SemEval-2026-Task13-Subtask-A: Binary Machine-Generated Code Detection

Raviteja Gundu, Divya Navale, Divya Avuti

Abstract

The rapid progress of large language models (LLMs) has made it increasingly difficult to distinguish human-written code from machine-generated code. In this paper, we present our approach to SemEval-2026 Task 13 Subtask A (mbzuai-nlp, 2025), focusing on the binary classification of code authorship across seen and unseen programming languages. We evaluate a diverse set of architectures, including encoder-only (CodeBERT, GraphCodeBERT, ModernBERT), decoder-only (StarCoder), and unified transformers (UniXcoder), alongside feature-based baselines. Our experiments reveal that while large decoder-only models like StarCoder-3B achieve the highest overall performance on the original distribution, they are sensitive to training data volume. Conversely, encoder-based models benefit significantly from language-balanced training but struggle with cross-lingual generalization. Our findings suggest that while pretrained representations outperform classical features, achieving robustness under distribution shifts remains an open challenge.

1 Introduction

Recent advances in large language models (LLMs), such as GitHub Copilot and ChatGPT, have fundamentally transformed software development and computer science education. Trained on massive corpora of publicly available code, these models excel at tasks ranging from implementation to debugging, not only in widely used languages like Python and Java but also in less common ones. This ability to generalize across diverse programming languages and application contexts has lowered entry barriers for beginners and boosted productivity for experts, driving the rapid adoption of AI-based coding assistants in both academic and industrial settings.

However, the same capabilities that make LLMs powerful also introduce a critical challenge: distinguishing human-written code from machine-generated code has become increasingly difficult. As models improve at mimicking human coding styles across multiple programming languages and domains, the boundary between human and machine-generated code continues to blur. This ambiguity is not merely theoretical-it has tangible consequences for academic integrity, intellectual property protection, and software security.

This creates an urgent need for reliable detection mechanisms. While existing detection systems often achieve high accuracy on familiar programming languages and well-defined domains, such as algorithmic problem-solving, they frequently fail to generalize to unseen languages (e.g., JavaScript, Go, PHP). This lack of robustness under distribution shifts severely limits their real-world applicability.

In this work, we focus on the challenge of building a generalizable binary detector for machine-generated code. Specifically, we investigate the following research question: *How effectively can a detection model distinguish human-written from machine-generated code when evaluated on previously unseen programming languages?* Through systematic out-of-distribution evaluation, we quantify the generalization gap of existing approaches and explore strategies to improve detection robustness across linguistic variations.

This problem setting follows the definition of Subtask A of SemEval-2026 Task 13, *Detecting Machine-Generated Code* (mbzuai-nlp, 2025).

2 Related Work

The growing integration of large language models (LLMs) into software development has prompted significant research interest in distinguishing human-written from machine-generated code. Early efforts have primarily focused on binary classification of generated code in limited programming languages and domains.

Nguyen et al. (Nguyen et al., 2024) introduced *GPTSniffer*, a CodeBERT-based binary classifier designed to detect ChatGPT-generated code in Python and Java using the CodeSearchNet dataset (Husain et al., 2019). Their approach achieved strong in-domain accuracy but struggled to generalize to unseen programming languages, highlighting the limitations of dataset diversity in code authorship detection. Similarly, (Idialu et al., 2024) proposed a stylometric-based detection framework that leveraged code-level writing style features to identify GPT-4-generated code at the class granularity. While effective for stylistic differentiation, such methods depend heavily on consistent structural and lexical features, reducing their robustness across varied code domains.

Orel et al. (Orel et al., 2024) introduced *Droid*, a comprehensive resource suite for machine-generated code detection covering multiple programming languages, generation models, and problem settings. Their work contributed two components relevant to this study: (1) a data filtering pipeline that removes low-quality, boilerplate, and near-duplicate samples to increase authorship signal strength, and (2) a CatBoost-based classification baseline using gradient-boosted decision trees over lexical and structural code features. These components demonstrated competitive performance under cross-model and cross-language evaluations, reinforcing the effectiveness of feature-driven detection methods in broader authorship attribution scenarios.

3 Methodology

We formulate SemEval-2026 Task 13 Subtask A as a binary classification problem. Given a code snippet x, the detector predicts a label $y \in \{0, 1\}$ where 0 denotes human-written code and 1 denotes machine-generated code. To study robustness under cross-language we evaluate transformer-based detectors spanning different architectural families, along with feature-driven classical baselines.

3.1 Transformer Families

To cover diverse inductive biases in code modeling, we evaluate three transformer families:

Encoder-only Transformers learn bidirectional contextual representations and are commonly used for classification via a pooled embedding. We fine-tune *CodeBERT*, *GraphCodeBERT*, and *ModernBERT* by adding a lightweight classification head on top of the pooled sequence representation. The resulting model is optimized end-to-end for binary prediction.

Decoder-only Transformers are autoregressive and are pretrained for next-token prediction, often capturing strong generative priors and long-range code patterns. We evaluate *StarCoder-3B* by adapting the decoder representations for sequence classification using a task-specific classification head.

Unified Transformers combine bidirectional encoding with autoregressive decoding capabilities, which can be beneficial for learning both global and local signals. We evaluate *UniXcoder* as a unified code model and fine-tune it for binary classification with a classification head over the sequence representation.

3.2 Parameter-Efficient Fine-Tuning

For large models, we employ parameter-efficient fine-tuning to reduce memory and compute overhead. In particular, we use *PEFT* with *LoRA* adapters for CodeBERT and StarCoder-3B, updating only a small set of trainable low-rank matrices while keeping the backbone largely frozen. This allows stable fine-tuning under constrained GPU budgets while preserving most pretrained knowledge.

3.3 Language Balancing Strategy

The official training split exhibits strong language skew toward Python, which may encourage detectors to overfit to Python-specific stylistic artifacts rather than general authorship cues. To mitigate this risk, we create a balanced training set by downsampling Python to 50K total instances (stratified by label) while retaining all C++ and Java instances. A similar stratified downsampling strategy is applied to the validation split. Sampling is performed with a fixed random seed (random state $= 42$) to ensure reproducibility.

3.4 Feature-Based Baselines

To benchmark transformer detectors against classical machine-learning approaches, we implement two feature-driven baselines: *Random Forest* and *CatBoost*. Both models are trained on engineered lexical and structural features extracted from code snippets. These baselines provide a reference point for measuring the benefit of pretrained representations and help identify whether simple stylistic features can generalize across languages.

4 Experimental Setup

4.1 Data Splits

We use the official SemEval-2026 Task 13 dataset for Subtask A, restricted to human-written and machine-generated samples. The original training set contains 500K examples and the original validation set contains 100K examples, with a strong language skew toward Python. The test set is kept unchanged (1K samples) to ensure comparability across all experiments.

4.2 Training/Validation Distributions

The original training distribution is dominated by Python (457,306 samples), whereas C++ and Java together contribute fewer than 45K instances. To reduce this imbalance, we construct a balanced training split by downsampling Python to 50K total while preserving all C++ and Java examples. Similarly, we downsize validation from 100K to 10K using stratified sampling across `language` and `label`.

Language	Label	Original	Balanced
C++	Human	11,147	11,147
	Machine	12,245	12,245
Java	Human	9,225	9,225
	Machine	10,077	10,077
Python	Human	218,103	23,760
	Machine	239,203	26,240
Total		**500,000**	**92,694**

Table 1: Training set distribution before and after Python downsampling.

4.3 Tokenization and Input Formatting

All transformer models use their corresponding `AutoTokenizer`. Inputs are padded/truncated to model-dependent maximum sequence lengths. For experiments where preprocessing is enabled (notably for StarCoder variants), we additionally apply lightweight normalization intended to stabilize authorship signals (e.g., trimming overly long scaffolding and normalizing superficial literals where applicable). These steps are treated as controlled ablations rather than mandatory preprocessing.

4.4 Optimization and Training Protocol

All models are fine-tuned end-to-end (or via LoRA adapters for PEFT runs) for a fixed duration of 3 epochs. We use the AdamW optimizer with a learning rate of 2×10^{-5} and standard cross-entropy loss for binary classification.

For experiments on the original dataset, we standardize the maximum sequence length to 512 tokens and use a global batch size of 256. For experiments on the balanced dataset, we increase the context window to capture longer-range dependencies, utilizing maximum sequence lengths of 512 and 8192 depending on the model's architecture. Correspondingly, batch sizes for the balanced split are adjusted per model to accommodate these larger contexts within memory constraints.At inference, predictions are obtained from the model logits using a fixed threshold of 0.5.

4.5 Evaluation Metrics

We evaluate using **accuracy**, and **macro F1**. Macro-averaging ensures that the classes contribute equally to the final score, which is important when measuring generalization across languages.

5 Results

5.1 Overall Performance on the Original Dataset

Model	Precision	Recall	Accuracy	Macro F1
StarCoder-3B	0.5720	0.6033	**0.55**	**0.5230**
UniXcoder	**0.6058**	0.5905	0.3860	0.3849
CodeBERT	0.5317	0.5208	0.31	0.304
ModernBERT	0.5965	0.5438	0.3010	0.2876
GraphCodeBERT	0.4453	0.4615	0.2750	0.2702
CatBoost	0.5696	0.5339	0.2980	0.2859
RandomForest	0.2440	**0.9110**	0.3500	0.3480

Table 2: Overall performance of models on the original dataset.

As shown in Table 2, StarCoder-3B achieves the strongest overall performance, with an accuracy of 0.55 and a macro F1 score of 0.5230. This demonstrates the effectiveness of large decoder-only pretrained code models for machine-generated code detection under the original, Python-dominated training distribution.

Among encoder-only models, UniXcoder achieves high macro precision (0.6058) and recall (0.5905) but substantially lower accuracy (0.3860), suggesting unstable decision boundaries when generalizing across languages. CodeBERT, ModernBERT, and GraphCodeBERT exhibit weaker performance, with macro F1 scores below 0.31, highlighting the difficulty of general-purpose encoders in this cross-domain setting.

Feature-based baselines show contrasting behavior. CatBoost achieves moderate precision but low overall accuracy, while Random Forest attains very high recall (0.9110) at the expense of extremely low precision (0.2440), indicating a strong bias toward predicting machine-generated code. These results confirm that classical models relying solely on surface-level features struggle to balance predictions under distribution shift.

5.2 Overall Performance on the Balanced Dataset

Model	Accuracy	Macro F1
StarCoder-3B	0.2720	0.2699
UniXcoder	0.2820	0.2745
CodeBERT	0.2970	0.2914
ModernBERT	0.3310	0.3246
GraphCodeBERT	**0.3920**	**0.3915**
RandomForest	0.3500	0.3480

Table 3: Overall performance of models on the balanced dataset.

Table 3 reports results for models trained on the balanced dataset constructed as described in Section 4.1. In this setting, GraphCodeBERT achieves the best performance, reaching an accuracy of 0.3920 and a macro F1 score of 0.3915. This improvement suggests that structural representations become more effective when dominant language artifacts are reduced.

ModernBERT also benefits from balancing, improving to 0.3310 accuracy and 0.3246 macro F1, outperforming CodeBERT and UniXcoder. In contrast, StarCoder-3B experiences a notable drop in performance when trained on the balanced dataset, falling to 0.2720 accuracy and 0.2699 macro F1. This indicates that aggressive downsampling may remove useful training diversity for large decoder-only models that benefit from larger-scale exposure.

Overall, these results reveal a trade-off between reducing language skew and preserving sufficient training signal. Encoder-based and structure-aware models benefit more consistently from balancing, while decoder-only models achieve higher peak performance on the original dataset.

5.3 Per-Language Performance: Seen vs. Unseen Languages

Table 4 presents per-language macro F1 scores for models trained on the original dataset. The test languages are divided into seen (Java, C++, Python) and unseen (C, C#, Go, JavaScript, PHP) languages.

For seen languages, all models achieve their highest and most stable performance, particularly on Python. StarCoder-3B shows strong macro F1 on Java and C++, while encoder-based models such as GraphCodeBERT and ModernBERT remain comparatively weaker even on seen languages, indicating limited capacity to exploit training familiarity.

For unseen languages, performance varies substantially across models. StarCoder-3B demonstrates strong generalization to Go (0.6738) and competitive performance on C and C++, suggesting that large decoder-only models capture transferable authorship cues beyond training languages. Encoder-based models, particularly GraphCodeBERT and ModernBERT, show significant degradation on unseen languages such as PHP, where macro F1 drops below 0.20.

Feature-based methods such as CatBoost perform poorly across most unseen languages, reinforcing the difficulty of relying on surface-level features for cross-language authorship detection. Notably, all models struggle on PHP, highlighting it as a particularly challenging unseen language under the current training regime.

5.4 Summary of Findings

Across all experiments, large pretrained code models consistently outperform classical baselines. Decoder-only transformers achieve the highest overall performance on the original dataset, while encoder-based and structurally aware models show improved robustness when training data is balanced. The per-language analysis reveals substantial performance gaps between seen and unseen languages, underscoring the importance of multilingual pretraining and diverse training distributions for robust machine-generated code detection.

6 Additional Analysis

6.1 Exploratory Experiments with Larger Models

We additionally explored a larger decoder-only model, StarCoder-7B, to assess whether increased

Language	StarCoder-3B	UniXcoder	CatBoost	GraphCodeBERT	ModernBERT
C	0.4811	0.4491	0.1758	0.1911	0.2477
C#	0.4186	0.3306	0.2043	0.1586	0.1984
C++	0.4663	0.3599	0.2788	0.2532	0.2768
Go	**0.6738**	0.3667	0.2105	0.2734	0.2105
Java	0.4772	0.3434	0.1841	0.1957	0.1777
JavaScript	0.4118	**0.4578**	0.2944	0.1986	0.2735
PHP	0.4023	0.1652	0.1652	0.1037	0.0588
Python	0.4832	0.4429	**0.4301**	**0.4309**	**0.4561**

Table 4: Per-language Macro F1 scores for each model.

model capacity improves machine-generated code detection. While StarCoder-7B performs competitively, it does not consistently outperform the 3B variant, indicating diminishing returns under limited fine-tuning and compute budgets. Furthermore, StarCoder-7B exhibits more consistent behavior across unseen languages but does not achieve the same peak performance observed with the smaller model. These observations suggest that increased scale alone is insufficient to guarantee stronger generalization and that larger models may require more extensive optimization to fully realize their potential.

6.2 Preprocessing Strategies

We experimented with additional preprocessing strategies for StarCoder-3B, specifically applying: (1) insertion of language tags to preserve cross-lingual signals, (2) smart line-based trimming to remove boilerplate, and (3) normalization of literals to reduce superficial stylistic artifacts. Inputs were padded or truncated to a fixed sequence length. However, the preprocessed StarCoder-3B variant did not outperform the best baseline configuration.

Model	Precision	Recall	Accuracy	Macro F1
StarCoder-3B	**0.5720**	**0.6033**	**0.55**	**0.5230**
StarCoder-3B (preprocessed)	0.5411	0.5577	0.4940	0.4742
StarCoder-7B	0.5179	0.5258	0.5190	0.4784

Table 5: Exploratory analysis of model scale and preprocessing effects for StarCoder.

6.3 Limitations and Error Patterns

Across all architectures, PHP remains the most challenging unseen language, with consistently low macro F1 scores. This suggests that language-specific syntax, library usage, or domain differences introduce patterns not adequately captured by current detectors. Additionally, models frequently confuse human-written boilerplate with machine-generated code, indicating that stylistic similarity remains a key source of error.

7 Conclusion

In this work, we investigated the effectiveness of transformer-based models and classical baselines for detecting machine-generated code. Through systematic evaluation on SemEval-2026 Task 13 Subtask A, we demonstrated that architecture choice significantly influences robustness. Our results indicate that large decoder-only models, specifically StarCoder-3B, provide the strongest generalization capabilities on the original distribution, successfully transferring knowledge to some unseen languages like Go. However, we found that encoder-based models such as GraphCodeBERT and ModernBERT offer better stability when trained on balanced datasets, emphasizing the trade-off between model scale and data distribution strategies.

Despite these successes, a significant generalization gap remains. All evaluated models exhibited severe performance degradation on specific unseen languages, most notably PHP, suggesting that current detectors rely heavily on language-specific syntactic patterns rather than universal traits of machine-generation. Future work will focus on narrowing this gap by exploring cross-lingual contrastive alignment, incorporating more granular structural features, and extending detection capabilities to hybrid human-machine collaborative scenarios.

8 Github Link

The code and experimental results for this paper are available at: https://github.com/ravitejagundu11/Polyglot-Code.

References

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Precious Idialu, Xiaolong Wang, and Md Tarek Rahman. 2024. Stylometric and structural analysis for gpt-4 code authorship detection. In *Proceedings of the 2024 AAAI Conference on Artificial Intelligence*. AAAI Press.

mbzuai-nlp. 2025. Semeval-2026 task 13: Detecting machine-generated code. https://github.com/mbzuai-nlp/SemEval-2026-Task13. Accessed October 2025.

Phuong T. Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2024. Gptsniffer: A codebert-based classifier to detect source code written by chatgpt. *Journal of Systems and Software*, 214:112059.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. 2024. Droid: A resource suite for AI-generated code detection. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.