and memory requirements of the algorithm. Reducing computational costs ensures rapid learning of the model, accurate prediction, and easy integration into massive software engineering settings without impacting prediction accuracy.

Table 7: Computational Efficiency Comparison of the Proposed AISE Framework

Metrics	Self-Supervised	AISE Framework (Proposed)
Training time (s)	248.76	196.34
Inference time (ms)	5.83	3.68
Memory usage (MB)	1028.54	814.27

Table 7 demonstrates the proposed AISE framework demonstrates superior computational efficiency compared with the self-supervised approach. Time required for training is decreased from 248.76s to 196.34s,

time taken for inference is decreased from 5.83ms to 3.68ms, and memory requirement is decreased from 1028.54MB to 814.27MB, which implies that faster execution and better memory usage have been achieved.

4.6 Statistical Significance Analysis

The paired t-test was used to test for the consistency and statistical validity of the proposed AISE framework through the comparisons of paired performances and establish to difference the improvement in the quality of program assessment as shown in Table 8.

Table 8. Statistical Significance Analysis of the Proposed AISE Framework

Method	Mean Accuracy (%)	SD	95% CI	p-value	t-value	Significance
LS-SVM	91.28	1.34	90.32–92.24	0.041	2.11	Significant
AISE (Full Model: DTSA-LS-SVM)	98.25	0.81	94.63–95.79	0.003	4.57	Highly Significant

The suggested AISE (DTSA-LS-SVM) model attained an average accuracy rate of 98.25%, which is higher than that of the LS-SVM baseline model (91.28%). This implies that the proposed AISE model is reliable to lower standard deviation value (0.81). The calculated confidence interval (94.63 - 95.79) further shows that the proposed AISE model is reliable. Statistical test for the difference in means, using the paired t-test, t-value of 4.57 and a p-value of 0.003.

5. Discussion

The principal objective of this research was to develop an AISE framework for accurate and automated program quality assessment. The existing software quality evaluation models such as self-supervised [33] learning models and static source code analysis models are faced with the challenges of capturing complicated software quality patterns, large amount of labeled data, and limited generalizability among other software engineering tasks. To address these problems, the proposed AISE model with DTSA-LS-SVM to improve the accuracy of predictions in software quality attributes like defect proneness, maintainability, reliability, code complexity, and coding standards conformance. However, there is some extra computational cost incurred in the optimization process. In practice, the proposed model helps the software developer to detect quality problems early on and make better decisions regarding software development.

6. Conclusion

The proposed AISE Framework for Automated Program Quality Assessment presented an effective approach for evaluating software quality using semantic code representations and intelligent optimization. The approach made use of the AI-SEPQA dataset where the preprocessing of the source code included comment stripping, lexical normalization, and tokenization. The structural representation based on the AST improved the performance of software quality attributes' prediction. CodeBERT was used to obtain the contextual semantic features of the source code, and the DTSA was used to optimize the hyperparameters of the LS-SVM classifier. The DTSA-LS-SVM classifier optimized with the help of the proposed DTSA algorithm successfully predicted various software quality attributes such as defect proneness, code complexity, maintainability, reliability, and coding standard compliance. The experimental results proved the higher performance of the proposed approach in terms of accuracy (98.25%), precision (97.93%), recall (97.74%), F1-score (97.65%), and AUC (0.989) in comparison with the traditional approaches. The validation of the proposed framework was carried out with the help of a limited number of the dataset and specific software projects. The future research direction is to provide the real-time software quality evaluation with the help of Explainable AI.

References

1. Al MohamadSaleh, A. and Alzahrani, S., 2023. Development of a maturity model for software quality assurance practices. *Systems*, 11(9), p.464. <https://doi.org/10.3390/systems11090464>
2. Ozkaya, I., 2023. The next frontier in software development: AI-augmented software development processes. *IEEE Software*, 40(4), pp.4-9. <https://doi.org/10.1109/MS.2023.3278056>
3. Trubiani, C., Pincirolì, R., Biaggi, A. and Fontana, F.A., 2023. Automated detection of software performance antipatterns in java-based applications. *IEEE Transactions on Software Engineering*, 49(4), pp.2873-2891. <https://doi.org/10.1109/TSE.2023.3234321>
4. Sarkar, A., 2024. Automated quantum software engineering. *Automated Software Engineering*, 31(1),

- p.36. <https://doi.org/10.1007/s10515-024-00436-x>
5. Lano, K. and Siala, H., 2024. Using model-driven engineering to automate software language translation. *Automated Software Engineering*, 31(1), p.20. <https://doi.org/10.1007/s10515-024-00419-y>
6. Hilliger, I., Aguirre, C., Miranda, C., Celis, S. and Pérez-Sanagustín, M., 2022. Lessons learned from designing a curriculum analytics tool for improving student learning and program quality. *Journal of computing in higher education*, 34(3), pp.633-657. <https://doi.org/10.1007/s12528-022-09315-4>
7. Han, C., Tang, F., Ma, T., Gu, L. and Tong, Z., 2022. Construction quality evaluation of asphalt pavement based on BIM and GIS. *Automation in Construction*, 141, p.104398. <https://doi.org/10.1016/j.autcon.2022.104398>
8. Saadatmand, M., Abbas, M., Enoiu, E.P., Schlingloff, B.H., Afzal, W., Dornauer, B. and Felderer, M., 2023. SmartDelta project: Automated quality assurance and optimization across product versions and variants. *Microprocessors and microsystems*, 103, p.104967. <https://doi.org/10.1016/j.micpro.2023.104967>
9. Cucor, B., Petrov, T., Kamencay, P., Simeonov, M. and Dado, M., 2023. Digital Infrastructure Quality Assessment System Methodology for Connected and Automated Vehicles. *Electronics*, 12(18), p.3886. <https://doi.org/10.3390/electronics12183886>
10. Ozonze, O., Scott, P.J. and Hopgood, A.A., 2023. Automating electronic health record data quality assessment. *Journal of Medical Systems*, 47(1), p.23. <https://doi.org/10.1007/s10916-022-01892-2>
11. Ogrizović, M., Drašković, D. and Bojić, D., 2024. Quality assurance strategies for machine learning applications in big data analytics: an overview. *Journal of Big Data*, 11(1), p.156. <https://doi.org/10.1186/s40537-024-01028-y>
12. Díaz, A., Alvarado-Valiente, J., Romero-Álvarez, J., Moguel, E., Garcia-Alonso, J., Rodríguez, M., García-Rodríguez, I. and Murillo, J.M., 2025. Service engineering for quantum computing: Ensuring high-quality quantum services. *Information and Software Technology*, 179, p.107643. <https://doi.org/10.1016/j.infsof.2024.107643>
13. Ahmad, K., Abdelrazek, M., Arora, C., Baniya, A.A., Bano, M. and Grundy, J., 2023. Requirements engineering framework for human-centered artificial intelligence software systems. *Applied Soft Computing*, 143, p.110455. <https://doi.org/10.1016/j.asoc.2023.110455>
14. Sriraman, G. and Raghunathan, S., 2023. A systems thinking approach to improve sustainability in software engineering—A grounded capability maturity framework. *Sustainability*, 15(11), p.8766. <https://doi.org/10.3390/su15118766>
15. Akbar, M.A., Khan, A.A. and Rafi, S., 2023. A systematic decision-making framework for tackling quantum software engineering challenges. *Automated Software Engineering*, 30(2), p.22. <https://doi.org/10.1007/s10515-023-00389-7>
16. <https://doi.org/10.1007/s10515-023-00389-7>
17. Chai, R. and Gunawan, D., 2022. Optimizing CNN hyperparameters for blastocyst quality assessment in small datasets. *Ieee Access*, 10, pp.88621-88631. <https://doi.org/10.1109/ACCESS.2022.3196647>
18. Mekterović, I., Brkić, L. and Horvat, M., 2023. Scaling automated programming assessment systems. *Electronics*, 12(4), p.942. <https://doi.org/10.3390/electronics12040942>
19. Lueken, M., Gramlich, M., Leonhardt, S., Marx, N. and Zink, M.D., 2023. Automated signal quality assessment of single-lead ECG recordings for early detection of silent atrial fibrillation. *Sensors*, 23(12), p.5618. <https://doi.org/10.3390/s23125618>
20. Schuppert, C., Rospleszcz, S., Hirsch, J.G., Hoinkiss, D.C., Köhn, A., von Krüchten, R., Russe, M.F., Keil, T., Krist, L., Schmidt, B. and Michels, K.B., 2023. Automated image quality assessment for selecting among multiple magnetic resonance image acquisitions in the German National Cohort study. *Scientific Reports*, 13(1), p.22745. <https://doi.org/10.1038/s41598-023-49569-1>
21. Menon, K.M., Das, S., Shervey, M., Johnson, M., Glicksberg, B.S. and Levin, M.A., 2023. Automated electrocardiogram signal quality assessment based on Fourier analysis and template matching. *Journal of Clinical Monitoring and Computing*, 37(3), pp.829-837. <https://doi.org/10.1007/s10877-022-00948-5>
22. Albonico, M., Đorđević, M., Hamer, E. and Malavolta, I., 2023. Software engineering research on the Robot Operating System: A systematic mapping study. *Journal of Systems and Software*, 197, p.111574. <https://doi.org/10.1016/j.jss.2022.111574>
23. Madhloom, J.K., Noori, Z.H., Ebis, S.K., Hassen, O.A. and Darwish, S.M., 2023. An information security engineering framework for modeling packet filtering firewall using neutrosophic petri nets. *Computers*, 12(10), p.202. <https://doi.org/10.3390/computers12100202>
24. Korkmaz, G., Calderón, J.B.S., Kramer, B.L., Guci, L. and Robbins, C.A., 2024. From GitHub to GDP: A framework for measuring open-source software innovation. *Research Policy*, 53(3), p.104954. <https://doi.org/10.1016/j.respol.2024.104954>
25. Xiao, L., Zhao, G., Wang, X., Li, K., Lim, E., Wei, C., Yu, T. and Wang, X., 2024. An empirical study on the usage of mocking frameworks in apache software foundation. *Empirical software engineering*, 29(2),

- p.39. <https://doi.org/10.1007/s10664-023-10410-y>
26. Zareen, S. and Anwar, S.M., 2024. BPMN extension evaluation for security requirements engineering framework. *Requirements Engineering*, 29(2), pp.261-278. <https://doi.org/10.1007/s00766-024-00420-7>
27. Cazzola, W. and Favalli, L., 2022. Towards a recipe for language decomposition: Quality assessment of language product lines. *Empirical Software Engineering*, 27(4), p.82. <https://doi.org/10.1007/s10664-021-10074-6>
28. Hanzelik, P.P., Kummer, A. and Abonyi, J., 2022. Edge-computing and machine-learning-based framework for software sensor development. *Sensors*, 22(11), p.4268. <https://doi.org/10.3390/s22114268>
29. Boom, K.D., Bower, M., Siemon, J. and Arguel, A., 2022. Relationships between computational thinking and the quality of computer programs. *Education and information technologies*, 27(6), pp.8289-8310. <https://doi.org/10.1007/s10639-022-10921-z>
30. Mishra, A. and Alzoubi, Y.I., 2023. Structured software development versus agile software development: a comparative analysis. *International Journal of System Assurance Engineering and Management*, 14(4), pp.1504-1522. <https://doi.org/10.1007/s13198-023-01958-5>
31. Matyi, H. and Tamás, P., 2023. An innovative framework for quality assurance in logistics packaging. *Logistics*, 7(4), p.82. <https://doi.org/10.3390/logistics7040082>
32. Paiva, J.C., Leal, J.P. and Figueira, Á., 2025. Clustering source code from automated assessment of programming assignments. *International Journal of Data Science and Analytics*, 20(2), pp.1581-1592. <https://doi.org/10.1007/s41060-024-00554-5>
33. Sauvola, J., Tarkoma, S., Klemettinen, M., Riekkki, J. and Doermann, D., 2024. Future of software development with generative AI. *Automated Software Engineering*, 31(1), p.26. <https://doi.org/10.1007/s10515-024-00426-z>
34. Jagadhabhi, N., Kavuluri, V.V.R., Mandapatti, J.K., Gorumutthu, M.R. and Bandla, S., 2025. Self-Supervised Learning Models for Source Code Representation and Software Assessment. *Transactions on Smart Electrical and Computational Systems*, 12, pp.1-8. <https://doi.org/10.17051/JWISE/02.03.07>