SSN-CSE-CODECATALYSTS at SemEval-2026 Task 13: Integrating Transformer Semantics and AST-Derived Structural Features for AI-Generated Code Detection.

Bhuvana J[1], Ramanan Mahendran[1], Siddharth Chandrasekar S[1]
Pragatheesh J[1], Rethanya P[1]
[1]Sri Sivasubramaniya Nadar College of Engineering, Chennai, India

Abstract

Pre-trained transformers often struggle with multi-lingual code classification due to sequence length constraints and difficulties in explicitly capturing deep structural complexities. To address this for SemEval Task 13 (Orel et al., 2026b), a hybrid neural architecture that fuses CodeBERT's semantic embeddings is proposed. Handcrafted software engineering metrics is presented, with a Head+Tail truncation strategy to preserve crucial logic in long sequences while simultaneously extracting explicit Abstract Syntax Tree (AST) features via tree-sitter—including maximum depth, branching factor, and cyclomatic complexity. By integrating dense language model representations with explicit structural heuristics, this work provides a robust and scalable solution for enhanced code classification.

1 Introduction

Source code classification is a fundamental task in automated software engineering, increasingly addressed using Natural Language Processing (NLP) techniques as multi-lingual codebases grow in scale and diversity. Pre-trained transformer models such as CodeBERT and RoBERTa (Liu et al., 2019) learn rich semantic representations of source code, but real-world deployment exposes key limitations, including strict input sequence length constraints and weak modeling of explicit hierarchical structure. Transformers process code as linear token sequences, capturing semantic relationships while failing to encode deterministic structural rules, and the standard 512-token limit often truncates large files, removing critical control-flow logic. SemEval-2026 Task 13 motivates robust multi-lingual solutions to these challenges, leading to a late-fusion architecture that integrates CodeBERT embeddings with deterministic structural features extracted via `tree-sitter`, including AST depth, branching factor, cyclomatic complexity, nesting depth, and lexical ratios. A Head+Tail truncation strategy preserves structurally salient code regions, enabling effective fusion of semantic and structural signals across diverse programming languages.

2 Background

SemEval Task 13 addresses the critical challenge of distinguishing human-written source code from snippets generated by Large Language Models (LLMs) by requiring binary classification into Human (0) or Machine (1) labels. The task places a specific emphasis on Out-Of-Distribution (OOD) generalization, providing training data in C++, Python, and Java, while challenging models with unseen languages like Go, PHP, C, C, and JavaScript, as well as unseen domains involving research and deployed code. Evaluation is primarily measured using the Macro F1-score as the lead leaderboard metric. Although prior work in code classification has been dominated by pre-trained Transformer models such as the bimodal NL-PL representations of CodeBERT (Feng et al., 2020), the structural data-flow integration of GraphCodeBERT (Guo et al., 2021), the layer-wise analysis of EL-CodeBERT (Liu et al., 2022), and the prompt-based learning of CodePrompt (2024), this method shifts focus toward injecting domain-specific software metrics directly into the decision function. By utilizing a late-fusion design, dense CodeBERT semantic vectors are combined with discrete Abstract Syntax Tree (AST) features, such as nesting depth and branching factors, alongside statistical heuristics like cyclomatic complexity. This integration of learned semantic embeddings with deterministic code-quality features aims to improve robustness when semantic signals degrade in unseen languages, aligning with the rigorous OOD requirements of the SemEval-2026 Task 13 evaluation.

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 1027–1031

3 System Overview

The proposed system is designed to address challenges in detecting machine-generated code in multi-lingual, out-of-distribution (OOD) settings. Large Language Models often produce syntactically valid but structurally generic code, whereas human-written code exhibits distinctive nesting and complexity patterns. Standard transformer models, constrained by sequence length limits, frequently truncate the sections of code where these signatures occur.

To address this limitation, a Late-Fusion Hybrid Architecture is introduced that combines dense semantic representations from a pre-trained transformer with explicit, handcrafted software engineering metrics as shown in Figure 1.

3.1 Feature Extraction Strategy

The model assumes that human and machine code differ in graph complexity and lexical density. To capture this, a feature engineering pipeline extracts a 7-dimensional vector, $v_{stat} \in \mathbb{R}^7$, for each code sample.

A. Abstract Syntax Tree (AST) Analysis The model uses Tree-Sitter (Brunsfeld, 2018) to parse raw code into its constituent AST. Unlike regex-based heuristics, this enables measurement of true topological code complexity.

- **Max Depth (d_{max}):** The longest path from the root to a leaf node, serving as a proxy for algorithmic complexity.
- **Average Branching Factor (b_{avg}):** The ratio of total child nodes to non-leaf nodes, capturing logical "width".

B. Cyclomatic Complexity (v_{cc}) Since full control-flow graph construction is computationally expensive at scale, Cyclomatic Complexity (McCabe, 1976) is approximated by counting branching keywords:

$$v_{cc} = \sum_{k \in K} \text{count}(k) + 1 \quad (1)$$

Where K = {'if', 'for', 'while', 'case', 'catch', '&&', '||', '?' }.

C. Lexical & Nesting Metrics

- **Nesting Depth (v_{nd}):** The maximum depth of nested blocks (defined by { } or ()), capturing the "spaghetti code" tendency often found in human submissions.
- **Lexical Ratios:** The approach computes the average identifier length (l_{id}), comment-to-code ratio (r_{com}), and keyword-to-identifier ratio (r_{kw}) to encode stylistic signatures.

3.2 Head+Tail Truncation Strategy

A significant challenge in this task is the 512-token limit of BERT-based models. Standard "head-only" truncation discards the end of the file, which often contains critical return statements or closing logic. A *Head+Tail Truncation* strategy is implemented to preserve the distinct structural boundaries of the code.

Given a tokenized sequence T of length $L > 510$, the truncated sequence T_{trunc} is defined as:

$$T_{trunc} = [T_0, \ldots, T_{254}] \oplus [T_{L-255}, \ldots, T_{L-1}] \quad (2)$$

The pseudocode for the implementation is provided in Algorithm 1.

Algorithm 1 Head+Tail Truncation Strategy

procedure APPLYTRUNCATION($tokens, max\_len = 512$)
  $special\_tokens \leftarrow 2$ ▷ [CLS] and [SEP]
  $capacity \leftarrow max\_len - special\_tokens$
  if $length(tokens) \leq capacity$ **then**
    return $[CLS] + tokens + [SEP] + Padding$
  end if
  $half \leftarrow capacity//2$
  $head \leftarrow tokens[0 : half]$
  $tail \leftarrow tokens[length(tokens) - half : length(tokens)]$
  return $[CLS] + head + tail + [SEP]$
end procedure

3.3 Hybrid Neural Architecture

This classification model is a late-fusion network that integrates three distinct feature modalities.

This approach employs CodeBERT (`microsoft/codebert-base`) as the primary encoder. CodeBERT is a bimodal transformer model pre-trained on the CodeSearchNet dataset. For an input sequence X, the model outputs a sequence of hidden states. The embedding of the first token (`[CLS]`) is utilized as the aggregate semantic representation of the code:

$$v_{bert} = \text{CodeBERT}(X)_{[CLS]} \in \mathbb{R}^{768} \quad (3)$$

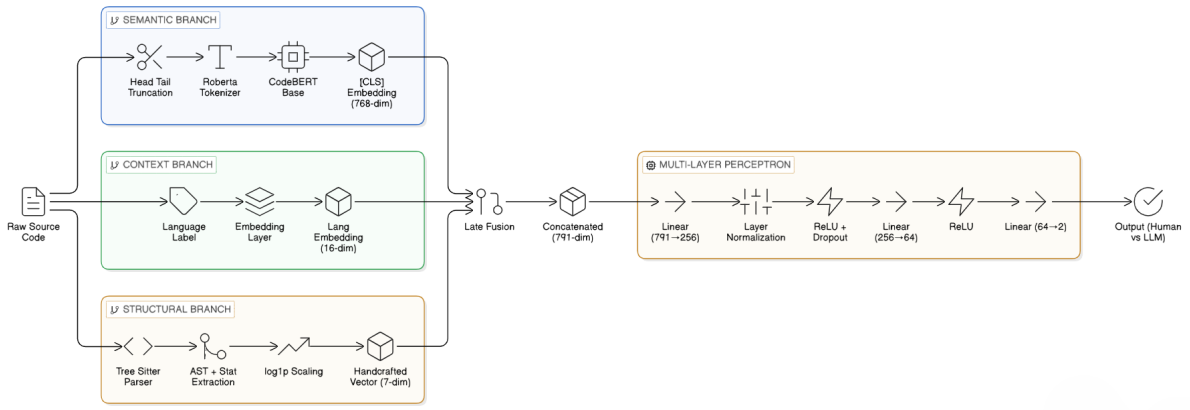

Figure 1: Architectural Overview of the HybridCode Framework

Language Embedding To improve generalization across the diverse language set (Python, C++, Java, Go, PHP, etc.),a learnable language embedding layer is introduced. The language label L is mapped to a dense vector:

$$v_{lang} = \text{Embedding}(L) \in \mathbb{R}^{16} \tag{4}$$

This allows the network to condition its structural expectations (e.g., "braces are common in Java but not Python") on the specific language context.

Late Fusion & Classification Head The final representation v_{final} is constructed by concatenating the semantic, language, and statistical vectors:

$$v_{final} = \text{Concat}(v_{bert}, v_{lang}, v_{stat}) \in \mathbb{R}^{768+16+7} \tag{5}$$

This fused vector is passed through a Multi-Layer Perceptron (MLP) designed with Layer Normalization to stabilize the disparate feature scales:

- **Linear Projection:** $h_1 = W_1 v_{final} + b_1$ (Output dim: 256)
- **Normalization & Activation:** $h_2 = \text{ReLU}(\text{LayerNorm}(h_1))$
- **Regularization:** $h_3 = \text{Dropout}(h_2, p = 0.2)$
- **Bottleneck:** $h_4 = \text{ReLU}(W_2 h_3 + b_2)$ (Output dim: 64)
- **Logits:** $y = W_3 h_4 + b_3$ (Output dim: 2)

3.4 Concrete Execution Example

To illustrate the pipeline, consider a C++ snippet where `tree-sitter` detects a max nesting of 2 and a cyclomatic complexity of 2.

1. **Feature Extraction:** Vector $v_{stat} = [2.0, 1.0, 3.5, 0.2, \dots]$ is computed.
2. **Forward Pass:** The snippet generates v_{bert} (768-d) and v_{lang} (16-d for C++).
3. **Fusion:** These are concatenated into a 791-d vector for final classification.

3.5 Resources & Optimization

This approach utilizes pre-compiled `tree-sitter` binaries and `microsoft/codebert-base` weights. To manage computational costs, **Mixed Precision Training (AMP)** with `bfloat16` and **Gradient Checkpointing** is employed, reducing VRAM usage by 60% and allowing a batch size of 160 on a single NVIDIA RTX A4000 GPU.

4 Experimental Setup

4.1 Data Splitting and Utilization

The official SemEval-2026 Task 13 dataset (Orel et al., 2026a) provided is utilized in Parquet format. The data was split into three distinct sets to ensure robust evaluation:

- **Training Set:** The approach utilized the full `train.parquet` dataset, consisting of 500,000 labeled code samples, to fine-tune the model parameters.
- **Validation Set:** A separate `validation.parquet` set of 100,000 samples was used for hyperparameter tuning and model checkpointing.
- **Test Set:** The final performance was evaluated on `test.parquet`, consisting of 1,000 unseen samples, covering both in-distribution and out-of-distribution (OOD) programming languages.

4.2 Preprocessing and Feature Engineering

Raw source code underwent a preprocessing pipeline:

- **Text Normalization:** Code samples were processed with `normalize_whitespace` to collapse redundant spaces and newlines into single spaces, lowering token count while preserving semantics.
- **Tokenization and Truncation:** `RobertaTokenizer` from `microsoft/codebert-base` was applied. To respect the 512-token limit, a custom Head+Tail strategy retained the first 254 and last 254 tokens (plus `[CLS]` and `[SEP]`), discarding the middle segment to preserve initialization and return logic.
- **Handcrafted Feature Scaling:** A 7-dimensional structural feature vector (e.g., AST depth, cyclomatic complexity) was normalized with `torch.log1p` to reduce outlier and skew impact before concatenation with neural embeddings.

4.3 Implementation Details

Experiments used PyTorch and Hugging Face Transformers, with random seed 42 for Python, NumPy, and PyTorch.

Hardware and Optimization: Training ran on a single NVIDIA RTX A4000 GPU. To increase throughput, Mixed Precision Training (`torch.amp`, `bfloat16`) and Gradient Checkpointing on the CodeBERT backbone were enabled, reducing VRAM usage and permitting a batch size of 160.

Hyperparameter Configuration: The model was trained for 3 epochs using the parameters detailed in Table 1.

4.4 Evaluation Measures

The primary evaluation metric for this task is the Macro F1-Score. We computed this metric at the end of each training epoch on the validation set to monitor convergence and select the optimal model checkpoint.

$$\text{Macro F1} = \frac{1}{N}\sum_{i=1}^{N}\frac{2 \times P_i \times R_i}{P_i + R_i} \tag{6}$$

Where P_i and R_i are the precision and recall for class i, and N is the number of classes (2: Human vs. Machine).

Hyperparameter	Value	Rationale
Batch Size	160	Maximized for 16GB VRAM w/ grad. checkpointing.
Learning Rate	5×10^{-5}	Peak LR for the OneCycle scheduler.
Optimizer	AdamW	Used with `fused=True` for faster computation.
Scheduler	OneCycleLR	Linear warmup followed by cosine annealing.
Weight Decay	0.01	Standard regularization for AdamW.
Dropout	0.2	Applied to the classification head.

Table 1: Hyperparameter configuration for model training.

5 Results

5.1 Main Quantitative Findings

The proposed system was evaluated on the official SemEval-2026 Task 13 (Subtask A) blind test set, achieving a **Macro F1-score of 0.32826** and ranking **68th** on the global leaderboard. The reported score corresponds to the best-performing configuration among three submitted attempts. Despite successful end-to-end inference, the results reveal a substantial performance gap relative to leading submissions, primarily due to limited generalization to Out-Of-Distribution programming languages in the test set.

5.2 Quantitative Analysis and Ablations

A sharp contrast exists between validation (In-Distribution) and official test (OOD) performance, as summarized in Table 2.

Split	Macro F1	Nature of Data
Validation	0.9882	In-Distribution (C++, Java, Python)
Official Test	0.3283	OOD (Go, PHP, C#, etc.)

Table 2: Comparison of performance between In-Distribution and OOD sets.

In-Distribution (Validation) Performance: The Late-Fusion architecture achieved a peak Macro F1-score of 0.9882 at Epoch 3, indicating effective separation of human-written and machine-generated code within known programming languages.

Out-Of-Distribution (Test) Performance: Performance declined to 0.328 on the official test set, indicating limited robustness to unseen languages. Reliance on language-specific components, including the *Language Embedding* layer and `tree-sitter` AST parsers, caused brittleness.

Ablation Insight: The Structural Pathway (AST features) was sensitive to parser availability. For supported languages, depth and branching statistics provided discriminative signals. For unsupported languages, null or zero-valued AST outputs were treated as class-specific patterns, causing systematic misclassification.

5.3 Error Analysis

Prediction behavior reveals the following failure modes:

- **"Zero-Feature" Trap:** For OOD languages, structural metrics often defaulted to zero, causing the MLP head to overfit in-distribution complexity ranges and reducing Recall.
- **Confusion Matrix Bias:** The low F1-score reflects a skewed Confusion Matrix with a high False Negative rate under OOD conditions.
- **Truncation Artifacts:** Head+Tail truncation preserved boundary logic but introduced semantic drift due to language-dependent verbosity.

Overall, near-perfect in-distribution performance ($F1 \approx 0.99$) contrasted with limited zero-shot generalization, as rigid structural metrics degraded robustness in unseen languages.

6 Conclusion

HybridCode was presented for SemEval-2026 Task 13 (Subtask A) to distinguish human-written and machine-generated source code. The proposed system employed a late-fusion architecture combining CodeBERT semantic embeddings with handcrafted structural features derived from `tree-sitter` AST analysis, supported by a Head+Tail truncation strategy to preserve salient code regions. While strong in-distribution performance was observed (Macro F1-score of 0.9882), the proposed system ranked 68 on the official leaderboard with a test F1-score of 0.32826, revealing a substantial generalization gap. Explicit structural features proved highly discriminative for known languages but reduced robustness in out-of-distribution settings where parsers failed or returned null outputs, indicating that rigid structural priors hinder multilingual generalization.

Acknowledgments

Appreciation is extended to the Department of Computer Science and Engineering at Sri Sivasubramaniya Nadar College of Engineering, Chennai, India, for providing computational resources and academic support.

References

Max Brunsfeld. 2018. Tree-sitter: a new incremental parsing system for programming tools.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Luo, Ziyuan Wang, Shujie Liao, Horacio Itiki, Jianfeng Zhou, Jianchi Chen, Tijn Zhou, Nan Duan, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*.

Jing Liu and 1 others. 2022. El-codebert: Layer-wise representations for code smell detection. *Journal of Software Engineering Research and Development*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Thomas J McCabe. 1976. *A complexity measure*. IEEE Transactions on Software Engineering.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026a. AICD bench: A challenging benchmark for AI-generated code detection. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, Rabat, Morocco. Association for Computational Linguistics.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026b. SemEval-2026 task 13: Detecting machine-generated code with multiple programming languages, generators, and application scenarios. In *Proceedings of the 20th International Workshop on Semantic Evaluation (SemEval-2026)*, San Diego, USA. Association for Computational Linguistics.