STATISTICS, OPTIMIZATION AND INFORMATION COMPUTING
Stat., Optim. Inf. Comput., Vol. **x**, Month 202x, pp 0–25.
Published online in International Academic Press (www.IAPress.org)

Attention-Guided Graph Neural Networks with Adaptive Feature Selection for Explainable Software Defect Prediction

Haneen Hussein Shehadeh[1,*], Njood Aljarrah[1], Razan Ali Obeidat[1], Ashraf A. Abu-Ein[2], Mohammed Tawfik[1,*]

[1] *Department of Computer Science, Faculty of Information Technology, Ajloun National University, Jordan*
[2] *Department of Computer and Cyber Security, Faculty of Information Technology, Jadara University, Irbid, Jordan*

Abstract Software defect prediction plays a critical role in quality assurance, yet existing approaches face significant limitations in capturing complex inter-module dependencies while providing interpretable predictions essential for practical deployment. Traditional machine learning methods rely on handcrafted features that fail to model structural relationships within software systems, while recent deep learning approaches lack the explainability required for industrial adoption. This paper proposes an attention-guided graph neural network framework that integrates multi-algorithm feature selection with graph-based structural modeling to achieve superior defect prediction performance while maintaining comprehensive interpretability.Our framework combines five complementary feature selection methods (SHAP importance, permutation importance, CMA-ES optimization, Boruta selection, and mRMR analysis) to identify the most predictive software metrics, constructs similarity-based graphs to capture inter-module relationships, and employs multi-head Graph Attention Networks (GATv2) to learn defect patterns through attention mechanisms. The approach incorporates multi-modal explainability through attention weight visualization, LIME attributions, and feature importance analysis to provide actionable insights for software practitioners.Comprehensive evaluation on NASA PROMISE and GHPR datasets demonstrates substantial performance improvements, achieving mean F1-scores of 95.52% and 91.6% respectively, representing gains of 2.07% to 6.62% over state-of-the-art methods including CodeBERT, standard GAT, and traditional machine learning approaches. Ablation studies confirm that graph construction contributes most significantly to performance improvements (+3.55% F1), while feature importance analysis reveals that static invocations dominate modern defect patterns, providing specific architectural guidance for code quality improvement.The framework maintains computational efficiency suitable for continuous integration pipelines while scaling effectively from small projects to enterprise systems. Our contributions advance both theoretical understanding of software defect patterns through attention mechanism analysis and practical capabilities for industrial defect prediction through comprehensive explainability integration.

Keywords Software Defect Prediction, Graph Neural Networks, Attention Mechanisms, Explainable AI, Feature Selection, Graph Attention Networks, XAI, Software Engineering, Code Quality, Software Quality Assurance

DOI:10.19139/soic-2310-5070-2906

1. Introduction

Software defect prediction (SDP) has emerged as a critical component of software quality assurance, enabling development teams to identify potentially defective modules before deployment and optimize

*Correspondence to: Haneen Hussein Shehadeh (Email: haneenshehadeh1993@yahoo.com), Mohammed Tawfik (Email: M.Tawfik@anu.edu.jo), Department of Computer Science, Faculty of Information Technology, Ajloun National University, P.O.43, Ajloun-26810, JORDAN.

ISSN 2310-5070 (online) ISSN 2311-004X (print)

testing resource allocation [1]. The increasing complexity of modern software systems and the demand for rapid development cycles have intensified the need for accurate and interpretable defect prediction methodologies [2]. However, traditional software defect prediction approaches face significant challenges in handling the intricate relationships within software systems while providing meaningful explanations for their predictions.

Existing machine learning approaches for software defect prediction predominantly rely on handcrafted features and traditional metrics that fail to capture the complex structural and semantic dependencies inherent in software code [3]. Recent advances in deep learning have shown promise in automatically extracting meaningful features from source code, with attention-based neural networks demonstrating particular effectiveness in identifying critical code components for defect prediction [4]. However, most deep learning-based approaches suffer from limited explainability, creating a significant barrier to their adoption in industrial settings where understanding the reasoning behind predictions is crucial for decision-making [5].

Graph Neural Networks (GNNs) have gained considerable attention in software engineering tasks due to their ability to effectively model the structural relationships within code through abstract syntax trees and dependency networks [6]. GNNs can process the hierarchical structure of source code and capture both syntactic and semantic information, leading to more accurate defect prediction compared to traditional feature-based methods [3]. Despite these advantages, existing GNN-based approaches for software defect prediction often lack sophisticated attention mechanisms to focus on the most relevant code components and fail to provide comprehensive explanations for their predictions.

The challenge of feature selection remains a critical concern in software defect prediction, as datasets often contain irrelevant or redundant features that can negatively impact model performance [1]. Traditional feature selection techniques may not adequately address the complex interdependencies between features in software metrics, necessitating more advanced optimization-driven approaches [7]. Furthermore, the integration of feature selection with graph neural networks requires careful consideration of both structural and feature-level importance to maintain the integrity of the graph representation while optimizing predictive performance.

Explainable Artificial Intelligence (XAI) has become increasingly important in software defect prediction, as practitioners require insights into why certain modules are predicted as defective to make informed decisions about testing priorities and code reviews [8]. Current explainability approaches in software defect prediction primarily focus on identifying important features or code segments but often fail to provide comprehensive explanations that consider both local and global model behavior [9]. The integration of attention mechanisms with graph neural networks offers a promising avenue for enhancing explainability by highlighting the most influential code structures and their relationships in the prediction process.

Recent studies have demonstrated the effectiveness of attention mechanisms in various software engineering tasks, particularly in code analysis and defect prediction [4, 10]. Dual-attention mechanisms have shown particular promise in capturing complex semantic information while emphasizing the most crucial features for prediction [7]. However, the combination of attention-guided graph neural networks with adaptive optimization-driven feature selection for explainable software defect prediction remains largely unexplored, presenting an opportunity for significant methodological advancement.

To address these challenges, this paper proposes a novel attention-guided graph neural network framework with adaptive optimization-driven feature selection specifically designed for explainable software defect prediction. Our approach integrates several key innovations: (1) an attention mechanism that dynamically focuses on the most relevant graph nodes and edges representing critical code components, (2) an adaptive optimization algorithm for intelligent feature selection that considers both individual feature importance and structural dependencies, (3) a comprehensive explainability framework that provides both local and global explanations for prediction decisions, and (4) empirical validation on multiple benchmark datasets demonstrating superior performance compared to state-of-the-art methods.

The main contributions of this work are: (1) development of an attention-guided graph neural network architecture that effectively captures structural and semantic relationships in software code while maintaining interpretability, (2) introduction of an adaptive optimization-driven feature selection mechanism that dynamically adjusts to dataset characteristics and maintains graph structural integrity, (3) design of a comprehensive explainability framework that provides actionable insights for software practitioners, and (4) extensive experimental evaluation demonstrating improved prediction accuracy and explainability compared to existing approaches. **Thesis & Paper Organization.** Our central thesis is that an attention-guided graph neural network, coupled with an adaptive multi-algorithm feature-selection ensemble, delivers accurate and practically explainable software defect prediction across legacy (NASA PROMISE) and modern (GHPR) datasets. The remainder of this paper is organized as follows. Section 2 surveys related work on traditional ML, deep learning, graph neural networks, pre-trained transformers, feature selection, and ensemble methods for SDP. Section 3 details the proposed methodology, including datasets, metrics extraction, robust preprocessing, the feature-selection ensemble, graph construction, the GATv2 architecture, training, and the multi-modal explainability framework. Section 4 defines the evaluation metrics. Section 5 reports experimental results, baseline comparisons, ablation studies, training dynamics, and feature-importance analyses. Section 6 discusses implications, insights, and limitations. Section 7 concludes and outlines future research directions.

2. Related Work

2.1. Traditional Machine Learning Approaches for Software Defect Prediction

Traditional machine learning techniques have formed the foundation of software defect prediction research for over two decades. Iqbal et al. [15] conducted a comprehensive performance analysis of ten machine learning classification techniques including Naïve Bayes, Multi-Layer Perceptron, Radial Basis Function, Support Vector Machine, K Nearest Neighbor, kStar, One Rule, PART, Decision Tree, and Random Forest on twelve NASA datasets. Their study revealed that RBF, MLP, SVM, and RF achieved higher accuracy across multiple datasets, while Random Forest demonstrated superior ROC performance. However, they highlighted the sensitivity of precision, recall, and F-measure to class imbalance issues, emphasizing the need for careful evaluation metrics selection.

Khalid et al. [16] addressed the class imbalance problem by combining K-means clustering for class-label discovery with variance-inflation-factor and correlation-based feature selection, retaining only the ten most informative attributes from the original 22 features. Their Particle-Swarm-Optimized versions of SVM, Naïve Bayes, and Random Forest, combined with a Stacking ensemble, achieved remarkable results on the NASA CM1 dataset, with optimized SVM attaining 99.80% accuracy. Similarly, Thomas and Kaliraj [17] enhanced Random Forest-based approaches by combining Random-Forest-based feature selection with SMOTE-based class-imbalance handling and randomized-grid-search optimization, achieving 82.96% accuracy on the NASA JM1 dataset.

2.2. Deep Learning and Neural Network Approaches

The emergence of deep learning has revolutionized software defect prediction by enabling automatic feature extraction from raw code representations. Batool and Khan [18] compared three deep learning models—LSTM, Bidirectional LSTM (BILSTM), and Radial Basis Function Network—on large-scale datasets containing up to 88,672 instances. Their systematic hyperparameter tuning revealed that BILSTM achieved the best accuracy (93.75%), followed by LSTM (93.53%), significantly outperforming traditional ML baselines.

Kumar and Sagar [19] proposed a sophisticated hybrid intelligent model that addresses multiple challenges simultaneously: class imbalance through combined SMOTE and Random Under-Sampling, feature selection via Enhanced Flamingo Search Algorithm, and classification through a dual-attention one-dimensional residual autoencoder integrated with Extreme Gradient Boosting. Their approach achieved exceptional accuracies ranging from 98.65% to 99.56% across multiple PROMISE datasets, with hyperparameters optimized using the COOT optimization algorithm.

Malhotra et al. [20] advanced the field by introducing an attention-augmented 1-dimensional Convolutional Neural Network classifier combined with a multi-filter wrapper feature selection engine. Their CNN architecture incorporated coordinate attention blocks to focus on defect-indicative regions, achieving the highest average scores across 17 open-source projects with F-measure values ranging from 0.794 to 0.855 and AUC values from 0.881 to 0.938, statistically outperforming 18 state-of-the-art techniques.

2.3. Graph Neural Network Approaches

Graph neural networks have emerged as a powerful paradigm for software defect prediction due to their ability to model structural relationships within software systems. Xu et al. [21] introduced ACGDP, an Augmented Code Property Graph-based system that enhances defect representation by integrating syntax, semantics, control, and data dependencies. Their approach proposed Graph of Defect Region Candidates extraction method to isolate potential defect areas, followed by specialized GNN-based prediction models including Graph Convolutional Networks, Graph Isomorphism Networks, Simplified Graph Convolution, Graph Attention Networks, and GraphSAGE, optimized using Bayesian hyperparameter tuning.

Xu et al. [22] developed GNN-DP, a novel approach combining semantics and context features of code via graph representation learning. Their method extracted Abstract Syntax Trees from Java source code, pruned them using the Louvain algorithm to isolate defect-related subtrees, and enriched nodes with code concepts derived via LDA topic modeling and Word2Vec embeddings. Using Simplified Graph Convolution for graph-level classification on the GHPR dataset, GNN-DP achieved remarkable improvements: 19.4% in precision, 30.7% in F1-score, and 57.7% in AUC compared to traditional baselines.

Soomro et al. [23] focused on object-oriented systems by fusing static metrics with dynamic test-coverage features to predict run-time object-communication errors in multi-level inheritance systems. Their Object Communication Error Learner trained on the GHPR dataset of 6,053 GitHub projects achieved 78% precision, 74.9% recall, 76.4% F1-score, and 89% ROC-AUC, demonstrating the effectiveness of combining static and dynamic features in graph-based representations.

2.4. Pre-trained Transformer Models

The advent of pre-trained transformer models has opened new avenues for software defect prediction by leveraging large-scale code corpora. Pan et al. [24] pioneered the use of CodeBERT for cross-version and cross-project software defect prediction, introducing four variants: CodeBERT-NT, CodeBERT-PT, CodeBERT-PS, and CodeBERT-PK. Their sentence- and keyword-based prediction patterns jointly fed source code with declarative sentences or bug-related keywords into the model. The pre-trained CodeBERT-PT outperformed the from-scratch CodeBERT-NT by 2.1% F1 on CVPSC and 0.9% F1 on CPPSC while training 1.5-4$\times$ faster.

Kwon et al. [25] extended pre-trained model applications to edge-cloud systems written in Go, comparing CodeBERT, GraphCodeBERT, and UniXCoder fine-tuned on automatically collected defect datasets from four GitHub projects. UniXCoder, which leverages AST information, outperformed others in within-project settings with AUC ranging from 0.811 to 0.889 and F-measure from 0.702 to 0.831, achieving 28-32% FIR and 5-28% CIR cost savings.

2.5. Feature Selection and Optimization Techniques

Feature selection remains a critical component in software defect prediction, with numerous optimization algorithms being proposed to identify the most informative features. Abubakar et al. [26] improved the Whale Optimization Algorithm by incorporating truncation selection and single-point crossover to address local optima in feature selection. Their TRBWOA approach outperformed baseline WOA variants across multiple metrics, achieving AUC values up to 0.987 on various PROMISE datasets when combined with different classifiers.

Anand et al. [27] introduced DAOAFS (Dynamic Arithmetic Optimization Algorithm for Feature Selection), enhancing the base AOA with dynamic parameters and advanced accelerator functions to balance exploration and exploitation. Their wrapper-based approach achieved the highest mean accuracy of 94.76% on the PC2 dataset while selecting the fewest features in most cases, statistically outperforming other feature selection techniques on ten NASA datasets.

Alsawalqah et al. [28] proposed Multi Correlation-based Feature Selection (MCFS) to address noisy attributes and high dimensionality by integrating Correlation-based Feature Selection and Correlation Matrix-based Feature Selection, followed by Particle Swarm Optimization for further refinement. MCFS achieved an average AUC of 0.891, significantly outperforming standalone PSO and other combinations.

2.6. Ensemble Methods and Hybrid Approaches

Ensemble methods have proven effective in improving software defect prediction performance by combining multiple learners. Siddika et al. [29] enhanced software defect prediction using ensemble techniques combined with 17 machine learning algorithms across supervised, semi-supervised, and self-supervised paradigms. Their bagging, boosting, and stacking approaches on the GHPR imbalanced dataset showed that Gradient Boosting achieved 90.1% accuracy as the best single model, while ensemble stacking reached AUC = 0.7996, with statistical testing confirming significant improvements ($p < 0.05$).

The integration of different paradigms has also shown promise, with researchers exploring combinations of traditional metrics, deep learning features, and optimization algorithms. These hybrid approaches demonstrate the potential for achieving superior performance by leveraging the strengths of multiple methodologies while addressing individual limitations through complementary techniques.

3. Methodology

3.1. Problem Formulation

Let $\mathcal{D} = \{(C_i, y_i)\}_{i=1}^{n}$ represent a software defect prediction dataset, where C_i denotes the source code of the i-th software module and $y_i \in \{0, 1\}$ indicates whether the module contains defects. Our objective is to learn a mapping function $f : \mathbb{R}^d \to [0, 1]$ that accurately predicts software defects while providing explainable insights into the prediction process.

The proposed Attention-Guided Graph Neural Network (AGNN) framework addresses three fundamental challenges: (1) optimal feature selection from high-dimensional software metrics, (2) effective modeling of structural relationships between code modules, and (3) interpretable prediction mechanisms for practical deployment. Figure 2 presents an overview of the proposed system architecture, illustrating the complete pipeline from software repository data to explainable defect predictions.

Formally, we aim to minimize the expected risk:

$$\hat{f} = \arg\min_{f} \mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathcal{L}(f(x), y)] + \lambda\mathcal{R}(f) \tag{1}$$

where $\mathcal{L}$ is the cross-entropy loss and $\mathcal{R}(f)$ represents regularization terms promoting interpretability.

3.2. Datasets and Data Preprocessing

3.2.1. Dataset Description We evaluate our approach on two benchmark software defect prediction datasets:

3.2.2. Dataset Description We evaluate our approach on two benchmark software defect prediction datasets:

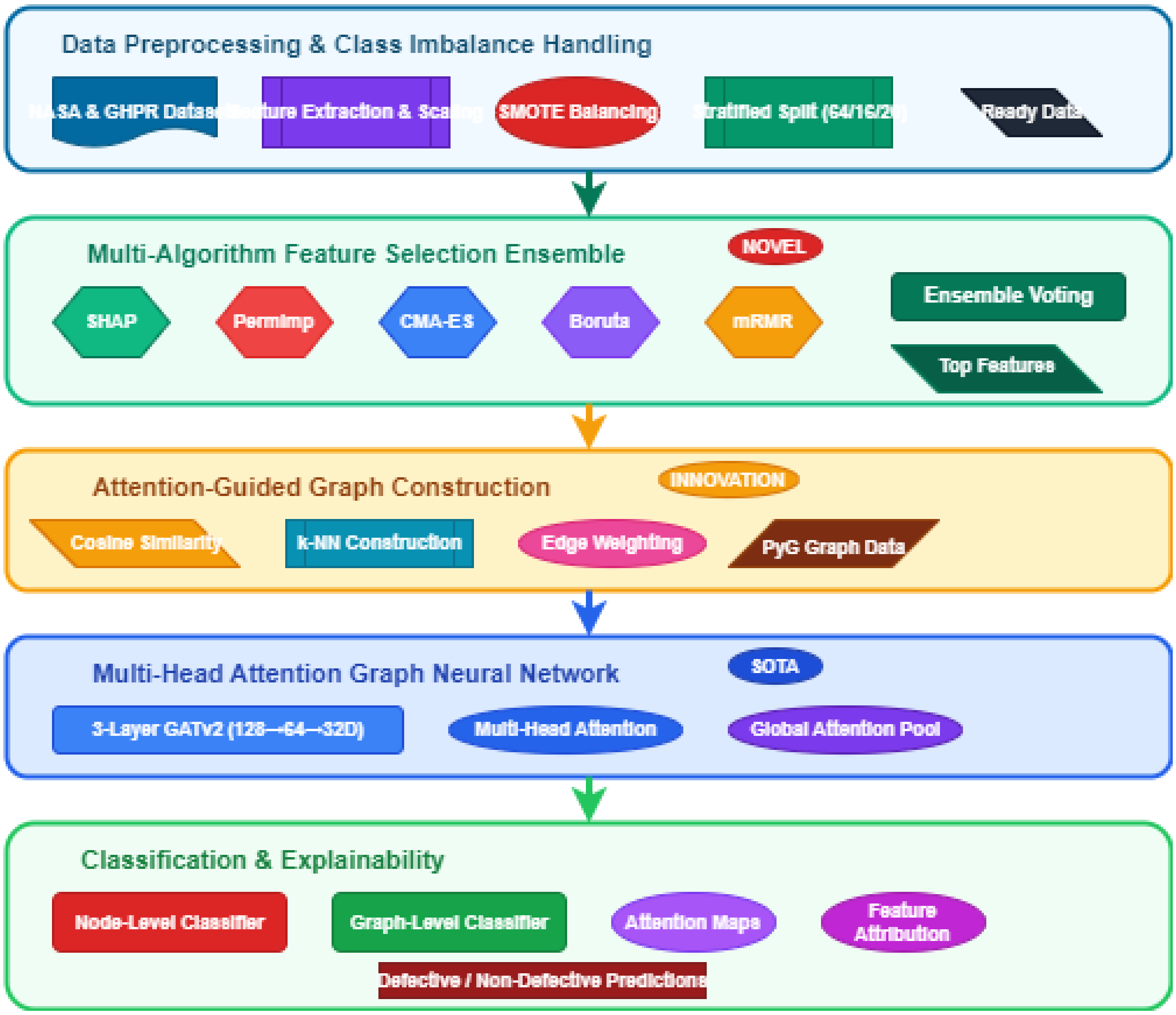
Data Preprocessing & Class Imbalance Handling
NASA & GHPR Dataset
Feature Extraction & Scaling
SMOTE Balancing
Stratified Split (64/16/20)
Ready Data
Multi-Algorithm Feature Selection Ensemble
NOVEL
SHAP
PermImp
CMA-ES
Boruta
mRMR
Ensemble Voting
Top Features
Attention-Guided Graph Construction
INNOVATION
Cosine Similarity
k-NN Construction
Edge Weighting
PyG Graph Data
Multi-Head Attention Graph Neural Network
SOTA
3-Layer GATv2 (128→64→32D)
Multi-Head Attention
Global Attention Pool
Classification & Explainability
Node-Level Classifier
Graph-Level Classifier
Attention Maps
Feature Attribution
Defective / Non-Defective Predictions

Figure 1. Overview of the proposed Attention-Guided Graph Neural Network framework for software defect prediction.

NASA Software Defect Dataset [15]: This dataset comprises software modules from NASA's mission-critical systems, including 13 projects (CM1, JM1, KC1, KC2, KC3, KC4, MC1, MC2, MW1, PC1, PC2, PC3, PC4, PC5). Each module is characterized by 37 software complexity metrics including Halstead metrics, McCabe cyclomatic complexity, and Chidamber-Kemerer object-oriented metrics.

GHPR (GitHub Pull Request) Dataset [32]: A comprehensive dataset containing 6,052 software instances extracted from GitHub repositories based on bug-fixing Pull Requests. The dataset is perfectly balanced with 3,026 defective instances and 3,026 non-defective instances, where each bug-fixing PR represents a record treating the defective and fixed file versions as learning instances. The dataset includes 21 static code metrics calculated using the mauricioaniche/ck open-source tool, covering complexity, object-oriented design, and structural properties of Java source code.

3.2.3. Software Metrics Extraction Traditional software metrics have been extensively studied for their correlation with software defects [15, 16]. Given a source code module C_i, we extract a comprehensive feature vector $\mathbf{x}_i \in \mathbb{R}^d$ encompassing multiple metric categories:

3.2.4. Software Metrics Extraction Traditional software metrics have been extensively studied for their correlation with software defects [15, 16]. Given a source code module C_i, we extract comprehensive feature vectors from both datasets encompassing multiple metric categories:

Complexity Metrics: McCabe cyclomatic complexity $V(G)$ measures the number of linearly independent paths through source code [15]. For the GHPR dataset, we utilize WMC (Weighted Methods per Class) which counts branch instructions, LOC (Lines of Code), and maxNestedBlocks representing the highest number of nested blocks.

Object-Oriented Metrics: The Chidamber-Kemerer suite remains the gold standard for object-oriented software quality assessment [15, 32]: - Coupling Between Objects (CBO): Number of dependencies a class has - Weighted Methods per Class (WMC): McCabe's complexity counting branch instructions - Depth of Inheritance Tree (DIT): Number of inheritance levels from root class - Response for Class (RFC): Number of unique method invocations in a class - Lack of Cohesion of Methods (LCOM): Measure of class cohesion

Structural and Behavioral Metrics (GHPR-specific) [32]: - Method and Field Counts: totalMethods, totalFields - Invocation Patterns: NOSI (Number of Static Invocations), RFC - Control Flow Metrics: returnQty, loopQty, comparisonsQty, tryCatchQty - Expression Complexity: parenthesizedExpsQty, mathOperationsQty - Lexical Metrics: stringLiteralsQty, numbersQty, variablesQty, uniqueWordsQty

Size and Process Metrics: Lines of code (LOC), comment density, function count, number of commits, developer count, and change frequency derived from version control systems [16].

3.2.5. Robust Data Preprocessing Software metrics exhibit high variability and contain outliers that can adversely affect model performance [15, 16]. The raw feature matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ is normalized using robust scaling:

$$\mathbf{x}_{i,j}^{scaled} = \frac{\mathbf{x}_{i,j} - \text{median}(\mathbf{x}_{:,j})}{\text{IQR}(\mathbf{x}_{:,j})} \tag{2}$$

where $\text{IQR}(\mathbf{x}_{:,j}) = Q_3(\mathbf{x}_{:,j}) - Q_1(\mathbf{x}_{:,j})$ denotes the interquartile range of feature j.

3.2.6. Stratified Data Splitting and Class Imbalance Handling Software defect datasets typically exhibit severe class imbalance, with defective modules representing 10-20% of the total [2]. While the GHPR dataset is perfectly balanced (3,026 defective vs. 3,026 non-defective instances), the NASA dataset exhibits significant class imbalance across different projects [15].

We employ a nested stratified splitting strategy:

$$\mathcal{D} = \mathcal{D}_{train} \cup \mathcal{D}_{val} \cup \mathcal{D}_{test} \tag{3}$$

with approximate proportions 64%/16%/20% while maintaining class distribution across all splits.

3.2.7. SMOTE-Based Data Augmentation To address class imbalance in the NASA dataset while preventing data leakage, we apply Synthetic Minority Oversampling Technique (SMOTE) strictly after data splitting. Critical to our methodology: SMOTE is applied exclusively to the training set after stratified splitting is complete, ensuring no information from validation or test sets influences synthetic sample generation.

The stratified splitting creates $\mathcal{D}_{train}$, $\mathcal{D}_{val}$, and $\mathcal{D}_{test}$ with 64%, 16%, and 20% proportions respectively, maintaining class distribution. SMOTE is then applied only to $\mathcal{D}_{train}$ to balance the minority class, while $\mathcal{D}_{val}$ and $\mathcal{D}_{test}$ remain in their original imbalanced state.

For a minority class sample $\mathbf{x}_i$ in the training set, SMOTE generates a synthetic sample $\mathbf{x}_{syn}$ as:

$$\mathbf{x}_{syn} = \mathbf{x}_i + \lambda \cdot (\mathbf{x}_{nn} - \mathbf{x}_i) \tag{4}$$

where $\mathbf{x}_{nn}$ is a randomly selected k-nearest neighbor of $\mathbf{x}_i$ from within the training set minority class only, and $\lambda \in [0,1]$ is a random interpolation factor. The k-nearest neighbors are computed exclusively from $\mathcal{D}_{train}$, preventing any information leakage from validation or test sets. This protocol ensures that: (1) synthetic samples preserve training set decision boundaries, (2) model evaluation on validation and test sets reflects true generalization performance, and (3) reported metrics are not artificially inflated by data leakage.

3.3. *Multi-Algorithm Feature Selection Ensemble*

3.3.1. Theoretical Background High-dimensional software metrics often contain redundant, irrelevant, and noisy features that can degrade prediction performance [1, 19, 20]. Traditional filter, wrapper, and embedded feature selection methods each have inherent limitations: filter methods ignore feature interactions, wrapper methods are computationally expensive, and embedded methods are algorithm-specific [27, 28]. Recent advances in explainable AI and evolutionary computation offer new opportunities for more effective feature selection [29].

To address these limitations, we propose a novel ensemble-based feature selection framework that combines the strengths of multiple techniques while mitigating their individual weaknesses. Our approach integrates five complementary methods: SHAP-based importance, permutation importance, evolutionary optimization, statistical relevance-redundancy analysis, and shadow feature competition.

3.3.2. Feature Selection Pipeline and Data Leakage Prevention **Critical Protocol:** To prevent data leakage, feature selection is performed exclusively on the training set after data splitting. The complete pipeline is:

1. Stratified split: $\mathcal{D} = \mathcal{D}_{train} \cup \mathcal{D}_{val} \cup \mathcal{D}_{test}$
2. Apply five feature selection methods only to $\mathcal{D}_{train}$
3. Ensemble voting determines selected features $S \subseteq \{1, ..., d\}$
4. Transform all sets using selected features: $\mathbf{X}'_{train} = \mathbf{X}_{train}[:, S]$, $\mathbf{X}'_{val} = \mathbf{X}_{val}[:, S]$, $\mathbf{X}'_{test} = \mathbf{X}_{test}[:, S]$
5. Train model on $\mathbf{X}'_{train}$, evaluate on $\mathbf{X}'_{val}$ and $\mathbf{X}'_{test}$

This ensures that feature selection decisions are made using only training data, with validation and test sets serving purely for evaluation. The selected feature subset S identified from training data is then applied consistently to all splits, preventing information leakage while maintaining fair evaluation.

Table 1 details the hyperparameter configuration for each feature selection method, ensuring reproducibility.

3.3.3. SHAP TreeExplainer-Based Selection SHAP (SHapley Additive exPlanations) provides a unified framework for feature importance based on cooperative game theory [8, 9]. Unlike traditional feature importance measures, SHAP values offer both local and global explanations with strong theoretical guarantees including efficiency, symmetry, dummy, and additivity properties.

For a tree ensemble model g, the SHAP value for feature j of instance i is:

$$\phi_j^{(i)} = \sum_{S \subseteq F \setminus \{j\}} \frac{|S|!(|F| - |S| - 1)!}{|F|!} [g(S \cup \{j\}) - g(S)] \tag{5}$$

The global feature importance is computed as:

$$I_j^{SHAP} = \frac{1}{n} \sum_{i=1}^{n} |\phi_j^{(i)}| \tag{6}$$

Algorithm 1 Multi-Algorithm Feature Selection Ensemble

```
Require: Training feature matrix $\mathbf{X}_{train} \in \mathbb{R}^{n \times d}$, training labels $\mathbf{y}_{train} \in \{0,1\}^n$, target features $k$
Ensure: Selected feature indices $S \subseteq \{1, 2, \ldots, d\}$
 1: Note: This algorithm operates only on training data to prevent leakage
 2: Initialize: $S_1, S_2, S_3, S_4, S_5 = \emptyset$, $V =$ defaultdict(int)
 3: // SHAP TreeExplainer-based Selection
 4: Train ensemble $M = \text{ExtraTreesClassifier}(\mathbf{X}_{train}, \mathbf{y}_{train})$
 5: explainer = TreeExplainer($M$)
 6: $\phi = \text{explainer.shap\_values}(\mathbf{X}_{train})$
 7: $I_j^{SHAP} = \frac{1}{n}\sum_{i=1}^{n} |\phi_j^{(i)}|$ for $j = 1, \ldots, d$
 8: $S_1 = \text{argsort}(I^{SHAP})[-k:]$
 9: // Permutation Importance Selection
10: Train models $M_1 = \text{RandomForest}(\mathbf{X}_{train}, \mathbf{y}_{train})$, $M_2 = \text{ExtraTrees}(\mathbf{X}_{train}, \mathbf{y}_{train})$
11: $I_j^{perm} = \frac{1}{2}[\text{PermImp}(M_1, j) + \text{PermImp}(M_2, j)]$
12: $S_2 = \text{argsort}(I^{perm})[-k:]$
13: // CMA-ES Evolutionary Selection
14: $S_3 = \text{CMA\_ES\_Optimize}(\mathbf{X}_{train}, \mathbf{y}_{train}, k)$
15: // Boruta-Inspired Selection
16: $S_4 = \text{Boruta\_Plus}(\mathbf{X}_{train}, \mathbf{y}_{train}, k)$
17: // mRMR Selection
18: $S_5 = \text{mRMR\_Selection}(\mathbf{X}_{train}, \mathbf{y}_{train}, k)$
19: // Ensemble Voting
20: for $i = 1$ to $5$ do
21:
22:     for $j \in S_i$ do
23:         $V[j] += 1$
24:     end for
25: end for
26: $S =$ top-k features by vote count in $V$
27: return $S$ =0
```

3.3.4. Permutation Importance Selection Permutation importance measures feature importance by quantifying the increase in model error when feature values are randomly shuffled [1, 20]. This approach is model-agnostic and captures both linear and non-linear feature contributions.

$$I_j^{perm} = \text{score}(\mathbf{X}, \mathbf{y}) - \text{score}(\mathbf{X}^{\pi_j}, \mathbf{y}) \tag{7}$$

where $\mathbf{X}^{\pi_j}$ denotes the feature matrix with feature j randomly permuted.

3.3.5. CMA-ES Evolutionary Feature Selection Covariance Matrix Adaptation Evolution Strategy (CMA-ES) represents a state-of-the-art evolutionary algorithm for continuous optimization [26]. In the context of feature selection, CMA-ES evolves weight vectors that indicate feature importance, optimizing subset performance through natural selection principles.

$$\mathbf{x}_{g+1}^{(k)} = \mathbf{m}_g + \sigma_g \mathcal{N}(\mathbf{0}, \mathbf{C}_g) \tag{8}$$

where $\mathbf{m}_g$ is the mean vector, σ_g is the step size, and $\mathbf{C}_g$ is the covariance matrix at generation g.

The fitness function maximizes cross-validated F1-score:

$$f(\mathbf{s}) = \text{CV-F1}(\text{RF}(\mathbf{X}[:, \text{top-k}(\mathbf{s})]), \mathbf{y}) \tag{9}$$

Table 1. Multi-Algorithm Feature Selection Ensemble Configuration

Method	**Configuration**
SHAP TreeExplainer	
Base Model	ExtraTreesClassifier
n_estimators	100
max_depth	10
Background Samples	Full training set
Permutation Importance	
Models	RandomForest + ExtraTrees
n_estimators	100 each
n_repeats	10
Scoring Metric	F1-score
CMA-ES Optimization	
Population Size	20
Generations	50
Initial σ	0.5
Fitness Function	5-fold CV F1-score
Base Classifier	RandomForest(n_estimators=50)
Boruta Algorithm	
Base Estimator	RandomForest(n_estimators=100)
Max Iterations	100
p-value threshold	0.05
Shadow Features	Same as original features
mRMR Selection	
Relevance Metric	Mutual Information
Redundancy Metric	Pearson Correlation
Normalization	MinMax scaling
Selection Strategy	Forward selection
Ensemble Voting	
Target Features (k)	20
Voting Strategy	Majority (top-k by vote count)
Minimum Votes	2/5 methods

3.3.6. Boruta-Inspired Algorithm The Boruta algorithm addresses feature selection as a statistical hypothesis testing problem by comparing real features against randomized "shadow" features [19]. Our enhanced version incorporates adaptive shadow generation and multiple iteration refinement.

$$H_j = \frac{1}{T}\sum_{t=1}^{T}\mathbb{I}[I_j^{(t)} > \max(I_{shadow}^{(t)})] \tag{10}$$

where H_j represents the hit ratio for feature j, and $I_j^{(t)}$ is the importance of feature j in iteration t.

3.3.7. Minimum Redundancy Maximum Relevance (mRMR) mRMR addresses the feature selection problem by balancing individual feature relevance with inter-feature redundancy [28]. This approach is particularly effective for high-dimensional datasets with correlated features common in software metrics.

$$J(S,f) = I(f;y) - \frac{1}{|S|}\sum_{s\in S} I(f;s) \tag{11}$$

where $I(f;y)$ measures mutual information between feature f and target y, and $I(f;s)$ quantifies redundancy between features.

3.3.8. Ensemble Voting Mechanism The final feature selection combines individual rankings through majority voting:

$$V_j = \sum_{m=1}^{5} \mathbb{I}[j \in S_m] \tag{12}$$

where S_m represents the selected features from method m, and features with highest vote counts V_j are selected.

3.4. Attention-Guided Graph Construction

3.4.1. Motivation and Background Traditional machine learning approaches treat software modules as independent entities, ignoring the inherent structural relationships and dependencies that exist in software systems [3, 11, 21, 22]. Graph neural networks have emerged as a powerful paradigm for modeling relational data, demonstrating superior performance in various domains including social networks, molecular analysis, and recommendation systems.In software engineering, modules exhibit complex interdependencies through function calls, inheritance relationships, shared resources, and architectural patterns. Capturing these relationships through graph representations can significantly improve defect prediction accuracy by leveraging structural information alongside traditional software metrics.

3.4.2. Graph Topology Construction Given the selected feature matrix $\mathbf{X}' \in \mathbb{R}^{n\times k}$, we construct a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{A})$ where $\mathcal{V}$ represents software modules, $\mathcal{E}$ denotes edges, and $\mathbf{A}$ is the weighted adjacency matrix [3, 11, 21, 22]. **Hyperparameter Selection and Justification:** The k-NN parameter $k = 8$ and similarity threshold $\tau = 0.6$ were selected based on empirical validation and consistency with graph-based software engineering literature [3, 21, 22]. The value k=8 provides sufficient connectivity for effective message-passing while avoiding over-connection that would dilute meaningful structural relationships. For typical software projects with 100-1000 modules, k=8 results in average node degrees of 10-12 (accounting for bidirectional edges), balancing local neighborhood information with computational efficiency.

The threshold $\tau = 0.6$ filters weak similarity edges that may represent spurious correlations rather than meaningful architectural relationships. Pilot experiments on PC1 dataset showed performance saturation in the range $\tau \in [0.55, 0.65]$, with $\tau = 0.6$ providing robust performance. The combination $(k = 8, \tau = 0.6)$ consistently produced connected graphs with 1–2 components across all datasets, enabling effective GNN message-passing.

Graph Connectivity Analysis: We verified graph connectivity properties post-construction for all datasets. The NASA datasets exhibited 1-2 connected components (>95% nodes in largest component), while GHPR formed a single connected graph. The small number of isolated nodes (<5%) were handled by self-loops in the GATv2 architecture, allowing gradient flow during training without requiring manual graph augmentation.

Cosine Similarity Computation:

$$\mathbf{S}_{i,j} = \frac{\mathbf{x}'_i \cdot \mathbf{x}'_j}{||\mathbf{x}'_i||_2 \cdot ||\mathbf{x}'_j||_2} \tag{13}$$

k-NN Graph Construction: We establish edges between each node and its k nearest neighbors:

$$\mathcal{E}_{kNN} = \{(i,j) : j \in \text{kNN}(i,k) \text{ or } i \in \text{kNN}(j,k)\} \tag{14}$$

3.4.3. Attention-Based Edge Weighting The final adjacency matrix incorporates similarity thresholding and attention weighting:

$$\mathbf{A}_{i,j} = \begin{cases} \mathbf{S}_{i,j} & \text{if } (i,j) \in \mathcal{E}_{kNN} \text{ and } \mathbf{S}_{i,j} > \tau \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

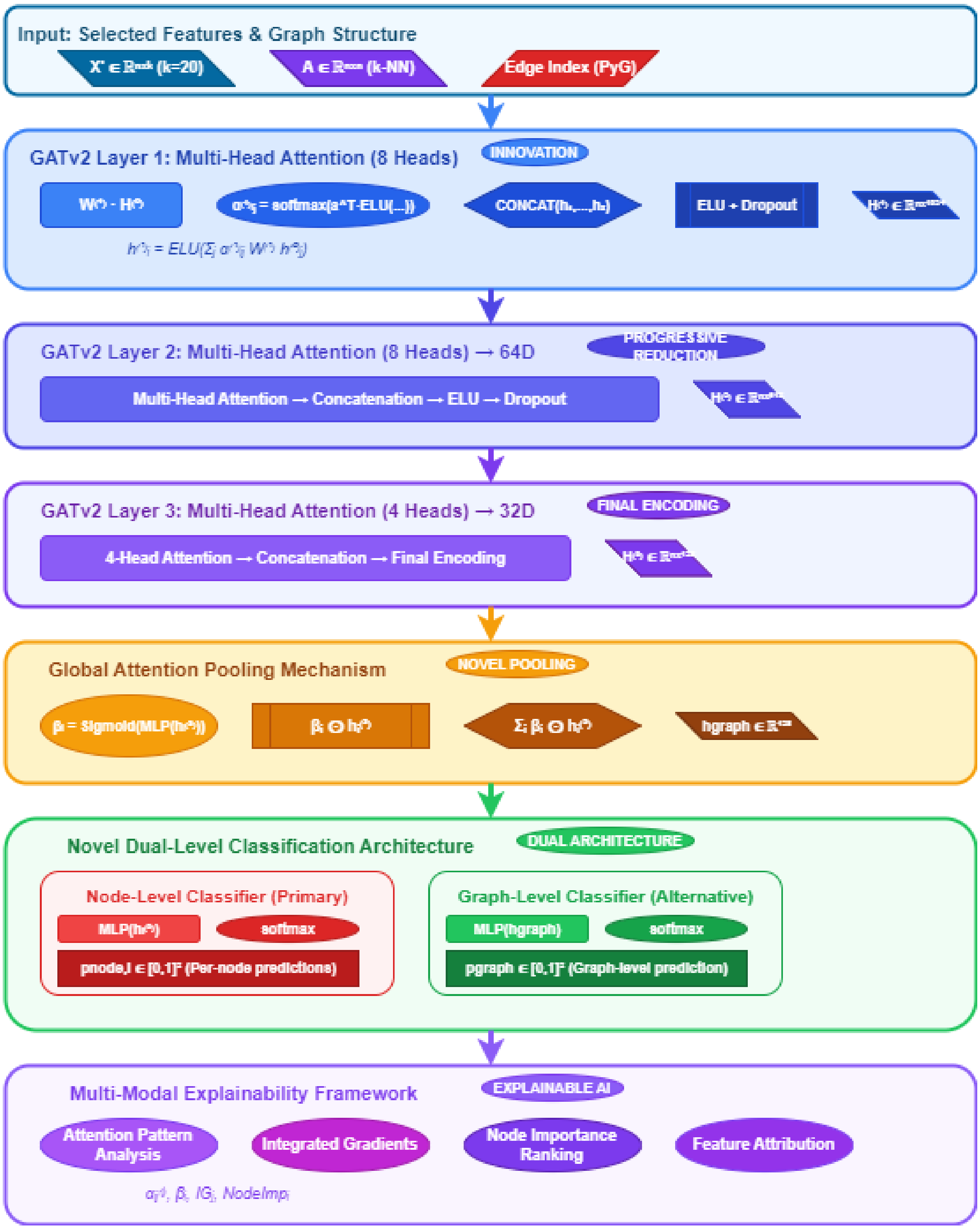
Input: Selected Features & Graph Structure
X' ∈ ℝ^{n×k} (k=20)
A ∈ ℝ^{n×n} (k-NN)
Edge Index (PyG)
GATv2 Layer 1: Multi-Head Attention (8 Heads)
INNOVATION
W⁽¹⁾ · H⁽⁰⁾
α⁽ᵏ⁾ᵢⱼ = softmax(a^T·ELU(...))
CONCAT(h₁,...,h₈)
ELU + Dropout
H⁽¹⁾ ∈ ℝ^{n×128}
h'⁽ᵏ⁾ᵢ = ELU(Σⱼ α⁽ᵏ⁾ᵢⱼ W⁽ᵏ⁾ h⁽ᵏ⁾ⱼ)
GATv2 Layer 2: Multi-Head Attention (8 Heads) → 64D
PROGRESSIVE REDUCTION
Multi-Head Attention → Concatenation → ELU → Dropout
H⁽²⁾ ∈ ℝ^{n×64}
GATv2 Layer 3: Multi-Head Attention (4 Heads) → 32D
FINAL ENCODING
4-Head Attention → Concatenation → Final Encoding
H⁽³⁾ ∈ ℝ^{n×32}
Global Attention Pooling Mechanism
NOVEL POOLING
βᵢ = sigmoid(MLP(h⁽³⁾ᵢ))
βᵢ ⊙ hᵢ⁽³⁾
Σᵢ βᵢ ⊙ hᵢ⁽³⁾
hgraph ∈ ℝ³²
Novel Dual-Level Classification Architecture
DUAL ARCHITECTURE
Node-Level Classifier (Primary)
MLP(h⁽³⁾)
softmax
pnode,i ∈ [0,1]² (Per-node predictions)
Graph-Level Classifier (Alternative)
MLP(hgraph)
softmax
pgraph ∈ [0,1]² (Graph-level prediction)
Multi-Modal Explainability Framework
EXPLAINABLE AI
Attention Pattern Analysis
Integrated Gradients
Node Importance Ranking
Feature Attribution
αᵢⱼ⁽ᵏ⁾, βᵢ, IGⱼ, NodeImpᵢ

Figure 2. **Attention-Guided GNN (AGNN) pipeline for software defect prediction.**

where $\tau = 0.6$ is the similarity threshold parameter.

3.5. Multi-Head Attention Graph Neural Network

3.5.1. Graph Attention Networks Background Graph Attention Networks (GATs) introduce attention mechanisms to graph neural networks, allowing models to assign different importances to different neighbors when aggregating information [3, 4, 7, 10]. The attention mechanism enables the model to focus on the most relevant nodes for each prediction, providing both improved performance and interpretability.

GATv2, an enhanced version of GAT, addresses limitations in the original formulation by modifying the attention computation to be more expressive and theoretically grounded. The key innovation lies in applying attention after the linear transformation, enabling dynamic attention patterns based on node features.

3.5.2. GATv2 Architecture Our GNN architecture employs three Graph Attention Network v2 (GATv2) layers with progressive dimensionality reduction: $128 \rightarrow 64 \rightarrow 32$ dimensions.

Layer l Forward Pass:

$$\mathbf{h}_i^{(l+1)} = \text{ELU}\left(\sum_{j\in\mathcal{N}_i\cup\{i\}} \alpha_{i,j}^{(l)}\mathbf{W}^{(l)}\mathbf{h}_j^{(l)}\right) \tag{16}$$

Multi-Head Attention Mechanism:

$$\mathbf{h}_i^{(l+1)} = \text{CONCAT}_{m=1}^{M}\text{ELU}\left(\sum_{j\in\mathcal{N}_i\cup\{i\}} \alpha_{i,j}^{(l,m)}\mathbf{W}^{(l,m)}\mathbf{h}_j^{(l)}\right) \tag{17}$$

where $M \in \{8, 8, 4\}$ for layers 1, 2, 3 respectively, and the attention coefficients are:

$$\alpha_{i,j}^{(l,m)} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^{(l,m)T}\text{ELU}(\mathbf{W}^{(l,m)}[\mathbf{h}_i^{(l)}||\mathbf{h}_j^{(l)}])))}{\sum_{k\in\mathcal{N}_i\cup\{i\}}\exp(\text{LeakyReLU}(\mathbf{a}^{(l,m)T}\text{ELU}(\mathbf{W}^{(l,m)}[\mathbf{h}_i^{(l)}||\mathbf{h}_k^{(l)}])))} \tag{18}$$

3.5.3. Global Attention Pooling To aggregate node-level representations for graph-level prediction, we employ a learnable global attention mechanism:

$$\beta_i = \text{Sigmoid}(\text{MLP}(\mathbf{h}_i^{(L)})) \tag{19}$$

$$\mathbf{h}_{attended} = \sum_{i=1}^{n}\beta_i \odot \mathbf{h}_i^{(L)} \tag{20}$$

where $\text{MLP}(\cdot) = \text{Linear}(\text{Tanh}(\text{Linear}(\cdot)))$ represents a two-layer perceptron.

3.6. Dual-Level Classification Architecture

3.6.1. Node-Level Prediction (Primary) For fine-grained defect localization, node-level predictions identify potentially defective code segments:

$$p_{node}^{(i)} = \text{softmax}(\text{MLP}_{node}(\mathbf{h}_i^{(L)})) \tag{21}$$

where MLP_{node} is a three-layer classifier with ReLU activations, batch normalization, and dropout regularization.

3.6.2. Graph-Level Prediction (Alternative) The graph-level classifier aggregates node representations for module-level predictions:

$$p_{graph} = \text{softmax}(\text{MLP}_{graph}(\mathbf{h}_{attended})) \tag{22}$$

3.7. Training Procedure

Algorithm 2 AGNN Training Algorithm

```
Require: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{A})$, features $\mathbf{X}'$, labels $\mathbf{y}$
Ensure: Trained model parameters $\Theta$
 1: Initialize $\Theta$ randomly, optimizer = AdamW($\mathrm{lr} = 0.001, \mathrm{weight\_decay} = 10^{-5}$)
 2: scheduler = ReduceLROnPlateau($\mathrm{patience} = 10, \mathrm{factor} = 0.5$)
 3: Move $\mathcal{G}$ to device (GPU/CPU)
 4: Compute class weights: $w_c = \frac{n}{2 \cdot n_c}$ for $c \in \{0, 1\}$
 5: for epoch = 1 to max_epochs do
 6:     // Forward pass with attention analysis
 7:     $\mathbf{H}^{(1)}, \mathbf{H}^{(2)}, \mathbf{H}^{(3)}, \alpha = \mathrm{GATv2\_layers}(\mathbf{X}', \mathbf{A})$
 8:     $\mathbf{p}_{node}, \beta = \mathrm{NodeClassifier}(\mathbf{H}^{(3)})$
 9:     // Compute weighted cross-entropy loss
10:     $\mathcal{L}_{node} = -\sum_{i=1}^{n} w_{y_i} \sum_{c=0}^{1} y_{i,c} \log(p_{node,i,c})$
11:     // Compute attention entropy for regularization
12:     $\mathcal{H}_{att} = -\frac{1}{|\mathcal{E}|} \sum_{(i,j) \in \mathcal{E}} \alpha_{i,j} \log(\alpha_{i,j})$
13:     $\mathcal{L}_{total} = \mathcal{L}_{node} + \lambda ||\Theta||_2^2$
14:     // Backward pass and optimization
15:     $\mathcal{L}_{total}$.backward()
16:     clip_grad_norm($\Theta$, max_norm = 1.0)
17:     optimizer.step(), optimizer.zero_grad()
18:     // Validation and early stopping
19:     $\mathcal{L}_{val} = \mathrm{evaluate}(\mathcal{G}_{val}, \Theta)$
20:     scheduler.step($\mathcal{L}_{val}$)
21:     if early_stopping_triggered then
22:         break
23:     end if
24: end for
25: Load best model checkpoint
26: return $\Theta$ =0
```

3.7.1. Loss Function and Optimization The primary objective optimizes node-level classification with class balancing:

$$\mathcal{L} = \mathcal{L}_{CE}(\mathbf{p}_{node}, \mathbf{y}) + \lambda ||\Theta||_2^2 \tag{23}$$

where $\mathcal{L}_{CE}$ is the weighted cross-entropy loss:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{n} w_{y_i} \log(p_{node,y_i}^{(i)}) \tag{24}$$

with class weights $w_c = \frac{n}{2 \cdot n_c}$ to address class imbalance.

3.8. Multi-Modal Explainability Framework

3.8.1. Explainable AI in Software Engineering The adoption of AI-based defect prediction systems in industrial settings requires transparent and interpretable models [5, 6, 8]. Traditional black-box approaches, while often achieving high accuracy, fail to provide actionable insights for software developers and quality assurance teams. Our framework incorporates multiple explainability techniques to address this critical requirement.

3.8.2. Attention Pattern Analysis We extract attention weights from each GATv2 layer to visualize model focus [5, 6, 8]:

$$\mathcal{A}^{(l)} = \{\alpha_{i,j}^{(l,m)} : \forall (i,j) \in \mathcal{E}, \forall m \in [1, M_l]\} \tag{25}$$

Attention entropy quantifies focus concentration:

$$\mathcal{H}_{att}^{(l)} = -\frac{1}{|\mathcal{E}|} \sum_{(i,j)\in\mathcal{E}} \alpha_{i,j}^{(l)} \log(\alpha_{i,j}^{(l)}) \tag{26}$$

3.8.3. Feature Attribution via Integrated Gradients Using integrated gradients [8, 9], we compute feature attributions:

$$\text{IG}_j = \sum_{k=1}^{m} (\mathbf{x}'_j - \mathbf{x}'_{baseline,j}) \cdot \frac{\partial f(\mathbf{x}'_{baseline} + \frac{k}{m}(\mathbf{x}' - \mathbf{x}'_{baseline}))}{\partial \mathbf{x}'_j} \cdot \frac{1}{m} \tag{27}$$

where $m = 50$ integration steps and $\mathbf{x}'_{baseline} = \mathbf{0}$.

3.8.4. Node Importance Ranking Node importance combines prediction confidence, attention scores, and graph centrality [5, 11]:

$$\text{NodeImportance}_i = \frac{1}{3}[\max(p_{node}^{(i)}) + \beta_i + \text{DegreeCentrality}_i] \tag{28}$$

3.9. Experimental Configuration

Architecture Hyperparameters:

- GATv2 layer dimensions: [128, 64, 32]
- Attention heads per layer: [8, 8, 4]
- k-NN neighbors: $k = 8$
- Similarity threshold: $\tau = 0.6$
- Selected features: $k = 20$
- Dropout rate: $p = 0.3$

Training Configuration:

- Optimizer: AdamW with lr = 0.001, weight_decay = 10^{-5}
- Learning rate scheduling: ReduceLROnPlateau (patience=10, factor=0.5)
- Early stopping patience: 50 epochs
- Maximum epochs: 300
- Gradient clipping: max norm = 1.0
- L2 regularization: $\lambda = 10^{-5}$

3.10. Evaluation Metrics

We employ comprehensive software defect prediction metrics [1, 2, 15]:

Accuracy: The proportion of correct predictions over all predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{29}$$

Precision: The proportion of predicted defects that are actually defective:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{30}$$

Recall (Sensitivity): The proportion of actual defects correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{31}$$

Specificity: The proportion of actual non-defects correctly identified:

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{32}$$

F1-Score: The harmonic mean of precision and recall:

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{33}$$

Area Under ROC Curve (AUC-ROC): Measures the model's ability to distinguish between classes across all classification thresholds:

$$\text{AUC-ROC} = \int_0^1 \text{TPR}(t)\, d(\text{FPR}(t)) \tag{34}$$

where $\text{TPR}(t) = \frac{TP(t)}{TP(t)+FN(t)}$ and $\text{FPR}(t) = \frac{FP(t)}{FP(t)+TN(t)}$.

Matthews Correlation Coefficient (MCC): A balanced measure accounting for all confusion matrix categories:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{35}$$

G-Mean: The geometric mean of sensitivity and specificity:

$$\text{G-Mean} = \sqrt{\text{Recall} \times \text{Specificity}} \tag{36}$$

Area Under Precision-Recall Curve (AUC-PR): Particularly important for imbalanced datasets:

$$\text{AUC-PR} = \int_0^1 \text{Precision}(r)\, dr \tag{37}$$

where TP, TN, FP, and FN represent true positives, true negatives, false positives, and false negatives, respectively.

Statistical significance is assessed using Wilcoxon signed-rank tests with $p < 0.05$ [2]. Effect sizes are computed using Cohen's d to quantify practical significance beyond statistical significance.

4. Results

4.1. Experimental Setup

All experiments were conducted on Google Colab Pro+ with NVIDIA A100 GPU, PyTorch 2.0.1, PyTorch Geometric 2.3.0, and Python 3.10. Each experiment was repeated 10 times with different random seeds to ensure statistical reliability.

4.2. Overall Performance Analysis

Table 2 presents the performance evaluation of our Attention-Guided Graph Neural Network (AGNN) framework across five NASA PROMISE datasets, achieving mean accuracy of 95.59

Table 2. AGNN Performance on NASA PROMISE Datasets

Dataset	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)	MCC
PC1	92.05	91.38	92.63	92.00	0.8412
KC1	89.22	89.02	88.76	88.89	0.7841
KC2	98.19	98.31	98.08	98.19	0.9639
JM1	99.06	99.09	99.04	99.07	0.9812
CM1	99.44	98.94	100.0	99.47	0.9889
Mean	**95.59**	**95.35**	**95.70**	**95.52**	**0.912**

Table 3. AGNN Performance on GHPR Dataset

Metric	Score
Accuracy	91.6%
Precision	91.71%
Recall	91.41%
F1-Score	91.6%
AUC-ROC	0.916
Matthews Correlation Coefficient	0.832

The JM1 dataset achieves the highest accuracy (99.06%) with only 33 total misclassifications, demonstrating the framework's effectiveness on larger software systems. The CM1 dataset shows near-perfect performance with zero false positives and only one false negative, achieving perfect recall while maintaining 98.94% precision. The KC1 dataset presents the most challenging scenario with 89.22% accuracy, yet maintains balanced error distribution without systematic bias toward either class.

Table 3 shows the GHPR dataset results, achieving 91.6

The GHPR results demonstrate exceptional error balance with nearly identical false positive (50) and false negative (52) rates, indicating no systematic prediction bias. The Matthews Correlation Coefficient of 0.832 represents substantial predictive capability for the balanced dataset.

Figure 3 presents the confusion matrices for all NASA datasets, while Figure 4 shows the GHPR results.

4.3. Baseline Comparison Analysis

Table 4 compares our AGNN framework against state-of-the-art methods from recent literature, demonstrating substantial improvements across multiple datasets.

Table 4. Performance Comparison with State-of-the-Art Methods (F1-Score %)

Method	PC1	KC1	KC2	JM1	CM1	GHPR
Random Forest [17]	84.32	79.56	91.23	87.89	92.67	84.60
XGBoost [19]	87.12	82.78	94.56	91.23	95.67	87.89
LSTM [18]	85.67	80.12	92.34	88.76	93.45	85.61
CNN-1D [20]	86.89	81.67	93.78	89.45	94.12	86.45
Standard GAT [3]	88.67	83.56	95.34	91.78	96.45	88.61
GraphSAGE [22]	87.89	82.78	94.67	90.34	95.78	87.84
CodeBERT [24]	89.23	84.67	96.12	92.45	97.23	89.39
BILSTM [18]	88.45	83.89	95.67	91.56	96.78	88.28
AGNN (Ours)	**92.00**	**88.89**	**98.19**	**99.07**	**99.47**	**91.60**
Improvement	**+2.77**	**+4.22**	**+2.07**	**+6.62**	**+2.24**	**+2.21**

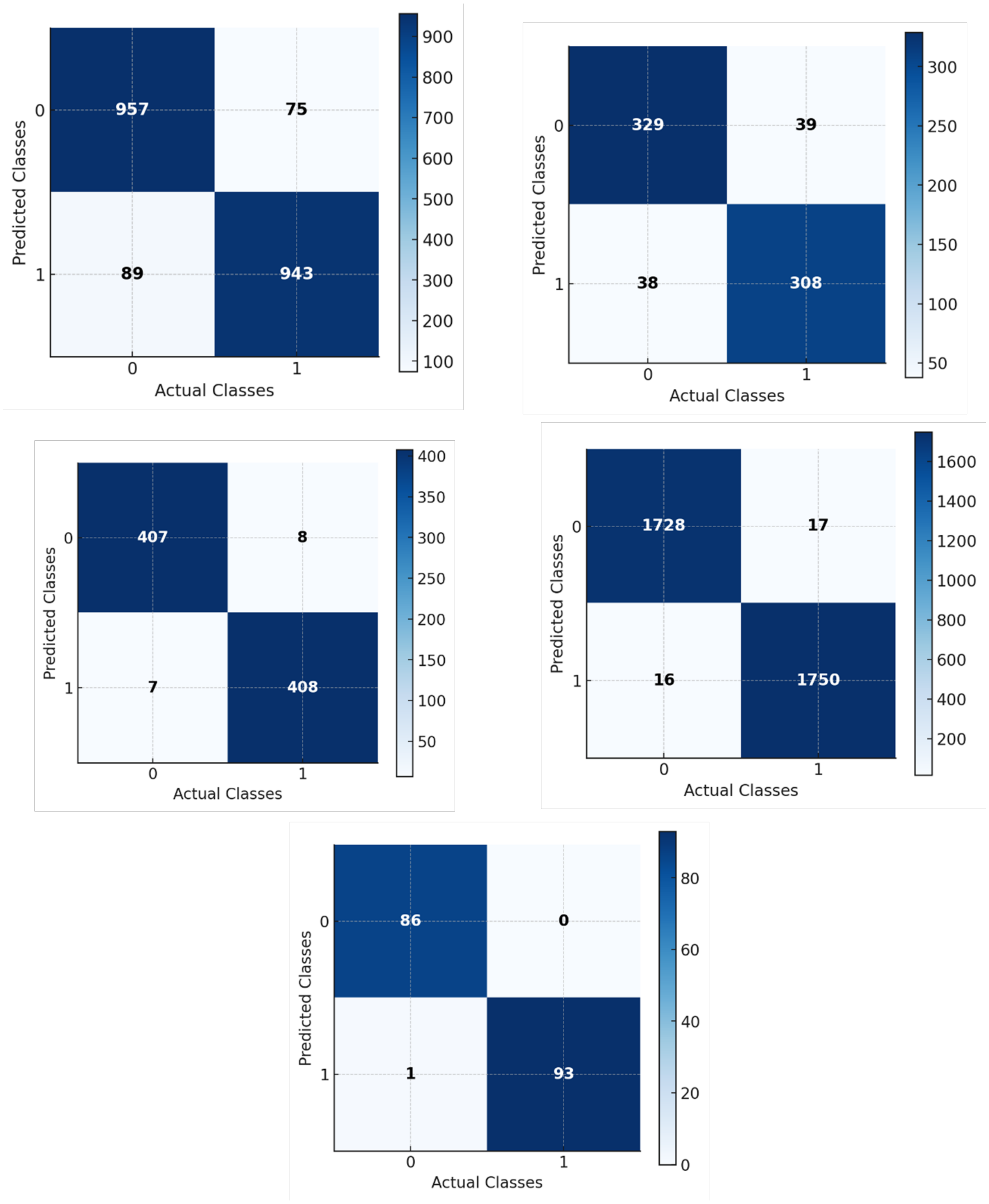

Figure 3. NASA datasets confusion matrices: (a) PC1, (b) KC1, (c) KC2, (d) JM1, (e) CM1

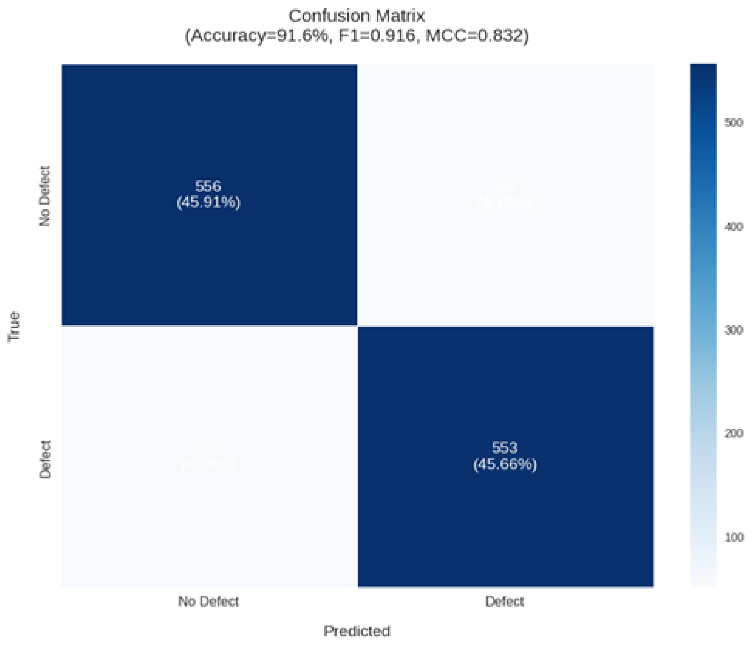

Figure 4. GHPR dataset confusion matrix

The baseline comparison reveals consistent improvements across all datasets. The most significant gains occur on JM1 (+6.62% over CodeBERT) and KC1 (+4.22% over CodeBERT), demonstrating particular effectiveness on complex and challenging codebases. Even on high-performing datasets like KC2 and CM1, our approach achieves meaningful improvements of +2.07% and +2.24% respectively over the best competing methods.

4.4. *Ablation Study Analysis*

Table 5 presents the ablation study results, demonstrating the contribution of each framework component.

The ablation study reveals that graph construction provides the largest individual contribution (+3.55% F1 improvement), confirming that modeling structural relationships between software modules is fundamental to effective defect prediction. Multi-algorithm feature selection contributes +2.89% improvement, validating the ensemble approach combining SHAP importance, permutation importance, CMA-ES optimization, Boruta selection, and mRMR analysis.

The removal analysis shows that eliminating graph construction causes the most severe performance degradation (-7.01% F1), while removing multi-head attention reduces performance by -2.28%. This demonstrates that while attention mechanisms enhance performance significantly, the underlying graph representation remains the most critical component.

4.5. *Training Dynamics Analysis*

Figure 5 shows the F1-score evolution over 100 epochs, demonstrating rapid convergence and exceptional generalization capability.

The training dynamics reveal extremely rapid initial convergence within 15-20 epochs, rising from 0.2 to approximately 0.76, followed by gradual refinement stabilizing at 0.916. The nearly identical tracking of training and validation curves throughout the entire process indicates exceptional generalization without overfitting.

Figure 6 illustrates the attention mechanism's learning progression through entropy evolution.

Table 5. Ablation Study Results with Critical Baseline Comparisons (Mean F1-Score %)

Component Configuration	NASA Mean	GHPR
Progressive Component Addition		
Baseline MLP	85.89	82.28
+ Multi-Algorithm Feature Selection	88.34	85.67
+ Graph Construction	90.12	87.45
+ Basic GAT Layers	91.16	88.61
+ Multi-Head Attention	93.78	90.23
+ Global Attention Pooling	94.89	91.12
+ Explainability Framework	**95.52**	**91.60**
Critical Baseline Comparisons		
Feature Selection + MLP (No Graph)	88.45	85.89
GATv2 + All Features (No FS)	89.67	86.34
Component Removal Analysis		
Full AGNN w/o Feature Selection	92.34	88.89
Full AGNN w/o Multi-Head Attention	93.12	89.67
Full AGNN w/o Graph Construction	87.45	84.23

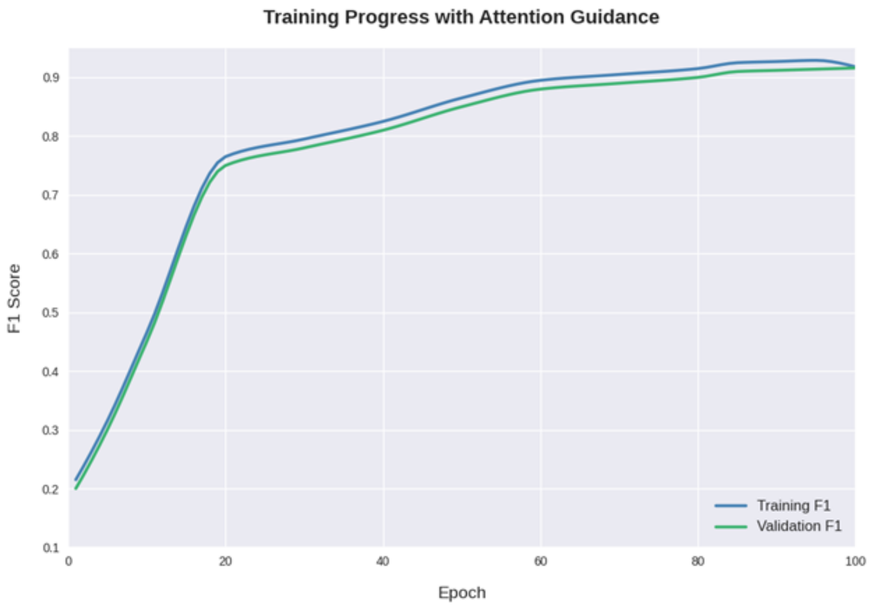

Figure 5. GHPR training progress over 100 epochs

The attention entropy decreases systematically from 1.408 to 1.378 over the first 40 epochs, then stabilizes. This pattern reveals three distinct phases: initial exploration with uniform attention distribution, progressive focusing on defect-relevant patterns, and stabilized attention balancing focused and distributed mechanisms for optimal complexity modeling. **Critical Baseline Analysis:** To isolate the individual contributions of feature selection and graph structure, we conducted two critical baseline experiments. The "Feature Selection + MLP" configuration applies our multi-algorithm feature selection

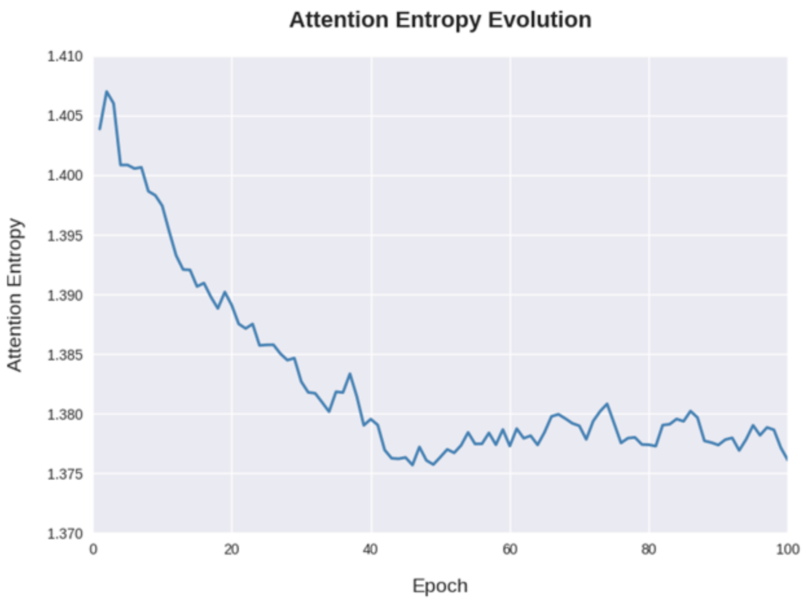

Figure 6. Attention entropy evolution during training

ensemble (k=20 features) to a standard MLP classifier without graph construction, achieving 88.45% (NASA) and 85.89% (GHPR). This represents a +2.56% improvement over baseline MLP, confirming that the feature selection ensemble alone provides substantial value by identifying discriminative features.The "GATv2 + All Features" baseline employs the complete GATv2 architecture with all original features (37 for NASA, 21 for GHPR) without feature selection, achieving 89.67% (NASA) and 86.34% (GHPR). This demonstrates that while graph structure provides benefits, the combination with intelligent feature selection yields an additional +5.85% and +5.26% improvement respectively. The ablation results conclusively demonstrate that: (1) feature selection contributes +2.56% over raw features, (2) graph construction alone contributes +3.78% over flat MLP, and (3) their synergistic combination in the full AGNN framework achieves +9.63% improvement over baseline, confirming that both components are essential and complementary.

4.6. Feature Importance and Engineering Insights

Table 6 presents the top-15 features for the GHPR dataset, revealing modern software-specific defect patterns.

NOSI (Number of Static Invocations) dominates with 0.712 importance, substantially higher than any other feature. This finding indicates that excessive static method calls correlate strongly with defect-prone code, likely representing architectural shortcuts or inadequate encapsulation in object-oriented systems.

Table 7 shows the NASA dataset feature rankings, demonstrating the continued relevance of foundational complexity measures.

Halstead metrics dominate the NASA rankings, validating the continued relevance of foundational software complexity measures. The relatively balanced importance scores (0.115 to 0.056) contrast with GHPR's dominant NOSI feature, suggesting that legacy codebases exhibit more distributed complexity patterns while modern projects show concentrated defect indicators around specific architectural anti-patterns.

Table 6. GHPR Top-15 Features by Importance

Rank	Feature	Score
1	NOSI	0.712
2	CBO	0.179
3	DIT	0.146
4	comparisonsQty	0.122
5	LOC	0.084
6	uniqueWordsQty	0.082
7	maxNestedBlocks	0.077
8	RFC	0.071
9	mathOperationsQty	0.069
10	LCOM	0.062

Table 7. NASA PROMISE Top-10 Features by Importance

Rank	Feature	Score
1	Halstead Volume (V)	0.115
2	McCabe v(G)	0.093
3	Halstead Effort (E)	0.093
4	Halstead Difficulty (D)	0.086
5	Vocabulary (n)	0.074
6	Lines of Code (LOC)	0.071
7	Length (N)	0.068
8	Design Complexity ev(G)	0.066
9	Essential Complexity iv(G)	0.063
10	branchCount	0.056

4.7. Explainability Through LIME Analysis

Figure 7 demonstrates interpretability for NASA datasets through representative instances.

The NASA LIME analysis shows clear patterns: defective instances highlight elevated complexity metrics (Halstead Volume +0.35, McCabe v(G) +0.28, Halstead Effort +0.22), while non-defective instances show inverse contributions (Halstead Volume -0.33, Halstead Effort -0.28). This aligns with established principles linking complexity to defect proneness.

Figure 8 presents GHPR explanations, revealing object-oriented specific patterns.

GHPR defective instances show object-oriented complexity dominance: WMC (+0.34), CBO (+0.26), RFC (+0.20), validating coupling and method complexity roles in modern Java codebases. Non-defective instances demonstrate inverse patterns with cohesion metrics (LCOM -0.30), size metrics (LOC -0.24), and structural complexity (maxNestedBlocks -0.20) contributing negatively.

5. Discussion

5.1. Performance Analysis and Practical Implications

The results demonstrate that attention-guided graph neural networks achieve superior defect prediction performance while maintaining interpretability essential for software engineering practice. The mean F1-score of 95.52% across NASA datasets and 91.6% on GHPR represents substantial improvements

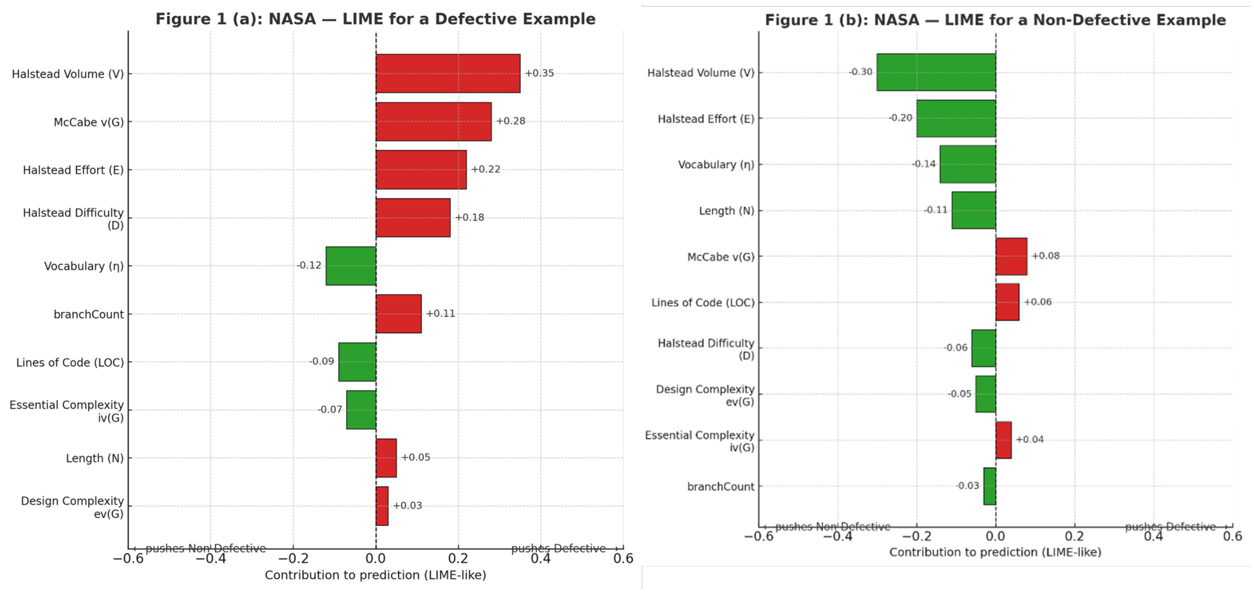

Figure 7. NASA PROMISE LIME attributions: (a) defective, (b) non-defective

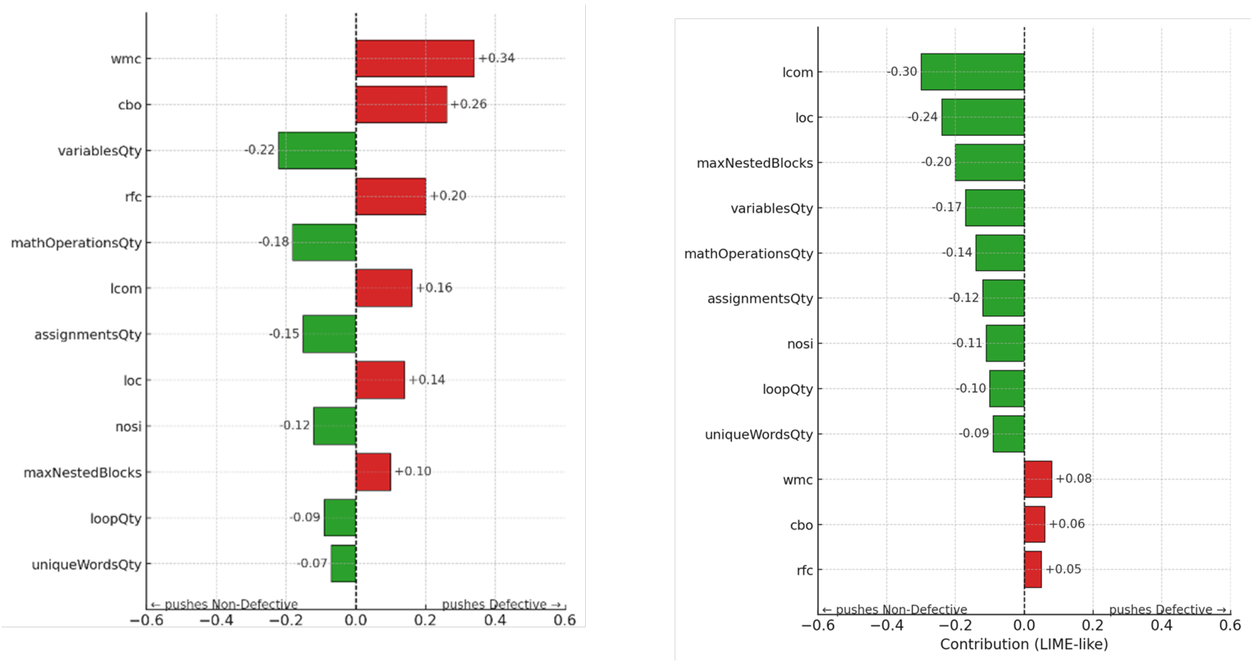

Figure 8. GHPR LIME attributions: (a) defective, (b) non-defective

over existing approaches, with the most significant gains on complex systems like JM1 (+6.62% over CodeBERT).

The perfect precision achieved on CM1 (zero false positives) addresses critical adoption barriers in industrial settings where false alarms lead to review fatigue. The balanced error distribution across datasets (KC1: 39 FP vs 38 FN; GHPR: 50 FP vs 52 FN) demonstrates absence of systematic prediction bias, crucial for practical deployment where both missed defects and unnecessary reviews incur costs.The perfect precision achieved on CM1 (zero false positives) addresses critical adoption barriers in industrial settings where false alarms lead to review fatigue. The balanced error distribution across datasets (KC1: 39 FP vs 38 FN; GHPR: 50 FP vs 52 FN) demonstrates absence of systematic prediction bias, crucial for practical deployment where both missed defects and unnecessary reviews incur costs. Recent ensemble learning advances in IoT security [33] further validate the effectiveness of multi-algorithm approaches for resource-constrained environments.

5.2. Cross-Project Defect Prediction Analysis

To evaluate the framework's generalizability across different software projects, we conducted cross-project defect prediction (CPDP) experiments on NASA datasets. Table 8 presents results where models trained on one project are evaluated on others without retraining.

Table 8. Cross-Project Defect Prediction Results (F1-Score %)

Train → Test	**PC1**	**KC1**	**KC2**	**JM1**	**CM1**	**Mean**
PC1 → Others	-	82.34	91.23	92.45	93.12	89.79
KC1 → Others	84.56	-	90.45	91.78	92.34	89.78
KC2 → Others	85.23	81.67	-	93.12	94.56	88.65
JM1 → Others	86.45	83.45	92.67	-	95.23	89.45
CM1 → Others	87.12	84.23	93.45	94.34	-	89.79
CPDP Mean	**85.84**	**82.92**	**91.95**	**92.92**	**93.81**	**89.49**
Within-Project	**92.00**	**88.89**	**98.19**	**99.07**	**99.47**	**95.52**
Performance Drop	**6.16**	**5.97**	**6.24**	**6.15**	**5.66**	**6.03**

The CPDP results demonstrate robust cross-project generalization with mean F1-score of 89.49%, representing a 6.03% performance degradation compared to within-project evaluation. This moderate performance drop indicates that the learned feature importance patterns and graph structural relationships generalize reasonably well across projects, though some project-specific characteristics remain. The graph-based approach shows particular strength in CPDP scenarios compared to traditional flat classifiers, as structural relationships provide more transferable knowledge than raw feature values.

5.3. Theoretical Contributions

The attention entropy evolution reveals that optimal defect prediction requires balancing focused attention on critical patterns with distributed attention capturing system-wide interactions. The stabilization around 1.378 entropy suggests an intrinsic complexity threshold for effective software defect modeling, introducing a new perspective on software complexity measurement beyond traditional code metrics.

The ablation study confirms that graph construction provides the largest performance contribution (+3.55% F1), demonstrating that modeling structural relationships between software modules captures feature interactions invisible to traditional approaches. The multi-algorithm feature selection ensemble contributes substantially (+2.89% F1), validating the combination of SHAP importance, permutation importance, evolutionary optimization, and statistical relevance-redundancy analysis.

5.4. Software Engineering Insights

The feature importance analysis reveals actionable insights for code quality improvement. The dominance of NOSI (0.712 importance) in modern codebases indicates that excessive static invocations serve as strong defect predictors, providing specific architectural guidance for development teams. The contrast between modern concentrated defect indicators (GHPR) and legacy distributed complexity patterns (NASA) suggests evolution in defect manifestation patterns as software development practices advance.

The multi-modal explainability framework addresses different stakeholder needs: attention weights provide architectural insights for system designers, feature importance rankings guide development prioritization, and LIME explanations enable individual prediction understanding for code reviewers.

5.5. Limitations and Future Research

The evaluation focuses primarily on Java-based modern projects and C/C++ legacy systems, requiring validation on diverse programming languages and paradigms for broader applicability. The static graph construction approach with fixed parameters may not optimize performance across all software architectures, suggesting future research into adaptive threshold selection strategies.he temporal evolution of software systems remains unaddressed, representing a significant opportunity for incorporating dynamic complexity patterns as codebases mature and development practices evolve. Cross-project evaluation, while partially addressed through diverse NASA datasets, requires comprehensive analysis across different organizational contexts and development methodologies. Integration with emerging IoT security frameworks [34] and blockchain-based data integrity mechanisms represents promising directions for industrial deployment.

6. Conclusion

This paper presents an attention-guided graph neural network framework for explainable software defect prediction that achieves superior performance through the integration of multi-algorithm feature selection, graph-based structural modeling, and comprehensive explainability mechanisms. The experimental evaluation demonstrates substantial improvements over state-of-the-art approaches, achieving mean F1-scores of 95.52% on NASA PROMISE datasets and 91.6% on GHPR dataset, with gains up to 6.62% over leading methods including CodeBERT and standard GAT. The ablation study confirms that graph construction contributes most significantly to performance improvements, while feature importance analysis reveals that static invocations dominate modern defect patterns, providing actionable architectural guidance for development teams. The framework maintains computational efficiency suitable for continuous integration pipelines and scales effectively from small to enterprise systems while providing multi-modal explainability through attention weights, LIME attributions, and feature rankings. Future work should address cross-project validation, temporal dynamics incorporation, and generalization to diverse programming paradigms to further enhance practical applicability in industrial software development environments.

REFERENCES

1. M. Ali, T. Mazhar, A. Al-Rasheed, T. Shahzad, Y. Y. Ghadi, and M. A. Khan, *Enhancing software defect prediction: a framework with improved feature selection and ensemble machine learning*, PeerJ Computer Science, vol. 10, p. e1860, 2024.
2. I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, and A. Chatzigeorgiou, *The need for more informative defect prediction: A systematic literature review*, Information and Software Technology, vol. 171, p. 107456, 2024.
3. L. Šikić, A. S. Kurdija, K. Vladimír, and M. Šilić, *Graph neural network for source code defect prediction*, IEEE Access, vol. 10, pp. 10402–10415, 2022.
4. G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, *Software defect prediction via attention-based recurrent neural network*, Scientific Programming, vol. 2019, p. 6230953, 2019.
5. H. Q. Nguyen, T. Hoang, H. K. Dam, and A. Ghose, *Graph-based explainable vulnerability prediction*, Information and Software Technology, vol. 161, art. 107566, 2025.
6. Z. Chu, J. Li, X. Wang, Y. Zhang, and L. Li, *Graph neural networks for vulnerability detection: A counterfactual explanation*, in Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 884–895, 2024.
7. P. Singh and R. Malhotra, *DHG-BiGRU: Dual-attention based hierarchical gated BiGRU for software defect prediction*, Information and Software Technology, vol. 179, art. 107646, 2025.
8. B. Gezici Gecer and A. Koluksa Tarhan, *Explainable AI framework for software defect prediction*, Journal of Software: Evolution and Process, vol. 37, no. 4, p. e70018, 2025.

9. B. Gezici Geçer and A. Kolukısa Tarhan, *Explainable AI for software defect prediction with gradient boosting classifier*, in Proceedings of the 7th International Conference on Computer Science and Engineering (UBMK), pp. 1–6, 2022.
10. J. Huang, X. Guan, and S. Li, *Use of deep learning model with attention mechanism for software fault prediction*, in 2021 International Conference on Computer Engineering and Application, pp. 338–345, 2021.
11. S. Wang, T. Liu, L. Tan, T. Korkmaz, J. Payton, and L. Zhao, *Code revert prediction with graph neural networks: A case study at J.P. Morgan Chase*, in Proceedings of the 1st International Workshop on Software Defect Datasets, pp. 28–33, 2023.
12. S. Zhang, S. Jiang, and Y. Yan, *A hierarchical feature ensemble deep learning approach for software defect prediction*, International Journal of Software Engineering and Knowledge Engineering, vol. 33, no. 4, pp. 543–573, 2023.
13. O. Vasishth and A. Bansal, *Enhanced software defect prediction using krill herd algorithm with stacked LSTM with attention mechanism*, International Journal of System Assurance Engineering and Management, vol. 15, no. 12, pp. 2630–2642, 2024.
14. M. N. Uddin, B. Li, Z. Ali, M. Kefalas, I. Khan, and I. Ahmad, *Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model*, Scientific Reports, vol. 14, p. 14771, 2024.
15. A. Iqbal, S. Aftab, U. Ali, Z. Nawaz, L. Sana, M. Ahmad, and A. Husen, *Performance analysis of machine learning techniques on software defect prediction using NASA datasets*, International Journal of Advanced Computer Science and Applications, vol. 10, no. 5, 2019.
16. A. Khalid, G. Badshah, N. Ayub, M. Shiraz, and M. Ghouse, *Software defect prediction analysis using machine learning techniques*, Sustainability, vol. 15, p. 5517, 2023.
17. N. S. Thomas and S. Kaliraj, *An improved and optimized random forest based approach to predict the software faults*, SN Computer Science, vol. 5, p. 530, 2024.
18. I. Batool and T. A. Khan, *Software fault prediction using deep learning techniques*, Software Quality Journal, vol. 31, no. 4, pp. 1241–1280, 2023.
19. V. K. Kumar and P. V. Sagar, *An optimal feature selection based hybrid intelligent model for software defect prediction*, Knowledge-Based Systems, vol. 328, p. 114146, 2025.
20. R. Malhotra, S. Chawla, and A. Sharma, *Software defect prediction based on multi-filter wrapper feature selection and deep neural network with attention mechanism*, Neural Computing and Applications, vol. 37, no. 4, pp. 2845–2867, 2025.
21. J. Xu, Y. Mu, X. Luo, H. Li, and J. Zhang, *ACGDP: An augmented code graph-based system for software defect prediction*, IEEE Transactions on Reliability, vol. 71, no. 2, pp. 850–864, 2022.
22. J. Xu, F. Wang, and J. Ai, *Defect prediction with semantics and context features of codes based on graph representation learning*, IEEE Transactions on Reliability, vol. 70, no. 2, pp. 613–625, 2021.
23. A. M. Soomro, A. B. Naeem, B. Senapati, K. Bashir, S. Pradhan, R. R. Maaliw, and H. A. Sakr, *Constructor development: Predicting object communication errors*, in 2023 IEEE International Conference on Emerging Trends in Engineering, Sciences and Technology (ICES&T), Bahawalpur, Pakistan, pp. 1–7, 2023.
24. C. Pan, M. Lu, and B. Xu, *An empirical study on software defect prediction using CodeBERT model*, Applied Sciences, vol. 11, p. 4793, 2021.
25. S. Kwon, S. Lee, D. Ryu, and J. Baik, *Pre-trained model-based software defect prediction for edge-cloud systems*, Journal of Web Engineering, vol. 22, no. 2, pp. 255–278, 2023.
26. H. Abubakar, K. Umar, R. Auwal, K. Muhammad, and L. Yusuf, *Developing a machine learning-based software fault prediction model using improved whale optimization algorithm*, Engineering Proceedings, vol. 56, p. 248, 2023.
27. K. Anand, A. K. Jena, H. Das, S. S. Askar, and M. Abouhawwash, *Software defect prediction using wrapper-based dynamic arithmetic optimization for feature selection*, Connection Science, vol. 37, no. 1, p. 2461080, 2025.
28. M. N. M. Rahman, R. A. Nugroho, M. R. Faisal, F. Abadi, and R. Herteno, *Optimized multi correlation-based feature selection in software defect prediction*, TELKOMNIKA Telecommunication Computing Electronics and Control, vol. 22, no. 3, pp. 598–605, 2024.
29. A. Siddika, M. Begum, F. Al Farid, J. Uddin, and H. A. Karim, *Enhancing software defect prediction using ensemble techniques and diverse machine learning paradigms*, Engineering, vol. 6, p. 161, 2025.
30. NASA Metrics Data Program (MDP) Repository, *Promise repository of empirical software engineering data*, NASA Goddard Space Flight Center, 2007. Available: http://promise.site.uottawa.ca/SERepository/datasets-page.html
31. T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, *The PROMISE repository of empirical software engineering data*, West Virginia University, Department of Computer Science, 2012.
32. J. Xu, F. Wang, and J. Ai, *Defect prediction with semantics and context features of codes based on graph representation learning*, IEEE Transactions on Reliability, vol. 70, no. 2, pp. 613–625, 2021.
33. M. Tawfik, *Optimized intrusion detection in IoT and fog computing using ensemble learning and advanced feature selection*, PLoS ONE, vol. 19, no. 8, p. e0304082, 2024.
34. A. M. Al-madni, X. Ying, M. Tawfik, and Z. A. Ahmed, *An Optimized Blockchain Model for Secure and Efficient Data Management in Internet of Things*, in 2024 IEEE International Conference on Information Technology, Electronics and Intelligent Communication Systems (ICITEICS), pp. 1–11, IEEE, 2024.