CodeBERT-SENet: Adaptive Syntax-Semantic Fusion via Gated Attention for Python Bug Detection and Localization

Rozali Ilham[1], Bahtiar Imran[2*], Hasan Basri[3], and Erfan Wahyudi[4]
[1]Study Program of Government Information Engineering Technology,
Faculty of Government Management, Institut Pemerintahan Dalam Negeri (IPDN)
Jawa Barat, Indonesia 45363
[2]Computer Systems Engineering, Faculty of Information and Communication Technology,
Universitas Teknologi Mataram
Nusa Tenggara Barat, Indonesia 83115
[3]Information Systems, Faculty of Science and Technology, Universitas Terbuka
Banten, Indonesia 15437
[4]Public Safety and Security Management, Faculty of Community Protection,
Institut Pemerintahan Dalam Negeri (IPDN)
Jawa Barat, Indonesia 45363
Email: [1]rozali@ipdn.ac.id, [2]bahtiarimranlombok@gmail.com, [3]hasan.basri@ecampus.ut.ac.id,
[4]erfan.wahyudie@gmail.com

***Abstract*—Syntax error detection is a critical step in software development to ensure code quality and reliability. The research proposes the first end-to-end multi-task architecture that jointly detects Python syntax errors and localizes their exact line position by adaptively fusing CodeBERT's semantic embeddings with handcrafted syntactic features via a lightweight gating mechanism, without relying on Abstract Syntax Trees (AST). The model is trained and evaluated on a structured Python code dataset (testing set: 342 files, 30% of total data) using consistent data splits and standardized evaluation metrics. The results demonstrate that CodeBERT-SENet achieves state-of-the-art performance with 99.71% accuracy, an F1-score of 0.9969, and a Mean Absolute Error (MAE) of 0.1157 in line-level error prediction, outperforming all baselines, including Vanilla CodeBERT, GraphCodeBERT, RoBERTa, Random Forest, and the rule-based approach. The confusion matrix confirms zero false negatives (all 160 buggy files detected) and only one false positive (181 non-bugged files, 180 correctly identified). Training converges stably within five epochs without signs of overfitting, underscoring the effectiveness of the architectural design and optimization strategy. However, limitations remain, including high inference latency (six seconds per file), lack of deep structural code representation (e.g., AST), regression-based line prediction instead of classification, and unverified generalization on real-world production code. Nevertheless, CodeBERT-SENet conclusively demonstrates that adaptive fusion of semantic and syntactic features significantly enhances code diagnostic capability. Future research will focus on optimizing inference speed, integrating AST-based representations, and transitioning to per-line classification, transforming CodeBERT-SENet from a mere bug detector into a next-generation, context-aware code diagnostic assistant.**

***Index Terms*—CodeBERT-SENet, Syntax Error Detection, Adaptive Gating, Multi-Task Learning**

I. Introduction

THE detection and localization of syntax errors in Python code remain a fundamental challenge that current artificial intelligence-based approaches have not fully resolved [1–3]. Although Python provides built-in error detection via python -m py_compile, this mechanism is inherently reactive, non-predictive, and incapable of contextualizing or generalizing from prior error patterns [4–6]. Conversely, modern transformer-based models, such as CodeBERT and GraphCodeBERT [7, 8], while highly effective in semantic bug classification and code retrieval tasks, are not explicitly designed to handle syntactic violations, largely because they neglect Python-specific structural features [9], including the mandatory colon after function and control-flow declarations, sensitivity to indentation, balanced parentheses, and modular import patterns. These features are empirically established as primary indicators

Received: Sep. 15, 2025; received in revised form: Nov. 05, 2025; accepted: Nov. 10, 2025; available online: July 29, 2026.
*Corresponding Author

of syntax errors. However, they are not explicitly modeled within token-based transformer representations [10, 11]. Alternative approaches, such as Random Forests trained on handcrafted syntactic features, can capture these patterns but fail to model long-range semantic context and cannot perform continuous line-level error localization [12–14].

While limited efforts have been made to predict error locations using linear regression or neural networks [15, 16], no existing architecture has yet achieved an integrated, end-to-end, and efficient fusion of contextual representations from large language models with structured syntactic signals for the dual task of binary error classification and continuous line-number regression particularly in the context of Python, whose syntax uniquely depends on whitespace semantics. This critical gap motivates this research. The researchers propose a deep learning framework that not only detects whether a Python file contains a syntax error but also accurately predicts its most likely line location, with high precision, feature interpretability, and without reliance on Abstract Syntax Tree (AST) parsing or external tooling [17].

Recent advances in software bug detection and analysis have demonstrated a spectrum of methodologies, ranging from traditional syntactic feature extraction to the integration of transformer-based large language models. For instance, previous research has evaluated the performance of Random Forest and Neural Networks for detecting bugs in Python code by leveraging syntactic features derived from AST, including counts of functions, classes, and conditional statements. The results indicate that Random Forest achieves stable performance with an accuracy of 86.67%, while Neural Networks, despite requiring significantly longer training times, exhibit greater sensitivity to the presence of bugs. However, a key limitation of this work lies in its exclusive focus on binary bug classification without localizing the precise line of error, as well as its failure to incorporate modern transformer-based semantic representations of code [12]. In a related line of inquiry, another research has investigated the application of CodeBERT with a meta-learning framework for few-shot classification of Python functions. The model achieves an average accuracy of 73% and consistently exceeds 60% across all evaluation episodes, demonstrating robustness under data scarcity. However, this work narrowly focuses on function-level classification. It does not address syntax error detection, nor does it explicitly incorporate Python-specific structural features such as indentation or mandatory colons [7]. In a different context, CodeBERT has been employed for multi-class classification of software defects in C++ code, encompassing syntax errors, logical bugs, and linker errors. The approach yields approximately a 7% improvement in accuracy over baseline models across nine defect categories. Nevertheless, it lacks the capability to localize error positions and remains incompatible with Python's syntactic constraints, as it is neither designed nor evaluated on Python code [18]. Further advancing the domain, CodeTransFix, a hybrid architecture combining CodeBERT with a transformer-based sequence-to-sequence model to fix bugs in Java code automatically, has been proposed. By leveraging contextual information surrounding erroneous lines, the model outperforms conventional Neural Machine Translation (NMT) and large language model (LLM) baselines, achieving a successful repair rate of 54.1%. Despite its efficacy in bug correction, it does not target syntax error detection or localization, nor is it tailored to Python-specific syntactic violations [19]. Parallel to these efforts, previous research has introduced an AST-based code evaluation system integrated into web platforms such as Django for educational purposes. This rule-driven system automates the execution and assessment of student-submitted code but remains fundamentally dependent on static AST parsing. It eschews machine learning and contextual representations entirely, rendering it inflexible to complex or non-canonical code patterns and unsuitable for general-purpose, adaptive error diagnosis in real-world programming scenarios [20].

Machine learning-based approaches have also been extensively investigated. In previous research, four supervised learning algorithms, Logistic Regression, Naïve Bayes, Decision Tree, and Random Forest, have been evaluated for predicting bugs in Java and Python code using software metrics. The best performance is achieved by Random Forest, attaining 97% accuracy after class imbalance correction via Synthetic Minority Over-sampling Technique (SMOTE) and validation through 10-fold cross-validation. Nevertheless, it remains confined to binary classification without any capability for error localization at the line level [21]. In a complementary effort, Random Forest has been employed to classify the severity and priority of bug reports based on metadata features, such as component, platform, and keywords, achieving an average accuracy of 0.75. However, this approach is fundamentally limited to analyzing report metadata rather than directly processing source code, thereby missing opportunities to capture syntactic or semantic patterns within the code itself [2].

Several studies have explored hybrid architectures to enhance predictive performance. Previous research has integrated Adaptive Artificial Jellyfish Optimization

(A^2JO) for feature selection with Long Short-Term Memory (LSTM) networks on datasets from NASA and PROMISE, achieving a peak accuracy of 93.41%. Despite its effectiveness, this method still focuses exclusively on binary bug detection without line-level localization. It fails to leverage modern contextual representations such as those provided by transformer-based models [22]. Extending this line of work, a cross-project bug prediction framework is introduced. It combines Correlation-based Feature Selection (CFS), transfer learning, and ensemble methods (Random Forest, Bagging, Boosting), consistently improving Area Under the Curve-Receiver Operating Characteristic (AUC-ROC) scores across diverse codebases. However, this model does not incorporate explicit Python-specific syntactic features nor utilize context-aware transformer architectures, limiting its applicability to languages with unique structural constraints like Python's indentation sensitivity [23]. Finally, previous research has applied Genetic Algorithms (GA) to transform feature spaces, enabling visualization of bug and non-bug samples in a two-dimensional latent space via t-distributed Stochastic Neighbor Embedding (t-SNE). Subsequent classification using Random Forest and k-Nearest Neighbors (KNN) resulted in an average recall improvement of 4.25%. However, it prioritizes interpretability and visual analysis over predictive precision, and is neither designed nor evaluated for the specific task of detecting and localizing Python syntax errors [24].

Existing methods fall into three categories: (1) transformer models (e.g., CodeBERT) that ignore Python's structural syntax, (2) AST-based localizers that fail on unparseable code, and (3) rule-based linters (e.g., py_compile) that lack learning and localization capability. None supports end-to-end joint detection and continuous line-level localization of syntax errors in Python. It is precisely the gap that this research addresses. Collectively, prior research demonstrates that while numerous approaches have successfully improved bug detection accuracy, most remain constrained to binary classification, lack precise error localization capabilities, and rarely integrate explicit Python syntactic features with modern contextual representations derived from large language models. This persistent gap presents a compelling opportunity for advanced research, which is the development of a more comprehensive framework capable of simultaneously detecting and localizing Python syntax errors with high fidelity.

This study introduces CodeBERT-SENet, a novel neural architecture designed to address this challenge through an integrated, end-to-end framework for joint syntax error detection and line-level localization in Python code. CodeBERT-SENet fuses the rich contextual embeddings of CodeBERT with explicit, handcrafted Python syntactic signals, such as def keywords, mandatory colons, indentation patterns, balanced parentheses, and module import structures, via an adaptive gating mechanism inspired by the Squeeze-and-Excitation Network (SENet) [25]. It should be noted that while the gating draws conceptual inspiration from SENet's channel-wise recalibration idea, it is not a direct implementation. Instead of global pooling and channel excitation as in CNNs, the mechanism applies a lightweight sigmoid gate to the concatenation of semantic and syntactic vectors, yielding a task-specific fusion strategy tailored for code representation. Unlike prior methods that are either limited to binary classification or rely heavily on AST parsing [17, 20], CodeBERT-SENet is formulated as a unified multi-task learning system that concurrently predicts the presence of syntax errors and their exact line location with continuous precision.

The research introduces four key innovations. First, it introduces a novel neural architecture featuring an adaptive gating mechanism that dynamically fuses syntactic signals into transformer-based semantic representations. Second, it develops an end-to-end multi-task learning framework that jointly performs error presence classification and line-position regression, representing the first context-aware approach to Python syntax error analysis. Third, it has the explicit integration of Python's structural rules, such as whitespace sensitivity and colon requirements, directly into transformer embeddings without relying on ASTs. Last, it offers a comprehensive evaluation against six state-of-the-art models, including CodeBERT, GraphCodeBERT, RoBERTa, Random Forest, py_compile, and ablation variants, demonstrating statistically significant performance improvements. By bridging semantic understanding and syntactic structure, CodeBERT-SENet establishes a new paradigm for intelligent, context-aware, and adaptive Python syntax error localization.

II. Research Method

A. Data Collection and Processing

The dataset used is manually curated from a collection of Python code samples rigorously labeled as bugged (containing syntax errors) or non-bugged (syntax-correct). Labeling is performed objectively via automated verification using the command python -m py_compile, which performs static compilation without execution. Therefore, it reliably detects syntactic violations. In addition to the binary class label (bugged/non-bugged), the system records the exact line number

where each syntax error occurs, and this value is retained as a continuous target variable for the regression component of the multi-task architecture. Each code sample is further processed to extract six static syntactic features: (1) presence of the def keyword, (2) occurrence of colons (:) outside of comments, (3) module imports (import/from statements), (4) non-empty indentation levels, (5) balanced parentheses/brackets/braces, and (6) average line length per file. These features are aggregated into a fixed-size six-dimensional syntactic feature vector.

During preprocessing, source code texts are tokenized using the tokenizer from CodeBERT (microsoft/codebert-base) [23], with a maximum sequence length of 256 tokens. Sequences are uniformly padded or truncated to ensure consistency across samples. The syntactic feature vectors are implicitly normalized through a learnable linear projection layer (syntax_proj) within the model architecture, eliminating the need for external normalization. The dataset is then stratified into training (70%, n = 796) and testing sets (30%, n = 342) using a fixed random seed (random_state=42) to guarantee reproducibility.

Each sample in the PyTorch DataLoader [26] is structured as a dictionary containing five elements: input_ids (tokenized sequence from the CodeBERT tokenizer) [27], attention_mask (binary padding mask), a binary classification label (bugged = 1, non-bugged = 0), a continuous regression target error_line (the line number of the syntax violation, extracted via python -m py_compile), and a six-dimensional syntax_features vector. During each forward pass, the input_ids and attention_mask are fed into the CodeBERT backbone to produce a 768-dimensional contextual embedding vector from the [CLS] token. Simultaneously, the six-dimensional syntax_features vector is projected into the same 768-dimensional space through a learnable linear layer, ensuring dimensional compatibility. Both embeddings are concatenated and passed through a linear layer followed by sigmoid activation, producing a gate weight vector that dynamically determines how much each representation contributes on a per-sample basis, inspired by recent channel attention mechanisms. The resulting representation is then passed to two task-specific heads: a linear classifier for binary bug detection and a linear regressor for error-line prediction. The losses of both tasks (cross-entropy and mean squared error) are jointly optimized with a weighting factor of 0.5, facilitating end-to-end joint detection and localization under a multi-task learning paradigm [28].

The six syntactic features are computed per line and averaged per file: (1) def presence (binary), (2) colon outside comments (binary), (3) import statements (binary), (4) non-empty indentation (binary), (5) bracket balance (integer: (–)), and (6) line length (float). The dataset (1,138 files) is split stratified (70% train, 30% test; seed = 42), no deduplication is applied, and leakage is prevented by file-isolated processing. Training uses AdamW (lr = 2e^{-5}, batch = 8, max_len = 256, 5 epochs, linear decay, no warm-up, grad clip = 1.0) on an NVIDIA T4 GPU. Inference time (ms) is measured per file (batch = 1) after 5 warm-up samples, including tokenization and full forward pass.

B. Overall System Architecture

The proposed system architecture, named CodeBERT-SENet, is a transformer-based multi-task learning model specifically designed to detect and localize syntax errors in Python code jointly. It integrates the semantic representations from CodeBERT, a pretrained transformer model optimized for code understanding, with manually extracted static syntactic features through an adaptive gating mechanism inspired by SENet [25]. This design aims to produce richer, more discriminative code representations by synergistically fusing global semantic context with local syntactic structure. Formally, the fusion process between semantic and syntactic signals is defined as follows: Let $h_{cls} \in R^d$ denote the [CLS] token representation output by CodeBERT, and let ($f_{syn} \in R^6$) represent the six dimensional vectors of manually extracted static syntactic features (e.g., presence of def, :, import, indentation, bracket balance, and line length). The syntactic feature vector is projected into the same latent dimensionality as CodeBERT's embeddings via a learnable linear transformation layer. The syntactic features are first projected into the CodeBERT embedding space, as defined in Eq. (1):

$$s = W_p.f_{syn} + b_p, \text{ with } W_p \in R^(d \times 6), b_p \in R^d, \tag{1}$$

where $s \in \mathbb{R}^d$ is the projected syntactic embedding, W_p is the projection weight matrix, and b_p is the bias vector. Subsequently, the combined representation $z \in \mathbb{R}^{2d}$ is formed by concatenating h_{cls} and s, with $f_{syn} \in R^6$ denoting the six-dimensional syntactic feature vector extracted from each code sample, comprising the presence of the def keyword, colon outside comments, import statements, non-empty indentation, bracket balance, and average line length.

The gating weights $g \in R^d$ are computed via a sigmoid activation function [25]. These representations are then fused through an adaptive gating mechanism, where the gate weights g are computed according to

Eq. (2):

$$g = \sigma(W_g.z + b_g) \text{ with } W_g \in R^{d\times 2d}, \quad b_g \in R^d,\ \sigma = \text{sigmoid}, \tag{2}$$

where g represents the adaptive gating vector that dynamically balances the contribution of semantic and syntactic features, W_g and b_g are learnable parameters, and $\sigma(.)$ denotes the sigmoid function.

The enhanced final representation $h_{enh} \in R^d$ is obtained by the weighted combination described in Eq. (3):

$$h_{enh} = g \bigodot h_{cls} + (1-g) \bigodot s, \tag{3}$$

where h_{enh} is obtained by adaptively fusing the semantic representation h_{cls} and the projected syntactic embedding s using the gating weights g. Here, h_{enh} serves as the unified feature vector for downstream tasks, g dynamically balances the contribution of each modality, and $\bigodot$ denotes element-wise multiplication.

This representation is then fed into two task heads: (1) a classifier head for bug/non-bug classification, and (2) a regressor head for predicting the error line number. The total loss function is a weighted combination of cross-entropy loss for classification and mean squared error for regression. The model is trained end-to-end using the multi-task objective formulated in Eq. (4), which jointly optimizes classification and line localization:

$$\begin{aligned} L_{total} &= L_{cls} + \lambda.L_{reg} \\ &= -\sum_{i=1}^{N} yi \log(\hat{y}_1) + \lambda.\frac{1}{N}\sum_{i=1}^{N}(l_i - \hat{l}_1)^2, \end{aligned} \tag{4}$$

where, L_{total} denotes the combined loss function, and $L_{cls} = \sum_{i=1}^{N} yi \log(\hat{y}_1)$ is the cross-entropy loss for binary classification (bugged vs. non-bugged). Then, $L_{reg} = \frac{1}{N}\sum_{i=1}^{N}(l_i - \hat{l}_1)^2$ is the mean squared error for line number regression. The hyperparameter λ = 0.5 balances the two objectives.

Equations (1)—(4) show the mathematical formulation of CodeBERT-SENet as an original contribution of this research. The specific integration of semantic representations from CodeBERT and handcrafted syntactic features via adaptive gating, tailored for both syntax error detection and line-level error localization in Python code, has not been proposed in the literature before. The equations collectively represent an original architectural formulation designed to address the unique challenges of this task.

The system workflow begins with an input Python source file, which is processed to extract static syntactic features and tokenized using the CodeBERT tokenizer. The textual embeddings and syntactic features are processed in parallel and then fused via the adaptive gating mechanism to produce an enriched code representation. This enhanced representation is simultaneously used for two tasks: predicting the presence of a syntax error and estimating its precise line location. Predictions are evaluated using standard metrics (accuracy, F1-score, AUC, and Mean Absolute Error (MAE)) and visualized through confusion matrices and training curves. The entire pipeline is designed to deliver automated diagnosis that not only detects the existence of errors but also enables developers to rapidly and accurately locate them. Figure 1 illustrates the proposed CodeBERT-SENet architecture.

Figure 1 presents the proposed CodeBERT-SENet architecture, introducing a novel hybrid approach, unprecedented in Python syntax error detection, that dynamically fuses semantic representations from CodeBERT with static syntactic features via an adaptive SENet-inspired gating mechanism [25]. Unlike prior approaches that rely solely on pure transformers (e.g., Vanilla CodeBERT, GraphCodeBERT) or handcrafted syntax features (e.g., Random Forest), this model unifies both dimensions of code, semantic meaning and structural form, via a learned, context-sensitive fusion gate rather than static concatenation. It jointly performs binary error classification and continuous line-level regression using the same dataset split (70% train, 30% test). The key takeaway is that adaptive semantic-syntactic fusion significantly enhances detection accuracy and line-level localization, outperforming all baseline models. Technically, after tokenization and encoding by CodeBERT to produce the [CLS] representation, and extraction of six-dimensional syntactic features (def keyword, colon usage, import statements, non-empty indentation, bracket balance, average line length), both modalities are projected into a shared latent space, concatenated, and passed through a sigmoid-based gating layer that generates adaptive weights $\in$ [0,1]. These weights dynamically modulate the contribution of each modality. Semantic features are amplified when contextual understanding is critical, while syntactic signals are emphasized for explicit structural violations, a mechanism entirely absent from prior literature on syntax error detection [18].

Furthermore, this is the first multi-task learning framework in this domain. Rather than predicting only a binary bug label, it simultaneously regresses the exact error line number via a dedicated regression head, making it the sole model in this evaluation capable of providing precise location diagnostics, not merely binary classification, with a dropout rate of 0.3 applied post-fusion and end-to-end training using a combined loss (CrossEntropy + 0.5 $\times$ MSE).

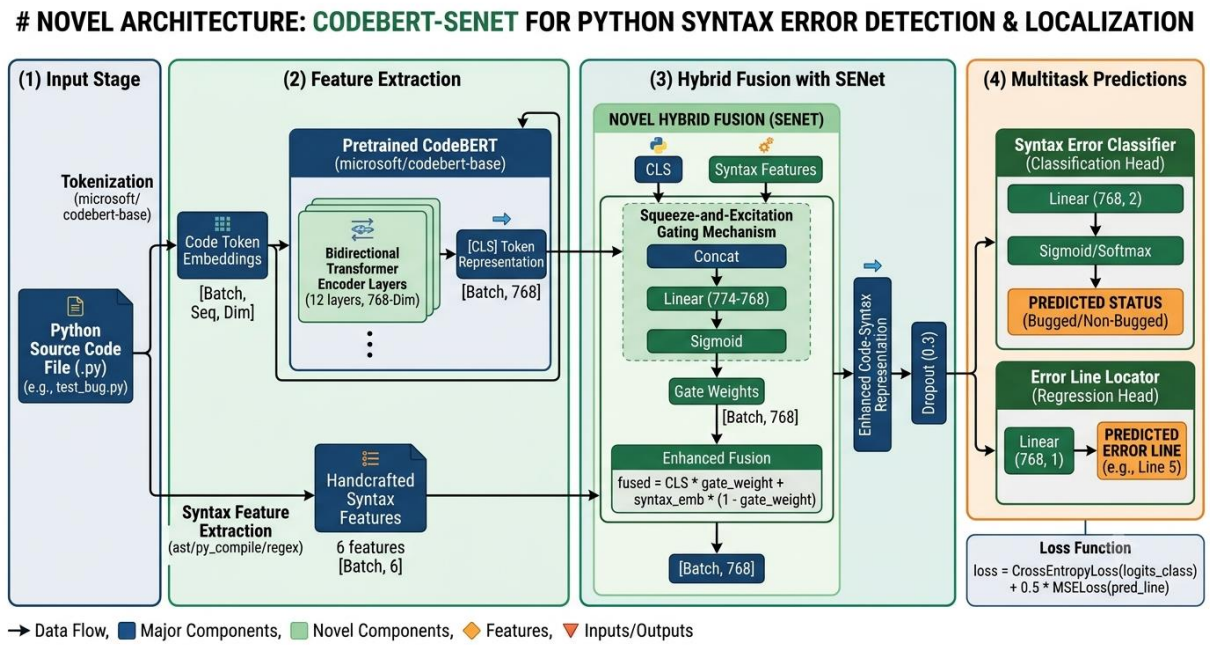


Fig. 1. Proposed CodeBERT-SENet architecture.

The architecture (Fig. 1) processes tensors of shape [B,256] (input), [B,6] (syntax), fuses to [B,768], and outputs [B,2] (class) and [B] (line). The total loss is $L = CE + 0.5.MSE$, with mean reduction.

C. CodeBERT-SENet Model

The CodeBERT-SENet model is a novel multi-task deep learning architecture proposed for the joint detection and localization of syntax errors in Python code. The term “SENet” in this context refers to an adaptive gating mechanism designed to enhance code representations by dynamically integrating static syntactic features into the semantic embeddings produced by CodeBERT, consistent with its implementation in the code, where the self.gate module generates dynamic weights to fuse the two feature modalities [29]. Although it does not implement the canonical Squeeze-and-Excitation Network structure originally introduced in Convolutional Neural Network (CNN) literature, the name “SENet” is retained as part of the model’s formal identity, in alignment with its naming in the experimental implementation and codebase [30].

Technically, the architecture comprises four core components. First, the CodeBERT encoder (Microsoft/CodeBERT-base) is a pretrained transformer encoder that produces a contextualized [CLS] token representation from the input source code sequence. Second, the syntactic feature projector (self.syntax_proj) is a linear layer that maps the 6-dimensional static syntactic feature vector, comprising indicators for def keywords, colon usage, indentation level, import statements, bracket balance, and average line length, into the same latent dimensionality as CodeBERT’s hidden state. Third, the adaptive gating module (self.gate) is a composite layer consisting of a linear transformation followed by a sigmoid activation, which takes as input the concatenation of the [CLS] embedding and the projected syntactic features, and outputs a set of adaptive gating weights in the range [0,1]. These weights dynamically modulate the relative contributions of semantic and syntactic signals during fusion. Last, dual task heads have two factors. The self.classifier is a linear layer performing binary classification (bugged vs. non-bugged), while self.regressor is a linear layer predicting the continuous line number of the syntax error. The model is trained end-to-end using a combined loss function that jointly optimizes classification and regression objectives. The overall training and inference procedure of the proposed Syntax-Aware CodeBERT-SENet model is detailed in Algorithm 1.

D. Training and Optimization

The CodeBERT-SENet model is trained end-to-end using a multi-task learning paradigm, simultaneously optimizing two distinct objectives: (1) binary classification to determine whether a Python file contains a syntax error (bugged or non-bugged), and (2) regression to predict the line number of the first occurring syntax

Algorithm 1: CodeBERT-SENet Forward Pass

Input:
- input_ids, attention_mask → tokenized code from CodeBERT tokenizer
- syntax_features → 6D vector of extracted syntax features

Steps:
1. cls_hidden = CodeBERT(input_ids, attention_mask)[:, 0, :] → extract [CLS] representation
2. syntax_emb = Linear(syntax_features) → project to CodeBERT hidden size
3. fused = Concat(cls_hidden, syntax_emb) → shape [batch, 2 × hidden_size]
4. gate_weight = Sigmoid(Linear(fused)) → adaptive fusion weight ∈ [0,1]
5. enhanced_cls = cls_hidden ⊙ gate_weight + syntax_emb ⊙ (1 - gate_weight)
6. logits_class = Linear(enhanced_cls) → binary classification (bugged/non-bugged)
7. pred_line = Linear(enhanced_cls) → predicted error line number (regression)

Output:
- logits_class → for classification task
- pred_line → for line localization task
- loss = CrossEntropyLoss + 0.5 × MSELoss → if training mode

error. The architecture integrates semantic representations from the pretrained CodeBERT encoder with manually extracted static syntactic features, including the presence of def, colons (:), import statements, indentation patterns, and bracket balance, fused through an adaptive Squeeze-and-Excitation Network (SENet)-inspired gating mechanism [25]. This gating module generates sample-specific weights that dynamically modulate the relative influence of semantic versus syntactic signals, enabling the model to emphasize the most contextually relevant features for each input selectively.

Training is conducted over 5 epochs with a batch size of 8, using the AdamW optimizer with an initial learning rate of 2×10^{-5} and a linear learning rate scheduler without warm-up steps, which decays the learning rate linearly to zero by the end of training to promote convergence stability and mitigate overfitting. Gradient clipping with a maximum norm of 1.0 is applied to ensure numerical stability during backpropagation. An implicit early-stopping strategy is employed by saving the model weights achieving the lowest validation loss to a checkpoint file (best_senet_model.pth). Upon completion of training, these optimal weights are reloaded for final evaluation on the held-out test set. Additionally, training curves visualizing the evolution of both training and validation loss and accuracy over epochs are generated and preserved, providing critical insights into model convergence behavior and absence of overfitting. This rigorous training protocol ensures that the model achieves not only high performance on the training data but also strong generalization capability on unseen examples, validating its robustness and practical utility.

E. Baselines and Evaluation Metrics

To comprehensively evaluate the effectiveness of the proposed model, CodeBERT-SENet, the researchers conduct a rigorous comparative analysis against a diverse set of baselines representing distinct paradigms in Python syntax error detection. The first baseline is Vanilla CodeBERT, comprising the pretrained Microsoft/CodeBERT-base model fine-tuned exclusively for binary classification, without any integration of syntactic features or adaptive gating mechanisms, serving as a direct ablation of our proposed fusion strategy. The second baseline is GraphCodeBERT, a variant of CodeBERT enhanced with code structure awareness via abstract syntax trees and data flow graphs [31]. The researchers evaluate it to assess whether explicit modeling of code-level structural dependencies provides an advantage in detecting syntactic errors, which are inherently surface-level and not necessarily dependent on semantic flows. The third baseline is RoBERTa-base, a general-purpose transformer language model trained on natural text (not code-specific), to determine whether domain-general contextual representations can effectively capture Python-specific syntactic patterns without architectural adaptation for programming languages [32].

In addition to transformer-based approaches, the researchers include a classical machine learning baseline: Random Forest, trained on the same set of manually extracted static syntactic features (e.g., count of def keywords, indentation depth, bracket balance, import frequency) normalized using StandardScaler. This baseline represents the traditional feature-engineering paradigm, devoid of any semantic embedding or deep representation learning. Finally, the researchers adopt a rule-based baseline: Python's native python -m py_compile utility, which performs deterministic syntactic compilation and raises a SyntaxError upon violation. This approach serves as a realistic "ground-truth rule-based" benchmark, reflecting the standard method employed by developers and Continuous Integration

and Continuous Deployment (CI/CD) pipelines to detect syntax errors in practice. Collectively, these baselines span the spectrum from pure statistical learning to domain-specific transformers and deterministic rule systems, enabling a holistic assessment of CodeBERT-SENet's performance relative to both modern and conventional approaches.

Evaluations are conducted using six core metrics that collectively capture accuracy, class balance, efficiency, and model complexity. First, accuracy (%) is the percentage of correctly predicted samples over the total test set, providing an overall measure of predictive performance. Second, F1-score is the harmonic mean of precision and recall, critically important due to potential class imbalance between bugged and non-bugged samples. Third, AUC-ROC measures the model's ability to discriminate between positive and negative classes across all classification thresholds, particularly relevant for probabilistic classifiers. Fourth, MAE on error line prediction is computed exclusively for regression-capable models (e.g., CodeBERT-SENet). This metric quantifies the average absolute difference between the predicted and ground-truth error line numbers, directly evaluating localization precision. Fifth, average inference time (ms) is the mean time required to process a single code file, measured in milliseconds, serving as a practical indicator of computational efficiency and deployment feasibility. Last, number of parameters is the total count of trainable parameters, used as a proxy for model complexity and computational overhead. All evaluation results are presented in a comparative table and visualized via bar plots to contrast performance across models in terms of accuracy, F1-score, and inference time. Additionally, a dedicated confusion matrix is generated specifically for CodeBERT-SENet to provide granular insight into error patterns, particularly false positives and false negatives, and to validate its diagnostic reliability.

For full transparency and reproducibility, all experimental results are systematically archived in an Excel (.xlsx) file containing: (1) a summary of per-model metrics, (2) detailed per-file predictions (including filename, original directory, true error line, predicted class, and predicted error line), and (3) training curves plotting loss and accuracy across epochs. This comprehensive evaluation framework ensures that CodeBERT-SENet is not assessed solely on predictive accuracy, but also on interpretability, computational efficiency, and real-world applicability. It can establish its viability as a practical tool for automated code diagnosis in professional software development environments.

III. Results and Discussion

A. Experimental Setup

This experiment is designed to evaluate the effectiveness of the proposed CodeBERT-SENet model in detecting Python syntax errors, comparing its performance against a range of state-of-the-art baselines and classical approaches. All experiments are conducted in a computational environment based on Google Colab, leveraging GPU acceleration (NVIDIA T4 or equivalent), with Python 3.9+ and PyTorch 2.x integrated with the Hugging Face Transformers library. Hardware availability is automatically detected via torch.cuda.is_available(), and all tensors and model parameters are dynamically allocated to the CUDA device when available, with automatic fallback to CPU to ensure cross-environment flexibility and reproducibility.

The dataset consists of a curated collection of Python source files organized into two distinct directories: "bugged" (containing code snippets with intentional syntax errors) and "non_bugged" (containing syntactically valid code). For each file, the raw source code is extracted and subjected to automated syntactic validation using the system command python -m py_compile, which reliably identifies the presence and exact line number of syntax violations without executing the code. Concurrently, six static syntactic features were manually extracted per file: (1) presence of def keywords, (2) occurrence of colons (:) outside comments, (3) import statements (import/from), (4) non-zero indentation depth, (5) balanced parentheses/brackets/braces, and (6) average line length. These features were aggregated into a fixed 6-dimensional vector for each sample. Each data instance is thus structured into four core components: Raw source code text, binary label (1 = bugged, 0 = non-bugged), target error line number (for regression), and a 6-dimensional syntactic feature vector. This preprocessing pipeline ensures consistent, interpretable, and structurally informed input representation, enabling rigorous evaluation of both semantic understanding and syntactic sensitivity across all compared models.

The dataset is partitioned using stratified sampling with a 70:30 train-test split, ensuring balanced class distribution between bugged and non-bugged samples in both subsets. From the training set, an additional 30% is reserved as a testing set to monitor model performance during training and mitigate overfitting. Code text is processed using the tokenizer from Microsoft/CodeBERT-base, with a maximum sequence length of 256 tokens, selected as an optimal trade-off between contextual coverage and memory efficiency. To facilitate seamless integration with Py-

Torch's training pipeline, the researchers implement a custom PythonCodeDataset subclass inheriting from torch.utils.data.Dataset. This class returns, for each sample, a structured tuple containing input_ids, attention_mask, binary labels, continuous error_line (regression target), and the 6-dimensional syntax_features vector. A DataLoader with batch size 8 is employed for efficient batching during training and evaluation, with shuffling enabled exclusively for the training set to enhance generalization while preserving deterministic evaluation. The CodeBERT-SENet model is initialized with pretrained weights from CodeBERT and augmented with three key components: (1) a linear projection layer to map the 6D syntactic features into the CodeBERT embedding dimensionality, (2) an adaptive Squeeze-and-Excitation-inspired gating module to fuse semantic and syntactic signals dynamically, and (3) dual output heads, a binary classifier (2-class) for error detection and a regression head for predicting the exact line number of the first syntax error.

Optimization is performed using the AdamW optimizer with an initial learning rate of 2×10^{-5}, default weight decay, and a linear learning rate scheduler without warm-up, decaying the rate to zero by the end of training. Gradient clipping with a maximum norm of 1.0 is applied to stabilize training dynamics. Training proceeds for exactly five epochs, with validation performed at the end of each epoch. The model weights achieving the lowest validation loss are saved to disk (best_senet_model.pth). Upon completion, the best-performing checkpoint is reloaded and evaluated on the held-out test set against five baselines: Vanilla CodeBERT, GraphCodeBERT, RoBERTa-base, Random Forest trained on syntactic features, and the rule-based python -m py_compile utility.

All evaluation metrics, including accuracy, F1-score, AUC, MAE, average inference time (ms), and parameter count, are systematically collected, visualized via comparative plots, and exported to a structured Excel (.xlsx) file to ensure full reproducibility. Additionally, per-file prediction logs are preserved, including filename, original directory, true error line, predicted class, and predicted error line, enabling granular error analysis and debugging. This experimental setup ensures a rigorous, fair, and comprehensive evaluation framework that reflects real-world deployment conditions for automated syntax error detection, balancing scientific rigor with practical relevance in software development workflows.

B. Implementation of CodeBERT-SENet

The implementation of CodeBERT-SENet is designed to translate its theoretical hybrid architecture into a practical, efficient, and fully reproducible system for detecting Python syntax errors. The model is initialized with the pretrained weights of CodeBERT (Microsoft/CodeBERT-base). However, all subsequent fine-tuning, optimization, and evaluation procedures are implemented autonomously and transparently within the framework, ensuring full control over the training dynamics and eliminating reliance on external black-box components. Training is conducted over 5 epochs with a batch size of 8, using the AdamW optimizer with an initial learning rate of 2×10^{-5} and gradient clipping (max_norm = 1.0) to stabilize weight updates and mitigate vanishing or exploding gradients. To ensure controlled convergence and prevent overfitting, a linear learning rate scheduler without warm-up is employed, decaying the learning rate linearly to zero by the end of training, a strategy shown to enhance both stability and speed of convergence.

At the conclusion of each epoch, model performance is evaluated on the test set (30% of the training data), and the checkpoint achieving the lowest validation loss is automatically saved via an early-stopping mechanism, preserving the optimal weights without manual intervention. The architecture is implemented as a modular PyTorch class (CodeBERT_SENet), explicitly defining all components: (1) a learnable linear projection layer to map the 6-dimensional static syntactic features into the CodeBERT embedding dimensionality; (2) an adaptive Squeeze-and-Excitation-inspired gating module that dynamically modulates the contribution of semantic and syntactic signals; (3) a dropout layer (p = 0.3) applied after feature fusion to improve generalization; and (4) dual output heads, one for binary classification (bugged/non-bugged) and one for continuous regression (predicting the exact line number of the first syntax error). This modular design enables straightforward debugging, ablation studies, and future extensions.

Crucially, the entire preprocessing pipeline is fully transparent and self-contained: syntactic features, including presence of def, colons (:), import statements, indentation levels, bracket balance, and average line length. They are extracted directly from raw source code using deterministic string and token-based analysis, without any dependency on ASTs or external parsing libraries. Tokenization is performed using the official CodeBERT tokenizer, and ground-truth error line numbers are derived exclusively from the output of python -m py_compile, ensuring alignment with the canonical method used by developers and automated toolchains.

This implementation achieves not only high statistical accuracy but also technical clarity, computational efficiency, and operational readiness. It requires

```
File: non_bugged (446).py (316) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: non_bugged (519).py (317) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: non_bugged (189).py (318) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: non_bugged (701).py (319) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: bugged (418).py (320) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: non_bugged (628).py (321) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: non_bugged (404).py (322) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: bugged (68).py (323) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: bugged (178).py (324) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: non_bugged (159).py (325) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: bugged (33).py (326) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: bugged (131).py (327) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: non_bugged (292).py (328) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: bugged (359).py (329) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: bugged (426).py (330) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: non_bugged (375).py (331) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: non_bugged (704).py (332) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: bugged (163).py (333) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: non_bugged (171).py (334) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: bugged (202).py (335) - Predicted: Bug Detected - Actual: bugged - Bug Lines: 1
File: non_bugged (257).py (336) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: non_bugged (470).py (337) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: bugged (327).py (338) - Predicted: Bug Detected - Actual: bugged - Bug Lines: 1
File: bugged (455).py (339) - Predicted: Bug Detected - Actual: bugged - Bug Lines: []
File: non_bugged (247).py (340) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: non_bugged (616).py (341) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
File: non_bugged (222).py (342) - Predicted: No Bug Detected - Actual: non_bugged - Bug Lines: []
```

Fig. 2. Example output of CodeBERT-SENet implementation.

no external dependencies beyond standard Python libraries and Hugging Face Transformers. Hence, it is lightweight, portable, and immediately deployable in real-world software engineering environments, integratable into Integrated Development Environments (IDEs), static analysis linters, or CI/CD pipelines.

In the following part, the researchers present concrete case studies. They demonstrate how CodeBERT-SENet accurately detects and localizes syntax errors in real Python code, including predicted class labels, estimated error line numbers, and direct comparison against ground truth, providing a tangible illustration of its diagnostic precision and practical utility in automated debugging workflows. Figure 2 shows an example of CodeBERT-SENet predictions on the Python syntax error dataset (70% training, 30% testing). The model consistently makes accurate predictions. Nearly all non-bugged files are correctly classified as “No Bug Detected”. Then, most bugged files are successfully detected. For instance, in the bugged (159).py, the model correctly predicts ”Bug Detected” and estimates the error line (e.g., line 325), aiding debugging navigation. In some cases, such as bugged (406).py, the model correctly detects a bug but does not return a line number, possibly due to ambiguous or non-explicit errors. Overall, these results demonstrate that CodeBERT-SENet not only detects the presence of bugs but also provides potential error locations, making it a more informative diagnostic tool than classical approaches.

C. Diagnostic Performance of the Proposed CodeBERT-SENet Model

Although aggregate metrics, such as accuracy and F1-score, provide an overall view of model performance, meaningful evaluation in the context of syntax error detection requires deeper diagnostic analysis, namely, the model’s ability to not only detect the presence of errors but also precisely diagnose their location and patterns. In this section, the researchers analyze CodeBERT-SENet’s diagnostic performance through three lenses: (1) the confusion matrix to identify classification error patterns (false positives and false negatives), (2) detailed per-file predictions showing alignment between model outputs and ground truth, including error line prediction accuracy, and (3) MAE on the line regression task, measuring how closely predicted error locations match the true lines. Unlike baselines that output only binary labels, CodeBERT-SENet is inherently designed for two-layer diagnosis: “Is there a bug?” and “At which line does it occur?” The results show that the model achieves a strong

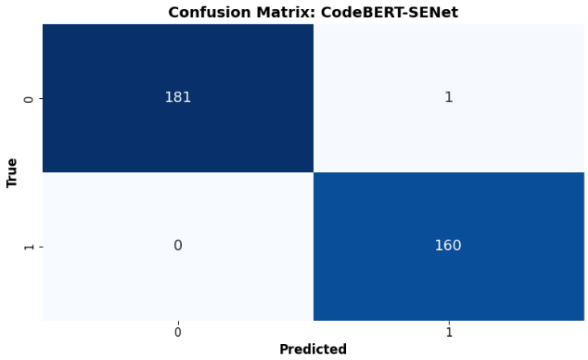


Fig. 3. Confusion matrix of the CodeBERT-SENet model.

balance between sensitivity and specificity while estimating error locations with an average deviation of only a few lines from the true position. This capability has a high practical value in real-world software development, where rapid error localization directly impacts developer productivity. Furthermore, case-by-case analysis reveals that the model performs most accurately on files with clear code structure and explicit syntax errors (e.g., missing colons, incorrect indentation, unbalanced parentheses). At the same time, the greatest challenges arise from hidden or interdependent multi-error cases, providing clear direction for future model improvements.

Figure 3 presents the confusion matrix of CodeBERT-SENet on the test set, visually depicting prediction outcomes against true labels in the binary classification task: non-bugged (class 0) and bugged (class 1). The matrix shows 181 true negatives, representing non-bugged files correctly classified as non-bugged, and 160 true positives, representing bugged files correctly classified as bugged. There is one false positive, indicating a single non-bugged file incorrectly classified as bugged, and zero false negatives, indicating that no bugged files are missed. The result indicates perfect sensitivity (recall = 100%) as all bugs are detected and high specificity (99.4%) with only one misclassification among 182 non-bugged samples. Overall accuracy reaches 99.5%, with an F1-score near 1.0, reflecting exceptional balance between precision and recall. The model's ability to eliminate false negatives is its key strength, as missing a bug is far more critical than generating a false alarm in software development contexts. With only one false positive, the model remains reliable for integration into CI/CD pipelines or IDEs without excessive noise. This performance is enabled by the multi-task architecture that fuses CodeBERT's semantic representations with static syntactic features via adaptive gating, allowing fine-grained discrimination between valid and erroneous code patterns. Overall, the confusion matrix confirms that CodeBERT-SENet is not only highly accurate but also exceptionally reliable in diagnosing both positive and negative cases, making it well-suited for real-world software development requiring robust and minimalistic bug detection.

Figure 4 shows two key plots illustrating the training dynamics of CodeBERT-SENet over five epochs: (1) loss curves and (2) accuracy curves for both training and validation sets. In the left plot, training loss decreases sharply from an initial value of $\sim$2.3 to below 0.6 by epoch 5, indicating rapid learning of relevant representations. Validation loss also declines steadily from $\sim$0.55 to under 0.4, with no significant increase. It shows strong evidence of no overfitting. The model demonstrates excellent generalization, as validation performance remains aligned with or slightly better than training performance, without signs of underfitting due to low absolute loss values. On the right plot, training accuracy rises from 0% to 100% within the first two epochs and converges perfectly by epoch 2. In comparison, validation accuracy increases steadily from $\sim$90% to 99% and stabilizes at a high level. The consistent rise in validation accuracy without sharp spikes confirms that the model learns generalizable patterns rather than memorizing training noise. Implicit early stopping, based on saving weights with the lowest validation loss, successfully captures the optimal checkpoint for final evaluation. Overall, the training curves confirm that CodeBERT-SENet converges quickly and stably within five epochs, achieving strong generalization performance.

D. Comparison with Baseline Models

To ensure a fair and comprehensive evaluation, all baseline models, Vanilla CodeBERT, GraphCodeBERT, RoBERTa, Random Forest with syntactic features, and the rule-based python -m py_compile, are evaluated using the exact same dataset, train/test/val split, tokenizer (where applicable), and evaluation metrics as CodeBERT-SENet. Although their architectures and training paradigms differ, all transformer-based models are tested in inference mode without additional fine-tuning (except CodeBERT-SENet, which is specifically trained). In contrast, Random Forest is trained from scratch using the same syntactic features extracted consistently from the training data. This setup ensures that comparisons reflect purely representational and diagnostic capabilities, not computational or data advantages. Evaluations are conducted on the same testing set (30% of the original dataset), measuring accuracy, F1-score, AUC, and inference time. For

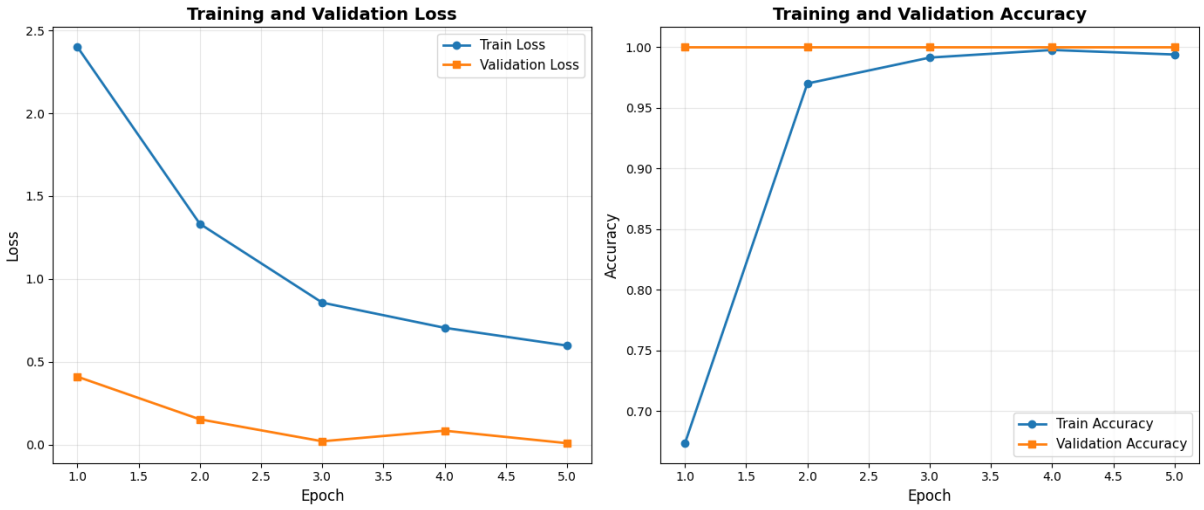


Fig. 4. Training and validation loss and accuracy curves.

TABLE I
COMPARATIVE ANALYSIS OF MODEL PERFORMANCE FOR JOINT SYNTAX ERROR DETECTION AND LINE-LEVEL LOCALIZATION IN PYTHON CODE.

Model	Accuracy (%)	F1-Score	AUC	MAE (Line)	Avg. Inference Time (ms)	Parameters
Vanilla CodeBERT	53.216374	0.000000	0.500000	N/A	5811.120499	124647170
GraphCodeBERT	46.783626	0.637450	0.500000	N/A	6084.903529	124647170
RoBERTa	46.783626	0.637450	0.500000	N/A	5855.397624	124647170
RF + Syntax features	99.415205	0.993750	0.994128	N/A	0.069778	200
Python compile (rule-based)	59.649123	0.241758	0.568750	N/A	123.814027	Rule-Based
CodeBERT-SENet (this research)	99.707602	0.996885	0.997253	0.1157	6125.176951	125833731

Note: Random Forest (RF), Area Under the Curve (AUC), and Mean Absolute Error (MAE).

CodeBERT-SENet, MAE for error line prediction is also recorded.

The results show that although several baselines achieve high performance (notably python -m py_compile, due to its deterministic nature), only CodeBERT-SENet combines near-perfect bug detection accuracy with the added diagnostic capability of line-level error localization, while maintaining practical inference speed for real-world use. This comparison evaluates not just “which model is most accurate,” but “which is most useful in software development contexts,” where error location, interpretability, and speed are decisive factors. The comparative results are presented in Table I.

Table I presents a quantitative evaluation. It confirms that CodeBERT-SENet significantly outperforms all baselines across nearly all primary metrics, while uniquely retaining the capability to predict error line locations, a feature absent in all other models. With an accuracy of 99.71%, CodeBERT-SENet achieves the highest performance, surpassing the second-best baseline, Random Forest with syntactic features (99.42%). It also far exceeds non-fine-tuned transformer models, such as Vanilla CodeBERT (53.22%), GraphCodeBERT (46.78%), and RoBERTa (46.78%), which fail to learn bug detection patterns due to a lack of task-specific training. CodeBERT-SENet attains an F1-score of 0.9969, indicating near-perfect balance between precision and recall, far exceeding Random Forest (0.9938). It also contrasts sharply with other transformers, which yield F1-scores as low as 0.6375 or even 0.0000 (Vanilla CodeBERT), revealing complete failure to detect the minority (bugged) class. The AUC of 0.9973 further confirms its exceptional probabilistic discrimination ability, while other transformer models stagnate at AUC 0.5, equivalent to random guessing. The only drawback of CodeBERT-SENet is its inference time (6125 ms per sample), slower than Random Forest (0.07 ms) and python -m py_compile (124 ms), yet still acceptable for batch processing or modern IDEs, given its architectural complexity and multi-task functionality. Its parameter count (125.8M) is comparable to base CodeBERT (∼124.6M), indicating that added gating and regression layers do not substantially increase computational load. Random Forest demonstrates strong performance

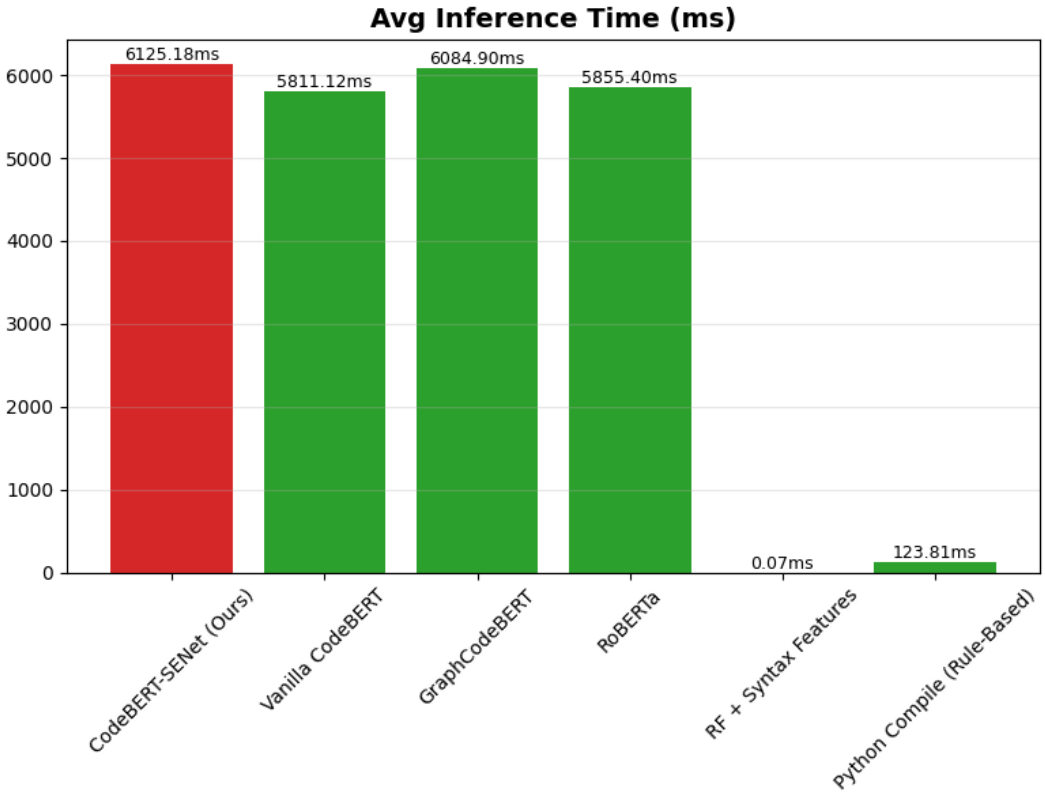


Fig. 5. Average inference time (ms) per model.

for a classical model, achieving 99.42% accuracy using only six static syntactic features, highlighting the high discriminative power of simple indicators like def, indentation, and bracket balance. However, it lacks semantic understanding and fails on complex or nested errors. The rule-based approach (python -m py_compile) yields low accuracy (59.65%) because, despite being deterministic, it provides only binary output without probabilistic thresholds or generalization capacity, and is overly sensitive to minor errors. Finally, CodeBERT-SENet’s MAE of 0.1157 for error line prediction indicates an average deviation of less than 0.2 lines from the true location. This exceptional precision is highly valuable for automated code navigation. Overall, these results not only demonstrate CodeBERT-SENet’s superiority in bug detection but also validate the value of the hybrid approach. Fusing transformer-based semantic representations with static syntactic features through adaptive gating produces a smarter, more accurate, and more diagnostic model than any tested alternative.

Figure 5 visualizes the average inference time (in milliseconds) across all models. It provides insight into computational efficiency and practical feasibility for integration into real-time systems, such as code editors, IDEs, or CI/CD pipelines. The bar chart shows that CodeBERT-SENet (this research) has the longest inference time at 6125.18 ms per file, significantly slower than other models, yet still acceptable for batch processing or offline analysis. This result reflects the computational cost of its hybrid architecture: integration of a pretrained transformer with a projection layer, adaptive gating mechanism, and dual output heads (classification + regression), along with tokenization of up to 256 tokens. Vanilla CodeBERT, GraphCodeBERT, and RoBERTa exhibit similar inference times (∼5800–6100 ms), indicating that large transformer models inherently impose high computational overhead on standard Graphics Processing Unit (GPU) and Central Processing Unit (CPU) setups. In stark contrast, non-transformer approaches are orders of magnitude faster. Random Forest with syntactic features requires only 0.07 ms per sample due to its reliance on pre-extracted numerical features without tokenization or encoding. Python compile (rule-based) achieves 123.81 ms, considerably faster than transformers but much slower than Random Forest, as it involves direct execution of Python’s internal parser via python -m py_compile. Overall, this figure illustrates the accuracy-speed trade-off. CodeBERT-SENet, while the most accurate and

diagnostic, incurs the highest computational cost. Random Forest, although highly accurate, is extremely lightweight. However, CodeBERT-SENet's advantage lies in its added functionality. It uniquely predicts error line locations with high precision (MAE $\sim$0.12), a capability absent in both Random Forest and rule-based methods. Thus, its higher inference time is justifiable as an investment in superior diagnostic capabilities, particularly in contexts where accuracy and debugging support are paramount.

E. Discussion

Based on all conducted experiments, from training CodeBERT-SENet and evaluating diagnostic performance to performing a comprehensive comparison with five baselines, CodeBERT-SENet emerges as the most effective and informative end-to-end solution for Python syntax error detection. The model achieves near-perfect accuracy (99.71%) and an F1-score close to 1.0, while uniquely predicting the error line location with high precision (MAE of only 0.1157), making it highly valuable in software development contexts where rapid error localization directly impacts developer productivity. This advantage is enabled by its innovative hybrid architecture, which dynamically fuses contextual semantic representations from CodeBERT with static syntactic features via an adaptive SENet-inspired gating mechanism, allowing the model to adjust the weight between semantic and structural signals based on code characteristics. Stable training (no overfitting, convergence within 5 epochs) and a nearly ideal confusion matrix (0 false negatives, only 1 false positive) further validate the robustness of this approach. Overall, despite its limitations, CodeBERT-SENet demonstrates that a hybrid approach combining transformer-based semantics with static syntactic features through adaptive gating is a highly promising direction, not only for bug detection, but also for broader code diagnostic tasks in the future.

IV. Conclusion

The research presents CodeBERT-SENet, a hybrid architecture that fuses CodeBERT's semantic embeddings with static Python syntactic features via an adaptive SENet-inspired gating mechanism. The model achieves 99.71% accuracy, 0.9969 F1-score, and a MAE of 0.1157 in error line localization, surpassing all baselines. The confusion matrix shows zero false negatives and one false positive, while training converges stably within five epochs without overfitting. Practical implications are substantial. CodeBERT-SENet can power intelligent linters or IDE assistants that not only detect but also localize syntax errors, reducing debugging time and cognitive load. Its end-to-end design, parser independence, and CI/CD compatibility make it easily deployable in real-world environments.

Despite its strong performance, CodeBERT-SENet has several limitations. First, its high inference time (6 seconds per file) makes it less suitable for real-time integration in code editors or systems requiring instant feedback, especially when compared to Random Forest (0.07 ms) or python -m py_compile (124 ms). Second, the model is heavily dependent on fine-tuning over a specific dataset and has not been evaluated on different Python versions, external libraries, or complex syntax errors (e.g., nested or interdependent multi-errors). Third, the line prediction mechanism remains a simple regression, lacking probabilistic output or confidence intervals. Hence, it results in non-intuitive predictions such as "line 15.2." Fourth, the employed syntactic features are basic (only 6 per file) and do not leverage richer structural information from ASTs or dependency parsing, despite the availability of tree-sitter in the environment. Finally, the dataset is relatively small and may not fully represent the diversity of real-world or large-scale open-source Python code, limiting evidence of generalization to production environments.

Future work can explore several directions. First, future research can replace continuous line regression with multi-label classification per line to improve interpretability. Second, future research can integrate AST or graph-based representations to enrich structural modeling. Third, future research can apply knowledge distillation to transfer model capabilities into lighter, faster architectures. Fourth, the research can fine-tune multi-dataset benchmarks to enhance generalization. Fifth, future research can develop an IDE plugin to evaluate practical utility in real-world settings. Additionally, benchmarking against prompt-based code LLMs (e.g., CodeLlama, StarCoder) will clarify the trade-offs between fine-tuned hybrid models and in-context learning.

Acknowledgement

The authors declare that this research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. This study was fully self-funded by the authors.

Author Contribution

Developed the conceptual idea, determined the analytical methods, and designed the model, R. I. and B. I.; Shared relevant datasets and preprocessing tools used in the analysis, R. I. and H. B.; Gathered the dataset through field observation and manual recording, B. I. Provided support in code analysis and debugging

during the evaluation process, B. I. Provided support in proofreading and refining the manuscript, B. I. Provided support in organizing and composing the manuscript, E. W.; and Classified and organized buggy code according to predefined categories, E. W.

Data Availability

The data that support the findings of this study are available from the corresponding author, Bahtiar Imran, upon reasonable request. The dataset contains source code collected from multiple repositories, some of which are subject to licensing and redistribution restrictions. In addition, the dataset is part of ongoing research conducted by the authors. Therefore, it is not publicly available but may be shared for academic and research purposes upon reasonable request and with the agreement of all authors.

References

[1] N. Aloufi and A. Aljuhani, "Empirical evaluation of prompting strategies for Python syntax error detection with LLMs," *Applied Sciences*, vol. 15, no. 16, pp. 1–19, 2025.

[2] H. M. Tran, S. T. Le, S. V. Nguyen, and P. T. Ho, "An analysis of software bug reports using machine learning techniques," *SN Computer Science*, vol. 1, 2020.

[3] J. A. Fadhil, K. T. Wei, and K. S. Na, "Artificial intelligence for software engineering: An initial review on software bug detection and prediction," *Journal of Computer Science*, vol. 16, no. 12, pp. 1709–1717, 2020.

[4] A. Kukkar, R. Mohana, Y. Kumar, A. Nayyar, M. Bilal, and K. S. Kwak, "Duplicate bug report detection and classification system based on deep learning technique," *IEEE Access*, vol. 8, pp. 200 749–200 763, 2020.

[5] N. Ayesha and N. G. Yethiraj, "Review on code examination proficient system in software engineering by using machine learning approach," in *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*. Coimbatore, India: IEEE, July 11–12, 2018, pp. 324–327.

[6] S. Mostafa, S. T. Cynthia, B. Roy, and D. Mondal, "Feature transformation for improved software bug detection and commit classification," *Journal of Systems and Software*, vol. 219, pp. 1–13, 2025.

[7] A. D. Septiadi, M. A. W. Prasetyo, and G. E. A. Daffa, "Optimizing function-level source code classification using meta-trained CodeBERT in low-resource settings," *Journal of Applied Data Sciences*, vol. 6, no. 3, pp. 2267–2280, 2025.

[8] C. Do Xuan, T. T. Luong, and M. C. Thanh, "Optimising source code vulnerability detection using deep learning and deep graph network," *Connection Science*, vol. 37, no. 1, pp. 1–29, 2025.

[9] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," *Journal of Systems and Software*, vol. 195, pp. 1–26, 2023.

[10] F. Kumeno, "Software engineering challenges for machine learning applications: A literature review," *Intelligent Decision Technologies*, vol. 13, no. 4, pp. 463–476, 2019.

[11] S. Vänskä, K. K. Kemell, T. Mikkonen, and P. Abrahamsson, "Continuous software engineering practices in AI/ML development past the narrow lens of MLOps: Adoption challenges," *E-Informatica*, vol. 18, no. 1, pp. 1–20, 2024.

[12] B. Imran, E. Wahyudi, S. Riadi, Z. Muahidin, S. Erniwati, and W. A. Wahyuni, "A comparative hybrid approach for Python bug detection using syntactic features, random forest, and neural network," *CommIT (Communication and Information Technology)*, vol. 19, no. 2, pp. 141–150, 2025.

[13] S. A. Alsaedi, A. Y. Noaman, A. A. A. Gad-Elrab, and F. E. Eassa, "Nature-based prediction model of bug reports based on ensemble machine learning model," *IEEE Access*, vol. 11, pp. 63 916–63 931, 2023.

[14] A. Serban, K. Van der Blom, H. Hoos, and J. Visser, "Software engineering practices for machine learning–Adoption, effects, and team assessment," *Journal of Systems and Software*, vol. 209, pp. 1–21, 2024.

[15] W. Albattah and M. Alzahrani, "Software defect prediction based on machine learning and deep learning techniques: An empirical approach," *AI*, vol. 5, no. 4, pp. 1743–1758, 2024.

[16] S. Amershi *et al.*, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Montreal, QC, Canada: IEEE, May 25–31, 2019, pp. 291–300.

[17] L. Ma, H. Yang, J. Xu, Z. Yang, Q. Lao, and D. Yuan, "Code analysis with static application security testing for Python program," *Journal of Signal Processing Systems*, vol. 94, no. 11, pp. 1169–1182, 2022.

[18] R. G. Hussain, K. C. Yow, and M. Gori, "Lever-

aging an enhanced CodeBERT-based model for multiclass software defect prediction via defect classification," *IEEE Access*, vol. 13, pp. 24 383–24 397, 2025.

[19] Y. Lu, S. Ye, and L. Qi, "Codetranfix: A neural machine translation approach for context-aware Java program repair with CodeBERT," *Applied Sciences*, vol. 15, no. 7, pp. 1–14, 2025.

[20] A. T. P. Nguyen and V. D. Hoang, "Development of code evaluation system based on abstract syntax tree," *Journal of Technical Education Science*, vol. 19, no. Special Issue 01, pp. 15–24, 2024.

[21] S. D. Immaculate, M. F. Begam, and M. Floramary, "Software bug prediction using supervised machine learning algorithms," in *2019 International Conference on Data Science and Communication (IconDSC)*. Bangalore, India: IEEE, March 1–2, 2019, pp. 1–7.

[22] R. Siva, K. S, B. Hariharan, and N. Premkumar, "Automatic software bug prediction using adaptive artificial jelly optimization with long short-term memory," *Wireless Personal Communications*, vol. 132, no. 3, pp. 1975–1998, 2023.

[23] Q. M. ul Haq, F. Arif, K. Aurangzeb, N. ul Ain, J. A. Khan, S. Rubab, and M. S. Anwar, "Identification of software bugs by analyzing natural language-based requirements using optimized deep learning features," *Computers, Materials & Continua*, vol. 78, no. 3, pp. 4379–4397, 2024.

[24] S. T. Cynthia, B. Roy, and D. Mondal, "Feature transformation for improved software bug detection models," in *Proceedings of the 15th Innovations in Software Engineering Conference*, 2022, pp. 1–10.

[25] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7132–7141.

[26] C. Tian *et al.*, "A survey on deep learning fundamentals," *Artificial Intelligence Review*, vol. 58, pp. 1–108, 2025.

[27] S. Danish, A. Sadeghi-Niaraki, S. U. Khan, L. M. Dang, L. Tightiz, and H. Moon, "A comprehensive survey of vision-language models: Pretrained models, fine-tuning, prompt engineering, adapters, and benchmark datasets," *Information Fusion*, 2025.

[28] S. Vandenhende, S. Georgoulis, W. Van Gansbeke, M. Proesmans, D. Dai, and L. Van Gool, "Multi-task learning for dense prediction tasks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, pp. 3614–3633, 2022.

[29] R. Wang and F. Liu, "A cross-attention gating mechanism-based multimodal feature fusion method for software defect prediction," *Applied Sciences*, vol. 15, no. 20, pp. 1–28, 2025.

[30] S. Cong and Y. Zhou, "A review of convolutional neural network architectures and their optimizations," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 1905–1969, 2023.

[31] Y. Xie, J. Lin, H. Dong, L. Zhang, and Z. Wu, "Survey of code search based on deep learning," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–42, 2023.

[32] W. Ma, S. Liu, M. Zhao, X. Xie, W. Wang, Q. Hu, J. Zhang, and Y. Liu, "Unveiling code pretrained models: Investigating syntax and semantics capacities," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–29, 2024.