Team_SLS at SemEval-2026 Task 13: Detecting Machine-Generated Code with CodeBERT and Structural Features

Sai Laasya Gorantla and Shreemithra Naveen and Steven Bethard
University of Arizona
College of Information Science
{saigorantla,shreemithran,bethard}@arizona.edu

Abstract

We describe our system for SemEval-2026 Task 13 Subtask A (Orel et al., 2026), which focuses on detecting whether source code is written by a human or generated by an AI system. We propose a hybrid approach that combines semantic embeddings from CodeBERT with lightweight, language-agnostic structural features extracted using Tree-sitter (Brunsfeld et al., 2026). We compute normalized structural ratios such as nesting depth, logic density, complexity per line, average line length, and punctuation frequency. These structural signals are concatenated with CodeBERT embeddings and passed to a linear classifier for binary prediction. Experimental results on the official validation split show that combining semantic and normalized structural representations substantially improves the model's detection performance on seen-language distributions. However, results on unseen test data reveal significant performance degradation under cross-language distribution shifts. On the official leaderboard, our system ranked 47th out of 81 participating teams.

1 Introduction

Recent advances in large language models have significantly improved automatic code generation across multiple programming languages. Modern AI systems can produce syntactically correct and stylistically consistent code, making it increasingly difficult to distinguish between human-written and machine-generated programs. This creates challenges in areas such as academic integrity, authorship verification, and software provenance.

SemEval-2026 Task 13 (Orel et al., 2026) addresses this problem by providing a multilingual benchmark for detecting AI-generated source code. The task requires systems to classify code snippets as either human-written or machine-generated across multiple programming languages and generators. The diversity of languages and the presence of stylistic overlap between human and AI-generated code make this task challenging.

We propose a hybrid detection approach that combines semantic representations from CodeBERT with structural features extracted using Tree-sitter. CodeBERT captures contextual and semantic information from source code, while Tree-sitter enables extraction of syntactic structure such as nesting depth, functions, loops, and conditional statements. These structural signals complement semantic embeddings by providing additional information about code organization.

Our main contributions are:

- Evaluating the combination of semantic embeddings from CodeBERT with structural features from Treesitter in a unified classifier.
- Analysis showing that our model fails to generalize to unseen programming languages, overfitting to the training data that was highly skewed towards Python.

2 Related Work

Pretrained transformer models have significantly advanced code understanding tasks. Early approaches such as GLTR (Gehrmann et al., 2019) relied on statistical irregularities in token distributions to detect machine-generated text. CodeBERT (Feng et al., 2020) trained a language model on large-scale code corpora paired with natural language and demonstrated strong performance on code classification, search, and defect detection tasks. Codex (Chen et al., 2021), a language model fine-tuned on publicly available code from GitHub, demonstrated strong performance in automatic code generation. And recent work such as DetectGPT (Mitchell et al., 2023) has shown that probabilities from a model of interest can be combined with random permutations of the input text for machine-generated text detection.

Prior work on code stylometry has demonstrated

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2520–2526

that patterns from abstract syntax trees can reveal authorship signals (Caliskan-Islam et al., 2015). Parsing frameworks such as Tree-sitter (Brunsfeld et al., 2026) enable consistent extraction of syntactic features across programming languages. Our work builds on these ideas by integrating pretrained semantic embeddings from large language models with structural signals from abstract syntax trees in a hybrid detection framework.

3 Pilot Studies

Before developing our final model, we conducted exploratory experiments to better understand the differences between human-written and machine-generated code. These pilot studies helped guide our final system design.

3.1 Structural Features Using Python AST

In our first experiment, we extracted structural features using Python's built-in `ast` module. We collected statistics such as the number of functions, loops, and imports. While these structural features captured some variation between human-written and AI-generated code, they achieved only 60-65% accuracy on a sample of the training data, and thus were not sufficiently strong when used alone.

3.2 LLMs as Direct Classifiers

In the second experiment, we evaluated whether existing large language models could directly classify code snippets without fine-tuning. We prompted several LLMs to distinguish between human-written and machine-generated code on a small subset of the training dataset. Although 55–60% of predictions were correct, the outputs were often inconsistent or overly confident, indicating limited reliability for this task.

3.3 Cross-LLM Generation and Detection

In the third experiment, we explored cross-LLM behavior. One model was used to generate code snippets, while another model attempted to classify them as human-written or AI-generated. We also asked a model to imitate human-written code styles and evaluated whether another model could detect the imitation. On a small subset of the training dataset, models correctly detected 50–65% of the AI-generated code. As AI models better mimic human style, detection accuracy decreases, highlighting the difficulty of the task.

3.4 Structural Feature Analysis

We first explored the structural properties of code using syntax tree analysis. Using Tree-sitter parsers, we extracted structural statistics such as syntax tree depth, the number of loops, conditional statements, and function definitions. These features capture structural complexity and organizational patterns in code.

Initial experiments showed that structural features alone provided limited discriminative power. While some differences between human-written and machine-generated code were observable, structural signals were insufficient to achieve high classification accuracy when used independently. (See Section 6.4 for a formal experiment with only Tree-sitter features.)

3.5 Semantic Representation with CodeBERT

We then evaluated the semantic embeddings generated by CodeBERT. CodeBERT provides contextual representations that capture variable usage, naming conventions, and programming patterns. These embeddings demonstrated strong performance in capturing semantic differences between human-written and AI-generated code. (See Section 6.4 for a formal experiment with only CodeBERT features.)

3.6 Feature Combination Motivation

Based on these observations, we hypothesized that combining semantic and structural features could improve detection performance. Semantic embeddings capture contextual information, while structural features capture syntactic organization. This complementary information motivated the development of our hybrid model, which integrates both feature types for classification.

4 Data

For SemEval-2026 Task 13 Subtask A, we use the official dataset released by the task organizers (Orel et al., 2026). The dataset is distributed in Parquet format and includes predefined splits for training, validation, and testing. The official splits contain:

train.parquet: 500,000 labeled code samples,
validation.parquet: 100,000 labeled code samples,
test.parquet: 500,000 unlabeled samples for final evaluation.

We used the validation set to evaluate model performance during training. In all splits, each sample includes:

code: the source code snippet,
label: binary indicator (0 = human-written, 1 = AI-generated),
language: programming language (e.g., Python, Java, C++, PHP, Go, C#),
generator: the generating model, when applicable.

The dataset spans multiple programming languages and generators, creating structural and stylistic variability. This multilingual and multi-generator setup motivates the integration of language-agnostic structural representations alongside semantic embeddings.

5 Methodology

Our approach integrates contextual semantic embeddings from CodeBERT with structural representations derived from abstract syntax trees (ASTs). The system consists of preprocessing, structural feature extraction, and neural classification.

5.1 Preprocessing

To prevent the model from exploiting superficial cues, we remove inline and block comments from all code snippets using regular expressions. This ensures that the classifier does not rely on metadata or explicit references to generative tools.

Since the dataset spans multiple programming languages, we implement heuristic language detection using keyword-based rules (e.g., `const`, `func`, `<?php`, `public class`, `#include`) to infer the appropriate grammar for parsing. Cleaned code snippets are tokenized using the CodeBERT tokenizer with a maximum sequence length of 512 tokens.

5.2 Structural Feature Extraction

Each code snippet is parsed using Tree-sitter. We recursively traverse the AST to compute structural statistics in a language-agnostic manner. Logical constructs are identified through generic node-type matching such as `if`, `for`, `while`, `switch`, and function-related nodes.

To ensure cross-language comparability and robustness to varying snippet lengths, structural statistics are normalized by the number of lines in the code. The resulting five structural ratios capture:

Nesting depth: maximum AST depth
Logic density: number of logical nodes per line
Structural complexity: nesting depth divided by number of lines
Average line length: characters per line
Parenthesis frequency: parentheses per line

If parsing fails, a zero vector is returned to maintain training stability.

The five structural features were selected to capture language-agnostic patterns that distinguish human-written from AI-generated code, without relying on language-specific syntax. Each feature targets a distinct axis of structural behaviour:

- **Nesting depth** reflects control flow structure. Human code tends to exhibit deeper nesting due to iterative problem-solving, whereas AI-generated code produces comparatively flatter structures.
- **Logic density** (logical nodes per line) measures the concentration of decision-making per line. AI models tend to distribute logic across more lines, resulting in a lower ratio.
- **Complexity per line** normalises nesting depth by code length, making the feature comparable across snippets of varying sizes.
- **Average line length** captures stylistic consistency. AI-generated code tends toward uniform, moderate line lengths following style conventions, while human-written code exhibits greater variability.
- **Punctuation rhythm** (parentheses per line) proxies function call density. AI code more frequently decomposes problems into helper function calls, elevating this ratio.

5.3 Model Architecture

We use the pretrained `microsoft/codebert-base` model to obtain 768-dimensional contextual embeddings, using the `[CLS]` token representation as a fixed-length semantic vector.

The five-dimensional structural ratio vector is projected into a 256-dimensional representation through a feed-forward network consisting of:

- Linear ($5 \rightarrow 128$)
- Layer Normalization
- ReLU activation
- Dropout (0.4)
- Linear ($128 \rightarrow 256$)

Layer Normalization is chosen over Batch Normalization to improve robustness under distribution shifts across programming languages. The

resulting 256-dimensional structural embedding is concatenated with the 768-dimensional semantic embedding, forming a 1024-dimensional fused representation.

During validation and inference, sigmoid activation is applied to the model outputs, and predictions are generated using a fixed threshold of 0.5. To enhance generalization, we apply multi-sample dropout during classification by passing the fused vector through multiple dropout layers with increasing rates (0.1–0.5) and averaging the logits before applying the sigmoid activation. The final layer outputs a single logit for binary classification.

6 Experiments

We evaluate our hybrid semantic-structural model on SemEval-2026 Task 13 Subtask A. Experiments are conducted using the official training split described in Section 4.

6.1 Training Setup

The model is trained for two epochs using the AdamW optimizer with a learning rate of 2×10^{-5}. Binary cross-entropy loss with logits is used as the training objective, and the batch size is set to 16.

6.2 Evaluation Metrics

Model performance is evaluated using Macro F1-score, which is the primary evaluation metric for the task. Macro F1 computes the unweighted average of F1-scores across both classes and is appropriate for binary classification with potential class imbalance. We also report accuracy for completeness. The official leaderboard ranking for Subtask A is based on Macro F1-score computed on the test set.

6.3 Reproducibility Details

All experiments were conducted using PyTorch and HuggingFace Transformers on a single NVIDIA GPU. A fixed random seed was used for training to ensure reproducibility. Structural features were extracted offline prior to training to maintain deterministic inputs.

6.4 Ablation Analysis

To evaluate the contribution of the structural features, we compared our hybrid model against two baselines: (1) a semantic-only model using CodeBERT embeddings, and (2) a syntactic-only model using the five extracted structural ratios passed through the MLP. Table 1 shows that while semantic embeddings provide the strongest signal, the addition of structural ratios provides a critical performance boost on the validation set.

Model	**P**	**R**	**Macro F_1**	**A**
CodeBERT (Semantic)	0.942	0.938	0.940	
Tree-sitter (Syntactic)	0.615	0.589	0.602	
Hybrid (Proposed)	**0.975**	**0.974**	**0.975**	0.975

Table 1: Performance on the official validation set. (P: Precision, R: Recall, A: Accuracy).

7 Results and Discussion

Our team participated in Subtask A of SemEval-2026 Task 13, achieving a final score of 0.49655 on the official leaderboard, ranking 47th out of 81 participating teams. On the validation set, our hybrid CodeBERT + structural features model performed much better, as shown in the last row of Table 1, where it achieved an accuracy of 0.9747 and a macro F1-score of 0.9746. Precision and recall values were also consistently high, indicating that combining semantic embeddings with language-agnostic structural features was highly effective on the validation set. In contrast, on a sampled subset of 1,000 test examples (unseen data) performance was even lower than the leaderboard results: a macro F1-score of 0.3599 and overall accuracy of 0.3791.

This discrepancy between validation performance and test performance can like be attributed to two factors. Firstly, the training and validation distributions were heavily skewed toward Python, which accounted for 91.5% of the data (457,306 out of 500,000 samples). However, the test data had a markedly less skewed distribution across languages, as shown in Figure 1. While CodeBERT is pre-trained on a diverse multi-lingual corpus, the fine-tuning phase likely overfit the model's parameters and structural feature weights to Pythonic syntax, to the detriment of its performance on other languages such as PHP, C, and Java. Secondly, the official test set lacked explicit language metadata, unlike the training and validation sets. While our hybrid neural architecture was designed to be language-agnostic, the Tree-sitter preprocessor requires a language label to parse the code. Thus, the system had to rely on heuristically inferred language labels.

Thus, the lack of language-specific guidance during inference, combined with our model's over-

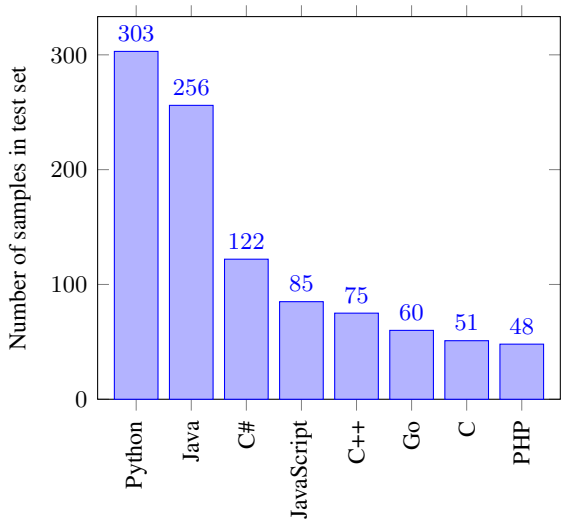

Figure 1: Distribution of programming languages in the SemEval-2026 Task 13 Subtask A test set.

Language	Original (%)	Resampled (%)
Python	91.0	47.5
Java	∼6.0	40.1
C++	∼3.0	11.7
Total Samples	∼1,000,000+	∼100,000

Table 2: Class Distribution Before and After Undersampling

Configuration	Test F1
Baseline (original distribution)	0.35
After undersampling	0.52
Relative improvement	+48.6%

Table 3: Model Performance Before and After Resampling

exposure to Python patterns, limited its ability to generalize its "Human vs. AI" detection capabilities across the broader linguistic spectrum. Consequently, the final leaderboard score of 0.49655 reflects the challenge of maintaining high precision when both the available metadata and the underlying language distribution shifts away from the training majority. Overall, these results indicate that hybrid semantic-structural modeling is effective for balanced datasets, but the challenge of unseen code generators and diverse languages in the test set significantly impacts leaderboard performance.

7.1 Post-hoc Rebalancing Analysis

The original training data exhibited a Python-dominant distribution (91%), which caused the model to default to predicting Python for ambiguous inputs. To address this, we applied undersampling to reduce the training set to approximately 100,000 samples, matched to the test data proportions across the three seen languages: Python (47.5%), Java (40.1%), and C++ (11.7%). Table 2 shows statistics of the resulting dataset.

Retraining on this balanced subset reduced the Python dominance bias and gave Java and C++ fairer representation during learning. As a result, the model no longer defaulted to predicting Python for ambiguous inputs. However, the overall test F1 score improved only modestly, from 0.35 to approximately 0.52 as shown in Table 3.

7.2 Post-hoc Language Detection Heuristic Analysis

Python performed best with approximately 83% accuracy, attributable to the model being trained on 91% Python data, providing strong heuristic signals such as indentation patterns and `def` keyword usage. The most significant failure mode involved the five unseen languages (C#, JavaScript, Go, C, and PHP), which collectively covered 366 test samples. The model had no prior knowledge of these languages and likely defaulted to predicting Python or Java for the majority of them.

Java and C# were heavily confused with each other due to near-identical syntax. Similarly, C was almost always misclassified as C++ given the shared use of `#include` directives and pointer syntax. In summary, the heuristics only meaningfully generalised for Python; all remaining languages were either entirely unseen or syntactically too similar to distinguish without dedicated training data.

7.3 Post-hoc Qualitative Analysis

A manual inspection of a small sample of the test data was conducted to assess different failure types. Figure 2 shows an example where using only the CodeBERT semantic features yields an incorrect classification, while the hybrid model that includes syntactic features as well yields a correct classification of AI-generated. Figure 3 shows an example where our hybrid model fails on a language it did not see during training.

```
def Range(a): return min(max(a, -10), 10)
def FirstProb(n):
    return (0, 0, 0)
def Escalar(a1, a2):
    return sum(x - y for x, y in zip(a1, a2))
def main():
    if Sum(FirstProb(int(stdin.readline()))) == 0:
        print('is␣idle␣body')
    else:
        print('is␣not␣idle␣body')
```

Figure 2: Example where CodeBERT alone incorrectly predicts human-written while our Hybrid model correctly predicts AI-generated.

```
<?php
fscanf(STDIN, "%d␣%d\n", $N, $D);
$count = 0;
for ($i = 0; $i < $N; $i++) {
    fscanf(STDIN, "%f␣%f\n", $x, $y);
    $distance = sqrt(pow($x, 2) + pow($y, 2));
    if ($distance <= $D) {
        $count++;
    }
}
echo $count;
```

Figure 3: Example where our Hybrid model incorrectly predicts AI-generated on this human-written PHP snippet, a language unseen during training.

8 Conclusions

In this work, we explored the task of detecting whether source code was written by a human or generated by an AI model in SemEval-2026 Task 13 Subtask A. Our system combines semantic embeddings from CodeBERT with structural features extracted using Tree-sitter, including nesting depth, logic density, and complexity per line.

On the official validation split, our hybrid model achieved high performance, with an accuracy of 0.9747 and a macro F1-score of 0.9746, indicating that the combination of semantic and structural features is highly effective for datasets dominated by seen languages.

However, evaluation on unseen test samples shows a substantial drop in performance (macro F1-score of 0.3599, accuracy 0.3791), and the official leaderboard score of 0.49655 (ranking 47th out of 81) reflects the difficulty of detecting AI-generated code when test metadata, such as the programming language, is unavailable. This gap highlights the challenge posed by imbalanced training distributions where Python comprised over 91% of the training data.

Future work could explore dataset rebalancing, improved language detection heuristics to handle datasets without explicit language labels, and ensemble strategies to improve generalization and robustness in detecting AI-generated code across all programming languages.

Acknowledgments

We acknowledge the SemEval-2026 Task 13 organizers for providing the dataset and evaluation framework.

References

Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Will Lillis, ObserverOfTime, dundargoc, Phil Turnbull, Christian Clason, Timothy Clem, Douglas Creager, Andrew Helwer, Antonin Delpeuch, Daumantas Kavolis, Riley Bruins, Michael Davis, Ika, bfredl, Tuan-Anh Nguyen, Amin Ya, and 10 others. 2026. tree-sitter/tree-sitter: v0.26.6.

Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 255–270, USA. USENIX Association.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger,

Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

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.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander Rush. 2019. GLTR: Statistical detection and visualization of generated text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 111–116, Florence, Italy. Association for Computational Linguistics.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. 2023. Detectgpt: zero-shot machine-generated text detection using probability curvature. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026. 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.