Team Poznan at SemEval-2026 Task 13: Detecting Machine-Generated Code with Multiple Programming Languages, Generators, and Application Scenarios

Dawid Siera*, **Anatol Kaczmarek***, **Wiktor Kamzela***, **Adam Dobosz***, **Jakub Dutkiewicz**
Politechnika Poznańska, Poznań, Poland
{dawid.siera,anatol.kaczmarek,wiktor.kamzela,adam.dobosz}
@student.put.poznan.pl
jakub.dutkiewicz@put.poznan.pl

Abstract

Detecting machine-generated code is crucial for maintaining software security, quality and academic integrity. Traditional approaches often rely on stylistic or statistical features, which are increasingly circumvented by advanced code generation models. This paper introduces a novel approach leveraging Graph Neural Networks (GNNs) to capture the structural characteristics of code, specifically modeling source snippets as Abstract Syntax Trees (ASTs). To enhance semantic comprehension, we integrate pre-trained CodeBERT embeddings into the architecture, creating a hybrid model that incorporates structural and semantic information. We evaluate our approach on SemEval-2026 Task 13, covering binary detection, multi-class authorship attribution, and hybrid code classification. Experimental results demonstrate that our GNN-based structural analysis outperforms standalone stylistic and transformer-based baselines across all subtasks, particularly in multi-class and adversarial settings. This work highlights the potential of GNNs for a more structural understanding of code authorship.

1 Introduction

Detecting machine-generated code is becoming crucial for maintaining the security, reliability, and overall quality of modern software systems. As software development increasingly integrates large language models and other automated tools, the volume of code produced by these systems is rapidly growing. While offering significant productivity gains, this trend introduces new challenges related to code integrity and appropriate attribution. The ability of Large Language Models (LLMs) like GPT4 (OpenAI et al., 2024) or Gemini 2.5 (Comanici et al., 2025) to solve problems supported by the agentic systems (Wang et al., 2024) allows students to increasingly leverage these tools to assist with programming assignments and projects, raising concerns about academic integrity and the development of fundamental coding skills. While LLMs can be valuable learning aids when used responsibly, undetected use of generated code undermines the assessment of a student's understanding and ability to independently solve problems (Pudasaini et al., 2024). Accurately identifying code generated by these models is therefore essential for ensuring fair evaluation and fostering genuine learning.

Existing approaches to code authorship detection can be broadly categorized into stylistic analysis and embedding-based methods. Stylistic analysis focuses on identifying differences in coding conventions, variable naming, and code formatting. While effective against simpler generators, these methods are easily bypassed by models designed to mimic human style. LLM and embedding-based methods, such as those utilizing CodeBERT (Feng et al., 2020) or similar pre-trained language models, aim to capture semantic representations of code snippets. However, these approaches often struggle to fully capture the underlying structural complexities of code. Furthermore, they frequently require large amounts of training data, particularly for specialized programming languages or domains.

This paper introduces a novel approach to detecting machine-generated code leveraging the power of Graph Neural Networks (GNNs). Rather than focusing on surface-level features or purely semantic embeddings, we propose representing code as an abstract syntax tree (AST). An AST captures the intricate relationships between code elements (variables, functions, classes), providing a richer and more structural representation of the code's logic. By modeling code in this manner, our approach aims to capture subtle yet crucial differences in the underlying program structure that distinguish between human-written and machine-generated code. Additionally, we propose a hybrid model, which combines syntactic information represented as AST

*These authors contributed equally as first authors.

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2275–2280

and semantic output produced by CodeBERT. We hypothesize that this hybrid approach will yield improved results, as it can effectively leverage heterogeneous information from both representations.

We demonstrate that our Hybrid GNN-CodeBERT classifier significantly outperforms both traditional stylistic methods and state-of-the-art embedding-based techniques on a range of benchmark datasets. Our experiments show improved accuracy in identifying code produced by various LLMs and in various task settings:

- Binary Machine-Generated Code Detection – Given a code snippet, predict whether it is (i) fully human-written or (ii) fully machine-generated.
- Multi-Class Authorship Detection – Given a code snippet, predict its author: (i) human, or (ii-xi) one of 10 LLM families: DeepSeek-AI, Qwen, 01-ai, BigCode, Gemma, Phi, Meta-LLaMA, IBM-Granite, Mistral, or OpenAI.
- Hybrid Code Detection – Classify each code snippet as one of: (i) human-written, (ii) machine-generated, (iii) hybrid – partially written or completed by an LLM, or (iv) adversarial – generated via adversarial prompts or RLHF to mimic humans.

This work highlights the potential of GNNs for achieving a deeper, more semantic understanding of code authorship and contributes to the growing body of research on trustworthy and secure software development, as well as tools for promoting academic integrity. Implementation of the proposed methods is available via a public repository[1].

2 Background

The SemEval-2026 Task 13 dataset is designed to benchmark the ability of systems to detect machine-generated code, focusing on generalization across diverse programming languages, code generation models, and application contexts (Orel et al., 2026). The dataset comprises three interconnected subtasks, each with a unique goal and evaluation setup. Data is released via Kaggle, HuggingFace Datasets, and as .parquet files within the GitHub repository (SemEval, 2026). Each subtask's dataset includes the code snippet, a label (represented as an integer ID), and associated metadata detailing the programming language and the generator used (where applicable).

Subtask A: Binary Machine-Generated Code Detection. This subtask challenges participants to classify code snippets as either fully human-written or fully machine-generated. The training data is limited to C++, Python, and Java code snippets sourced from the algorithmic domain (e.g., problems similar to those found on LeetCode). The test set contains: (i) seen languages & seen domains, (ii) unseen languages & seen domains (using Go, PHP, C#, C, and JavaScript), (iii) seen languages & unseen domains (research and production code), and (iv) unseen languages & unseen domains. The training set contains 500,000 samples (238,000 human-written and 262,000 machine-generated), with a separate 100,000-sample validation set.

Subtask B: Multi-Class Authorship Detection. This subtask expands on binary detection by requiring participants to identify the author of a given code snippet. Possible authors include Human and one of ten specific LLM families: DeepSeek-AI, Qwen, 01-ai, BigCode, Gemma, Phi, Meta-LLaMA, IBM-Granite, Mistral, and OpenAI. The test set focuses on the system's ability to generalize to seen and unseen authors (within known LLM families). The training set consists of 500,000 samples, with varying distributions across authors (ranging from 2,000 samples for BigCode and Gemma to 10,000 for OpenAI and 442,000 for human-written code). A 100,000-sample validation set is provided.

Subtask C: Hybrid Code Detection. This most complex subtask requires classifying code snippets into one of four categories: Human-written, Machine-generated, Hybrid (partially written or completed by an LLM), and Adversarial (generated with prompts or RLHF designed to mimic human writing). The training set comprises 900,000 samples (485,000 human-written, 210,000 machine-generated, 85,000 hybrid, and 118,000 adversarial) and a 200,000-sample validation set.

2.1 Related Work

Detecting code generated by Large Language Models (LLMs) is a rapidly evolving field, spurred by the increasing accessibility and sophistication of

[1] https://github.com/Dawid64/Semeval-machine-generated-code-detection

these tools. Existing research primarily focuses on leveraging stylistic features, structural analysis, or LLM-based code embeddings to distinguish between human-written and machine-generated code. This section surveys relevant work, highlighting the strengths and limitations of current approaches and contextualizing our contribution.

Stylometric and Structural Indicators Initial detection strategies frequently employed stylometric analysis, scrutinizing metrics such as variable naming schemes, indentation styles, and code complexity (Ramos, 2003). While effective against early generation models, these methods are vulnerable to evasion by LLMs fine-tuned to replicate human coding conventions. More recent investigations, such as (Park et al., 2025), suggest that structural attributes – including function length, nesting depth, and indentation patterns – remain statistically significant even when superficial style is mimicked. These findings indicate that while explicit stylistic features can be obscured, deeper syntactic structures captured by Abstract Syntax Trees (ASTs) may offer more robust signals for detection.

Learning-Based Detection Strategies Contemporary methods increasingly rely on pre-trained models to learn distinguishing representations. Zero-shot detection techniques operate without task-specific fine-tuning, often analyzing token-level statistics such as perplexity or entropy to flag AI output (Gehrmann et al., 2019; Lavergne et al., 2008). (Ye et al., 2024; Xu and Sheng, 2025) take a different approach, comparing original code to versions rewritten by an LLM. (Shi et al., 2024) suggests that LLMs are more capable of detecting code generated by themselves than by other models. Embedding-based methods utilize models like T5 (Raffel et al., 2023; Suh et al., 2024) to project code into high-dimensional semantic spaces, which are subsequently classified by simple architectures like Multi-Layer Perceptrons. DetectLLM (Su et al., 2023), used T5 model with a normalized perturbed log-rank (NPR) as zero-shot signals. Fine-tuning approaches adapt transformer models directly to the detection task. For example, GPTSniffer (Nguyen et al., 2023) fine-tunes CodeBERT for binary classification. Other works employ contrastive learning, training models like UniXcoder (Guo et al., 2022a) on positive and negative function pairs to maximize class separation (Xu et al., 2024).

Graph-Based Structural Analysis While transformer models capture semantics, they often overlook explicit structural relationships. (Guo et al., 2022b) proposed using ASTs with Graph Neural Networks (GNNs) for human code authorship attribution. Although their methodology aligns closely with our use of structural data, it is constrained to Python code and relies on datasets containing only human-written samples with limited author diversity (at most 30 samples per author). In contrast, our work extends graph-based structural analysis to the domain of AI-generated code detection, utilizing mixed authorship datasets and supporting multiple programming languages. Moreover, unlike their purely graph-based approach, we introduce a hybrid architecture that integrates GNNs with semantic embeddings (CodeBERT), thereby leveraging both structural and semantic information for improved robustness.

3 Experimental Setup

This section details the models employed for machine-generated code detection, including baseline approaches and our proposed architectures.

3.1 Baselines

We establish two baseline models to serve as reference points for evaluating our proposed architectures. These baselines represent the primary methodological categories in existing detection approaches.

CodeBERT Baseline We train a baseline using CodeBERT, a pre-trained transformer model for code understanding. We fine-tune CodeBERT for binary classification by attaching a classification head to the embedding of the first token of the input code sequence, using code provided by the SemEval Task 13 organizers.

Morphology AST-Based Baseline As a second baseline, we leverage Abstract Syntax Trees (ASTs) generated using Tree-sitter. We parse code into ASTs and extract four node types: variable, function, class, and other. For each node type, we quantify the occurrence of eight naming conventions – screaming snake case, snake case, kebab case, camel case, pascal case, upper case, lower case, and other – resulting in a total of 32 features per AST. We also include the total number of identifiers and counts for each naming convention, expanding the feature space to 41 dimensions.

These features are then used as input to a simple Multi-Layer Perceptron (MLP) classifier.

3.2 Our Models

We propose three architectures that combine complementary representations of code. Our models integrate structural graph-based approaches with semantic transformer-based methods to leverage both syntactic and semantic information for improved detection performance.

Graph Neural Network (GNN) Our first proposed model utilizes Graph Neural Networks to capture structural information from the AST. We preprocess the code using Tree-sitter and extract five node features: type_id, depth, num_children, is_named, and length. The type identifier feature (type_id) is embedded into a 32-dimensional vector and concatenated with the other node features to form the input to our GNN.

The GNN comprises two Residual Gated Graph Convolution layers each followed by a ReLU activation function, with an intermediate Dropout layer to mitigate overfitting. To aggregate information from across the entire AST, we apply max pooling to the node embeddings from the final GNN layer and pass the aggregated embedding to a classification MLP head.

Hybrid GNN-CodeBERT Model To combine the strengths of both structural and semantic information, we developed a hybrid model. This model integrates the GNN described above with the CodeBERT baseline. We concatenate the max-pooled GNN embedding from the final layer with the embedding of the first token from the final layer of CodeBERT. The resulting combined embedding is then fed into a classification MLP head.

Morphology-Augmented Hybrid Model Finally, we further enhance the hybrid model by incorporating morphological information about naming conventions. We add the 41-dimensional embedding generated by the Morphology AST-based baseline (described in the Section 3.1), encoded with an MLP, to the final embedding constructed from the GNN and CodeBERT models. This augmented embedding is then used for classification via a final MLP head. This approach aims to capture subtle stylistic cues that may distinguish between human- and machine-generated code.

4 Results

We evaluated all models on the validation set using both accuracy and macro-averaged F1-score (F1-macro) as the primary performance metrics. We focus on F1-macro as the primary metric due to the imbalanced nature of the dataset across the subtasks. Results are presented in Table 1, detailing the performance of each model across the three subtasks of SemEval-2026 Task 13.

Baseline Performance The AST-based feature baseline demonstrated surprisingly strong performance, particularly on Subtask A (Binary Machine-Generated Code Detection) and Subtask C (Hybrid Code Detection). These results significantly exceeded random performance, suggesting that LLM-generated code exhibits consistent naming conventions distinguishable from those employed by human developers.

CodeBERT achieved an F1-macro of 0.98 on Subtask A and yielded solid results on Subtask C. However, it struggled to generalize effectively on Subtask B (Multi-Class Authorship Detection), where the performance gap between classes was more pronounced.

Proposed Models Our initial Graph Neural Network (GNN) model achieved slightly better results than CodeBERT on Subtasks A and C, while exhibiting slightly lower performance on Subtask B. Notably, the GNN model contains approximately ten times fewer parameters than CodeBERT and was trained from scratch, without leveraging pre-training on extensive code corpora. This highlights its efficiency and potential for learning from limited data.

The Hybrid GNN-CodeBERT model consistently improved performance across all three subtasks, achieving near-perfect F1-scores on Subtask A and surpassing the CodeBERT baseline by 0.07 and 0.13 F1-score on Subtasks B and C respectively. These results underscore the importance of integrating structural (AST-based) and semantic (CodeBERT) information for improved code detection accuracy.

The Morphology-Augmented Hybrid Model yielded slightly improved F1-macro on Subtask B, while maintaining equal performance on Subtasks A and C. This suggests that while morphological information can provide a marginal benefit, CodeBERT may already be capable of effectively encoding such features during its pre-training pro-

Models	Task A		Task B		Task C	
	Accuracy	Macro F1	Accuracy	Macro F1	Accuracy	Macro F1
CodeBERT	0.9861	0.9860	0.9108	0.3161	0.7965	0.6784
Morphology AST-Based	0.7933	0.7931	0.461	0.1203	0.5541	0.4755
GNN	0.9881	0.9881	0.8670	0.3109	0.7870	0.7014
Hybrid GNN	**0.9949**	**0.9949**	**0.8647**	0.3843	**0.8702**	**0.8098**
Morphology-Augmented Hybrid GNN	**0.9949**	**0.9949**	0.8605	**0.3937**	0.8693	0.8082

Table 1: Results for evaluation on validation datasets for all 3 tasks.

cess.

5 Conclusion

This study demonstrates the feasibility of leveraging both structural and semantic information for accurate detection of machine-generated code. Our results indicate that simple features derived from Abstract Syntax Trees (ASTs), such as naming conventions, can provide strong signals for distinguishing between human- and machine-authored code, particularly in binary and hybrid detection scenarios. While pre-trained language models like CodeBERT offer strong performance, especially on tasks focused on code understanding, their effectiveness can be further enhanced by integrating structural representations through Graph Neural Networks.

The Hybrid GNN-CodeBERT model consistently outperformed all other approaches, achieving near-perfect accuracy on the binary detection task and significant improvements on multi-class authorship identification and hybrid code classification. This highlights the synergistic benefits of combining the code-understanding capabilities of transformer-based models with the structural awareness of GNNs. The marginal gains observed with the addition of explicit morphological features suggest that CodeBERT already captures a substantial degree of stylistic information within its embeddings.

These findings have implications for various applications, including detection of LLM-generated projects in education and code security analysis, ultimately contributing to a broader understanding of authorship, authenticity, and potential vulnerabilities within the expanding landscape of software development and increasingly automated coding practices.

6 Limitations

A primary limitation of our approach is that it is not language-agnostic. When trained on corpora containing code written in specific programming languages, the classifier inevitably learns language-specific features and structural patterns. As a result, the model does not generalize in a zero-shot setting to unseen programming languages and would require retraining or adaptation.

This limitation could be partially mitigated by explicitly incorporating programming language information as an input feature or by training language-specific variants of the model. However, the SemEval test set exhibits substantial data drift, particularly in terms of language distribution, which makes direct generalization challenging. For this reason, we report results on the official validation set, where the data characteristics are controlled and consistent with the training setup.

While this restricts direct comparability on the hidden test set, we believe the proposed modeling strategy and analysis remain relevant beyond the context of the workshop task.

References

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 3416 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

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. *Preprint*, arXiv:2002.08155.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander M. Rush. 2019. Gltr: Statistical detection and visualization of generated text. *Preprint*, arXiv:1906.04043.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022a. Unixcoder: Unified cross-modal pre-training for code representation. *Preprint*, arXiv:2203.03850.

Dixiao Guo, Anmin Zhou, Liang Liu, Shan Liao, and Lei Zhang. 2022b. A method of source code authorship attribution based on graph neural network. In *Proceedings of 2021 Chinese Intelligent Automation Conference*, pages 645–657, Singapore. Springer Singapore.

Thomas Lavergne, Tanguy Urvoy, and François Yvon. 2008. Detecting fake content with relative entropy scoring.

Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2023. Is this snippet written by chatgpt? an empirical study with a codebert-based classifier. *Preprint*, arXiv:2307.09381.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

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.

Shinwoo Park, Hyundong Jin, Jeong-won Cha, and Yo-Sub Han. 2025. Detection of llm-paraphrased code and identification of the responsible llm using coding style features. *arXiv preprint arXiv:2502.17749*.

Shushanta Pudasaini, Luis Miralles-Pechuán, David Lillis, and Marisa Llorens Salvador. 2024. Survey on plagiarism detection in large language models: The impact of chatgpt and gemini on academic integrity. *Preprint*, arXiv:2407.13105.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the limits of transfer learning with a unified text-to-text transformer. *Preprint*, arXiv:1910.10683.

Juan Ramos. 2003. Using tf-idf to determine word relevance in document queries.

SemEval. 2026. SemEval-2026-Task13: Detecting machine-generated code with multiple programming languages, generators, and application scenarios. `https://github.com/mbzuai-nlp/SemEval-2026-Task13`. GitHub repository, accessed 2026-03-02.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. 2024. Between lines of code: Unraveling the distinct patterns of machine and human programmers. *Preprint*, arXiv:2401.06461.

Jinyan Su, Terry Zhuo, Di Wang, and Preslav Nakov. 2023. Detectllm: Leveraging log rank information for zero-shot detection of machine-generated text. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12395–12412.

Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. 2024. An empirical study on automatically detecting ai-generated source code: How far are we? *Preprint*, arXiv:2411.04299.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6).

Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. 2024. Distinguishing llm-generated from human-written code by contrastive learning. *Preprint*, arXiv:2411.04704.

Zhenyu Xu and Victor S Sheng. 2025. Codevision: Detecting llm-generated code using 2d token probability maps and vision models. *arXiv preprint arXiv:2501.03288*.

Tong Ye, Yangkai Du, Tengfei Ma, Lingfei Wu, Xuhong Zhang, Shouling Ji, and Wenhai Wang. 2024. Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting. *Preprint*, arXiv:2405.16133.