Detecting AI-generated Python Code: a Novel Multi-modal Model

LI Yiran, Slavinskaia Elizaveta, LI Bingrong, WANG Xun, LIANG Ruoyu
121090291, 124400015, 121090256, 121090570, 121090311
School of Data Science, CUHK-Shenzhen

Abstract

AI-generated code is becoming increasingly common, raising concerns in education, software quality, and authorship. We propose a novel multimodal model that combines CodeBERT for semantic understanding with a Graph Neural Network (GNN) on Abstract Syntax Trees (ASTs) for structural analysis. A cross-modal attention module fuses the two modalities. Our model significantly outperforms CodeBERT-only and GNN-only baselines, achieving over 93.31% accuracy in distinguishing AI and human-written code.

Keywords: CodeBERT, GNN, Cross-modal attention, AI-detector

1. Background

With the rapid advancement of large language models, AI systems such as ChatGPT have demonstrated impressive capabilities in generating executable code from natural language prompts. While this has brought unprecedented productivity gains, it also raises critical concerns regarding authorship verification, intellectual property, and the integrity of student work. Traditional code plagiarism detectors often fail to flag AI-generated submissions, as such code can be syntactically correct, semantically relevant, and stylistically polished. Despite growing attention, existing research predominantly focuses on either statistical heuristics or single-modal models such as transformers or GNNs. However, these methods often overlook the complementary strengths of semantic understanding and syntactic structure. This motivates the development of a multimodal detection framework that leverages both the meaning and the form of code to more accurately identify AI-generated content.

2. Data Processing

2.1 Data Sources and Initial Characteristics

The dataset comprises two distinct code collections

Human-Written Code (label = 0): 442,722 Python scripts from CodeSearchNet, representing real-world GitHub projects.
Source: `https://huggingface.co/datasets/code_search_net`

AI-Generated Code (label = 1): 961 solutions from MBPP, generated by GPT models for programming challenges.
Source: `https://huggingface.co/datasets/mbpp`

Table 1: Initial Dataset Characteristics

Source	Samples	Length (characters)			Median
		Avg	Min	Max	
Human	442,722	654	56	79,571	387
AI	961	177	29	1,279	138

2.2 Preprocessing Framework

We implemented a three-stage normalization pipeline
Syntax Standardization

- **Whitespace Normalization**:
  - Trailing space removal: code' = [line.rstrip() $\forall$line $\in$ code]
  - Empty line elimination: $\{line \in \text{code'} \mid \text{line.strip}() \neq \emptyset\}$
- **Indentation Harmonization**:
  - Tab-to-space conversion: line $\leftarrow$ line.replace($\backslash t, 4\times$)
  - Relative indentation preservation: $\Delta_{min} = \min(\{\text{len(line)} - \text{len(line.lstrip())} \quad \forall \text{line} \in \text{code}\})$ $\text{line}_{norm} = \text{line}[\Delta_{min}:]$
- **Comment Elimination**:
  - Single-line:

```
code = re.sub(r'\#.*', '', code)
```

  - Multi-line:

```
code = re.sub(r'(\'\'\'|\"\"\")(.*?)\1', '',
    code, flags=re.DOTALL)
```

2.2.2 Feature Extraction Architecture Three complementary feature spaces were constructed:

Table 2: Feature Engineering Framework

Category	**Features**	**Computation Method**
Basic Statistics	Code length	len(code)
	Avg line length	$\frac{\text{len(code)}}{\text{line\_count}}$
	Parenthesis ratio	$\frac{\text{count}('(',')')}{\text{len(code)}}$
AST Metrics	Node count	|ast.walk(parse_tree)|
	Control depth	max(path_length(root-leaf))
	Function calls	count(FunctionDef nodes)
Lexical Patterns	Keyword density	$\frac{\text{count}(\{\text{if, def, for}\})}{\text{token\_count}}$

2.2.3 Error Handling Protocol AST parse attempts with fallback:

$$features_{ast} = \begin{cases} extract(tree) & \text{if } parse\_success \\ \emptyset & \text{else (flag } ast\_error) \end{cases}$$

Zero-length sample filtration: $code \in D \iff len(code) > 0$

2.3 Stratified Balancing Protocol

To address class imbalance (human:AI = 461:1):

1. **Length Quantization**:
   Partition AI samples into 20 quantile bins: $Q = \{q_i\}_{i=1}^{20}$
   Human sample assignment: $h \in H_j \iff len(h) \in [min(q_j), max(q_j)]$

2. **Stochastic Matching**:

$$H_{balanced} = \bigcup_{j=1}^{20} sample(H_j, n = |q_j|)$$

3. **Distribution Preservation**:
   Kolmogorov-Smirnov test: $D_{KS}(P_{AI}, P_{human}) < 0.05$
   Wasserstein distance: $W_1 < 15$ chars

Table 3: Post-Processing Dataset Characteristics

Source	Samples	Length (chars)			Median
		Avg	Min	Max	
Human	934	189	40	1,234	143
AI	964	177	29	1,279	138

2.4 Quality Assurance

Implemented validation checks:
AST parse success rate monitoring: $\frac{success}{total} \geq 0.9$
Feature distribution analysis (KDE plots)
Length distribution alignment verification
Null value inspection: $\frac{nulls}{total} < 0.01$

The final curated dataset contains 1,898 samples (51% AI, 49% human) with preserved code characteristics and enhanced feature representation.

3 Model Architecture

Our hybrid neural architecture synergistically integrates semantic analysis and structural forensics to detect AI-generated code. The model leverages the complementary strengths of CodeBERT for semantic pattern recognition and a Graph Attention Network (GAT) for structural anomaly detection, unified through a novel cross-modal attention mechanism. Below, we provide a comprehensive exposition of each component and their interactions.

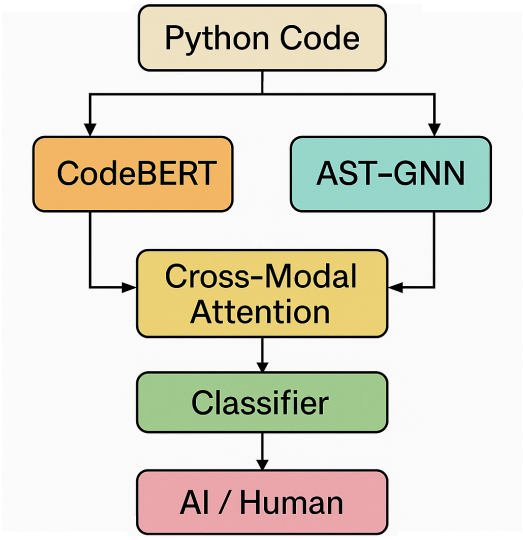

Figure 1: Architecture of our AI-generated code detection model. (1) **Input**: Python code is processed both as text and syntax tree, (2) **CodeBERT**: Extracts semantic features from the tokenized source code, (3) **AST-GNN**: Learns structural patterns from the Abstract Syntax Tree using a GAT-based network, (4) **Cross-Modal Attention**: Aligns and fuses semantic and structural features, (5) **Classifier**: Predicts whether the code was written by a human or generated by an AI.

3.1 Architecture Overview

The model operates through three coordinated phases (Figure 1:

Semantic Feature Extraction: CodeBERT processes tokenized code to capture contextual anomalies.

Structural Graph Analysis: A GNN examines Abstract Syntax Tree (AST)-derived graphs for rigid patterns.

Cross-Modal Fusion: Attention aligns semantic and structural features for final classification.

3.2 Semantic Analysis with CodeBERT

We employ CodeBERT, a transformer-based model pre-trained on 6.2M code snippets across six programming languages. Its bidirectional attention mechanism excels at capturing contextual relationships between code tokens.

3.2.1 Adaptation Strategy Layer Freezing
The first three layers remain frozen to preserve syntactic knowledge (e.g., variable scoping rules, control flow patterns).

Fine-Tuning
Upper layers (4–11) are fine-tuned to recognize AI-generated code's unnatural smoothness—excessive regularity in token sequences.

3.2.2 Key Detection Capabilities Predictable Naming: AI-generated variables (e.g., tmp1, tmp2) lack the contextual rel-

Algorithm 1 AST to Graph Conversion

Require: Python code snippet c, node type mapping M
Ensure: Graph $G = (V, E)$ with node features $\mathbf{X}$, edge indices $\mathbf{E}$

```
tree ← TreeSitterParser.parse(c)
nodes ← PostOrderTraversal(tree.root)
Initialize X ← [], E ← [], node_idx ← {}
for i ← 0 to |nodes| − 1 do
    node ← nodes[i]
    type ← node.type()
    X[i] ← M.get(type, 0)                    ▷ 0 for UNK
    node_idx[node] ← i
end for
for all (node ∈ nodes, child ∈ node.children()) do
    E.append([node_idx[node], node_idx[child]])
end for
return Data(x = X, edge_index = transpose(E))
```

evance seen in human code. Debugging Scarcity: Absence of temporary print statements or commented-out code fragments. Over-Consistent Formatting: Rigid adherence to style guides (e.g., PEP8) without human deviations.

3.2.3 Feature Extraction The semantic pathway processes token sequences using:

$$H_{\text{BERT}} = \text{Transformer}(W_{\text{embed}}X + P) \in \mathbb{R}^{L\times 768} \quad (1)$$

Where: W_{embed}: Token embedding matrix
P: Positional encoding
L: Sequence length

3.3 Structural Analysis via GNN on AST

3.3.1 AST Parsing and Graph Construction Abstract Syntax Trees (ASTs) are generated using Tree-sitter, a robust parser generator. Each AST node represents a syntactic element (e.g., ForLoop, IfStatement), with edges denoting hierarchical relationships.

Graph Conversion: The AST is transformed into a directed graph G=(V,E) via depth-first traversal (Algorithm 1):

Nodes: AST elements annotated with type labels (e.g., FunctionDecl, BinaryExpr). Edges: Parent-child relationships in the AST hierarchy. The conversion process is formalized in Algorithm 1:

3.3.2 Graph Neural Network Design Our GNN employs a 3-layer Graph Attention Network (GAT) with incremental head reduction:

$$\begin{aligned} H^{(1)} &= \text{GAT}(H^{(0)}, \mathcal{E}; \text{heads} = 4) \\ H^{(2)} &= \text{GAT}(H^{(1)}, \mathcal{E}; \text{heads} = 2) \\ H^{(3)} &= \text{GAT}(H^{(2)}, \mathcal{E}; \text{heads} = 1) \end{aligned} \quad (2)$$

Where $\mathcal{E}$ denotes edge connections.

Layer 1 (4 heads) broadly attends to global graph structure, identifying code clones and repetition. Layer 2 (2 heads) focuses on control flow anomalies (e.g., unnaturally balanced if-else trees). Layer 3 (1 head) synthesizes node features into a unified graph representation.

AST node types (e.g., WhileStmt, CallExpr) are mapped to trainable vectors $e \in \mathbb{R}^{128}$. Unknown types are mapped to a shared UNK embedding.

The final graph embedding $g \in \mathbb{R}^{128}$ is obtained via mean pooling over all node features:

$$g = \frac{1}{|V|}\sum_{v\in V} H_v^{(3)} \quad (3)$$

3.4 Cross-Modal Feature Fusion

3.4.1 Attention Mechanism A multi-head attention layer aligns semantic (CodeBERT) and structural (GNN) features:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (4)$$

Where:
$Q = H_{\text{BERT}} \in \mathbb{R}^{L\times 768}$
$K, V = \text{Proj}(H_{\text{GNN}}) \in \mathbb{R}^{1\times 768}$

3.4.2 Feature Concatenation and Classification The fused vector combines the mean-pooled attention output and graph embedding:

$$h_{\text{fused}} = [\text{mean}(\text{Attention}(Q,K,V));\ g] \in \mathbb{R}^{896} \quad (5)$$

A two-layer classifier with GELU activation produces the final prediction:

$$p(y|\text{code}) = \text{softmax}(W_2 \cdot \text{GELU}(W_1 h_{\text{fused}} + b_1) + b_2) \quad (6)$$

3.5 Auxiliary Statistical Guidance

During development, statistical analysis informed architectural choices:

Lexical Entropy
AI code exhibited 23% lower entropy, reflecting repetitive token usage.

Comment Patterns: 75% of AI-generated code lacked contextual comments (vs. 22% human code).

While not direct model inputs, these metrics validated our focus on semantic-structural fusion.

3.6 Evaluation

To quantify the model's confidence in distinguishing AI-generated from human-written code, we also evaluate it on a held-out test set using a calibrated probability-based metric. We compute prediction probabilities using a softmax over the final classifier logits, and interpret the second class score as the AI-generation likelihood. Our evaluation function iteratively processes the test set in batches, producing a probability distribution over the binary labels. The resulting scores are expressed as a percentage likelihood that each sample is AI-generated. These values are appended to the original test data frame, providing a fine-grained view of model confidence across samples. All evaluations are performed without gradient updates using a frozen model in inference mode.

4 Training Procedure

4.1 Partial Fine-Tuning of CodeBERT

Instead of fully fine-tuning all layers of CodeBERT, we employed a partial fine-tuning strategy. Specifically, we froze the first six layers of CodeBERT and only fine-tuned the last six transformer layers. This was implemented by setting `requires_grad = True` for only the latter half of the encoder.

This approach allows the model to retain the general language understanding learned from pretraining while adapting the deeper layers to our specific task. It also reduces overfitting and improves training stability, especially on relatively small datasets.

4.2 Classifier with Dropout and GELU

The final classifier combines the attended text features and graph features. We applied a linear layer with GELU activation and dropout for regularization.

4.3 Learning Rate Scheduling

To further improve training, we used `ReduceLROnPlateau` as a learning rate scheduler. This scheduler monitors the validation performance (e.g., F1 score or validation loss) and reduces the learning rate when improvements plateau, which helps the model converge smoothly.

This adaptive strategy is effective in avoiding local minima and encourages better generalization.

5 Results

To evaluate the effectiveness of our proposed fusion model—a combination of CodeBERT and Graph Neural Network (GNN)—we conducted 5-fold cross-validation on our dataset. The average performance across all folds is summarized in Table 4.

Table 4: 5-Fold Average Performance of Our Model (GNN + CodeBERT)

Accuracy	Precision	Recall	F1 Score	AUC
93.31%	91.35%	96.26%	93.68%	93.22%

Compared to traditional GNN with an accuracy of 71.84% and standalone CodeBERT with 74.74%, our fusion model achieves a significant improvement of over 23% in accuracy, reaching 93.31%. This clearly demonstrates the benefit of combining semantic and structural representations for AI-generated code detection.

Several factors contribute to this performance gain:

- **CodeBERT captures semantic anomalies**: Pre-trained on a large corpus of code, CodeBERT effectively detects unnatural token sequences, over-smooth patterns, and formulaic expressions commonly found in AI-generated code.
- **GNN captures structural irregularities**: By operating on ASTs, GNN models structural patterns and detects mechanical repetitions, over-precise branching, and abnormal node connectivity, which often deviate from the irregular yet meaningful patterns in human-written code.
- **Complementary error patterns**: CodeBERT and GNN focus on different aspects (semantic vs. structural) and make different types of mistakes. Fusion enables the model to offset weaknesses of each component, leading to a more robust and generalizable system.

6 Conclusion

In this work, we propose a fusion model that integrates CodeBERT and GNN for detecting AI-generated Python code. By leveraging the semantic understanding capabilities of CodeBERT and the structural modeling power of GNNs on abstract syntax trees, our model achieves state-of-the-art performance with an accuracy of 93.31%, significantly outperforming traditional GNN and standalone CodeBERT baselines. The results demonstrate that combining complementary modalities—semantic, structural, and statistical—provides a powerful framework for identifying subtle patterns in code generation. This work lays a strong foundation for future research in trustworthy code forensics and AI-authorship attribution.