Osint at SemEval-2026 Task 13: A Distribution-Aware Framework for Machine-Generated Code Detection and Multi-Source Authorship Attribution

Abhishek Anand[1], Shifali Agrahari[1], Shubham Nilesh Kannaujiya[1], Sanasam Ranbir Singh[1], Sujit Kumar[2]
[1]Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati, India
[2]Lee Kong Chian School of Medicine, Nanyang Technological University, Singapore
{a.shifali, abhishekanand}@iitg.ac.in

Abstract

The rise of code-generating LLMs such as DeepSeek, Qwen, and Meta-LLaMA has improved developer productivity but also increased risks of plagiarism, copyright misuse, and insecure machine-generated code. While AI-text detection is well studied, machine-generated source-code detection especially across multiple languages, LLM families, and OOD conditions-remains underexplored. SemEval-2026 Task 13 addresses this via two subtasks: (A) binary human–machine code detection and (B) multi-class authorship attribution across ten LLM families. For Subtask A, we fine-tune RoBERTa, CodeBERT, GraphCodeBERT, and StarCoderBase-1B, introducing a stratified sampling strategy with class-weighted loss to mitigate imbalance and OOD shifts. For Subtask B, we mitigate the extreme human-class imbalance using undersampling, inverse-frequency weights, syntactic noising, and curriculum-based dual-path training with TinyStarCoderPy and CodeBERT. Both results show that long-context modeling, distribution-aware sampling, and noise-robust training are crucial for reliable in real-world settings. Overall, long-context modeling, distribution-aligned sampling, and lightweight noise-robust training emerge as key factors for reliable machine-generated code detection and authorship attribution.

1 Introduction

The rapid advancement of Large Language Models (LLMs) has fundamentally transformed software development. Models such as DeepSeek (Bi et al., 2024), Qwen (Bai et al., 2023), and Meta-LLaMA (Touvron et al., 2023) are now capable of autonomously generating highly functional and production-ready code. While this progress significantly enhances developer productivity, it simultaneously raises critical concerns regarding copyright infringement, academic plagiarism, and the introduction of insecure or vulnerable machine-generated code.

In the existing literature, substantial efforts have been devoted to detecting AI-generated text. Prior studies have focused on several domains, including academic writing (Agrahari et al., 2025a; Peng et al., 2023), online reviews (Agrahari et al., 2025b), and news articles (Ishraquzzaman et al., 2025). These detection systems generally demonstrate strong performance in identifying machine-generated sentences and paragraphs. However, beyond these text-centric approaches, relatively few studies have investigated the detection of AI-generated source code (Xu and Sheng, 2024; Orel et al., 2025; Adham et al., 2025; Bukhari et al., 2023). Moreover, existing code detection methods primarily focus on binary classification between human-written and single-LLM-generated code, rather than addressing the more challenging task of multi-source authorship attribution across different LLMs and human developers. This limitation highlights a clear research gap and the need for more robust methods to identify and attribute machine-generated code, especially given its growing prevalence and security risks.

To address these challenges, SemEval-2026 Task 13[1] (Orel et al., 2026) introduced the shared task *'Detecting Machine-Generated Code with Multiple Programming Languages, Generators, and Application Scenarios."* In this work, we focus on two subtasks. **Subtask A: Binary Machine-Generated Code Detection.** Given a code, predict whether it is fully human-written or fully machine-generated. **Subtask B: Multi-Class Authorship Detection.** Given a code, predict its author: Human or one of ten LLM families (DeepSeek-AI, Qwen, 01-ai, BigCode, Gemma, Phi, Meta-LLaMA, IBM-Granite, Mistral, and OpenAI).

In this paper, for Subtask A we fine-tune

[1]https://github.com/SemEval-2026-Task-13

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2866–2876

four transformer-based models such as RoBERTa, CodeBERT, GraphCodeBERT, and StarCoderBase-1B to perform binary classification of machine-generated versus human-written code, incorporating long-context training, stratified sampling (Qian et al., 2009), and imbalance-aware optimization to improve robustness under severe out-of-distribution conditions. For Subtask B we fine-tune TinyStarCoderPy and CodeBERT for multi-class authorship detection, leveraging strategic undersampling, inverse-frequency class weighting, and curriculum-based syntactic noising to address the extreme class imbalance and enhance robustness across eleven author categories. Our result and error analysis reveals that length-aware sampling, long-context modeling, and noise-aware curriculum learning (Chaudhry and Sharma, 2024) are essential for cross-language robustness and stable authorship attribution.

2 Related Work

AI-Generated Text Detection: A large body of research has emerged on detecting AI-generated text, which can be broadly categorized into three methodological families: (i) *Zero-shot detection*, (ii) *Supervised training-based detection*, and (iii) *Watermarking-based detection*. *Zero-shot methods* (Gehrmann et al., 2019; Mitchell et al., 2023; Su et al., 2023; Wang et al., 2025) rely entirely on the intrinsic statistical properties or token-level logit distributions of LLM outputs. Early systems such as GLTR (Gehrmann et al., 2019) exploited likelihood-based anomaly detection, while DetectGPT (Mitchell et al., 2023) introduced curvature-based methods using model likelihood perturbations. Recent works such as DetectLLM (Su et al., 2023) and GenAI-ID (Wang et al., 2025) further leverage consistency signals and robustness metrics across multiple LLMs. *Supervised training methods* (Guo et al., 2023; Nguyen-Son et al., 2024; Agrahari et al., 2025b; Mao et al., 2024; Agrahari et al., 2025a) train classifiers or fine-tuned transformers on human–machine labeled corpora. Guo et al. (Guo et al., 2023) demonstrated the brittleness of supervised detectors under distribution shift, while (Agrahari et al., 2024), a linguistic-feature-enriched detector. More recent systems such as RAiDAr (Mao et al., 2024) and Sim-LLM (Nguyen-Son et al., 2024) employ rewriting-based consistency checks and semantic learning to improve robustness against unseen LLMs. *Watermarking methods* (zha; Fu et al., 2024; Cohen et al., 2025; Chen et al., 2025; Li et al.) embed statistical or semantic patterns into generated text during decoding, enabling post-hoc attribution.

Task	Split	Samples	Labels
Subtask A	Train	500,000	Yes
	Validation	100,000	Yes
	Test	500,000	No
Subtask B	Train	500,000	Yes
	Validation	100,000	Yes
	Test	500,000	No

Table 1: Statistics for Subtask A and Subtask B datasets.

AI-Generated Code Detection: With the rapid adoption of code-generating LLMs, an emerging line of research focuses on distinguishing human-written code from machine-generated code (Bukhari et al., 2023) analyzed stylistic and structural differences in Python code, demonstrating that LLM outputs exhibit consistent formatting and syntactic regularities. CGCode Detector (Xu and Sheng, 2024) a perplexity-based approach that leverages LLM token distributions to flag AI-generated assignments. More recent frameworks have expanded the problem to multilingual and domain-specific scenarios. DROID (Orel et al., 2025), introduced a large-scale resource suite targeting AI-generated Android applications, while CODEDETECTOR (Adham et al., 2025), a zero-shot detection framework using statistical and embedding-based signals. The SemEval 2026 shared task 13 (Orel et al., 2026) further emphasizes the growing importance of evaluating detectors across diverse programming languages, generation models, and real-world application settings.

3 Dataset

Subtask A Dataset: The dataset provided for Subtask A consists of code snippets labeled $y \in \{0, 1\}$, where Class 0 denotes human-written code and Class 1 corresponds to machine-generated code. The dataset spans three programming languages: Python, C++, and Java. Subtask A is challenging due to extreme code-length variation (median 464, max 475k characters) and outputs from 32 code generation models, creating high stylistic diversity. Dataset statistics appear in Table 2 and App. Fig. 3.

Subtask B Dataset: The dataset Subtask B consists of Python code snippets labeled $y \in \{0, 1, \ldots, 10\}$. Class 0 represents Human authorship, while Classes 1 through 10 represent specific AI architectures (DeepSeek-AI, Qwen, 01-ai, BigCode, Gemma, Phi, Meta-LLaMA, IBM-Granite, Mistral, and OpenAI). The primary challenge in

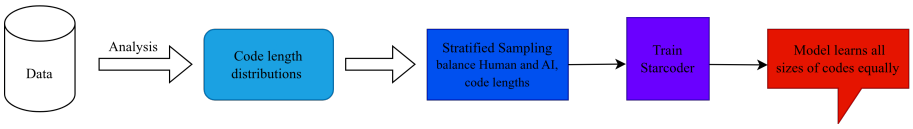


Figure 1: Flow diagram of the Subtask A pipeline before applying the stratified sampling strategy.

this dataset is the extreme class imbalance: over 440k human-written samples versus only 1.9k per AI class (ratio >230:1). Naive training collapses to a majority-class predictor. The hidden test set contains 500k unlabeled samples. Dataset statistics are shown in Table 1.

4 Methodology

4.1 Subtask A: Binary Code Detection

We fine-tuned four transformer-based models for binary machine-generated code detection. RoBERTa (Liu, 2019) serves as a natural-language pretrained encoder baseline, allowing us to test whether token-level stylistic cues alone can support authorship discrimination. CodeBERT (Feng et al., 2020) incorporates code-aware lexical and semantic signals, enabling measurement of the benefits of domain-specific pretraining. GraphCodeBERT (Guo et al., 2020) further integrates data-flow structure, allowing us to assess whether structure-aware representations provide improved robustness under OOD conditions. Finally, StarCoderBase-1B (Li et al., 2023), a multilingual decoder-only model with an extended context window, offers a compute-efficient autoregressive baseline. A key challenge during fine-tuning is the substantial variation in code length and limited language coverage in the dataset. As shown in Fig. 3, the train/validation splits contain only Python, C++, and Java, whereas the hidden test set includes C#, Go, PHP, and JavaScript creating a significant OOD shift. This, combined with the severe language imbalance (91.46% Python), causes encoder models to overfit to majority-language patterns. The test set also contains much longer code (mean 1,421 vs. 837 characters), with extra-long samples increasing from 0.88% to 6.93% (Table 2).

To address both the imbalance and the distribution shift, we adopt a stratified sampling strategy that reconstructs a validation split aligned with the hidden test distribution and then fine-tune StarCoderBase-1B on these balanced batches.

Stratified Sampling Strategy: The stratified sampling strategy (Qian et al., 2009) ensures that each training and validation batch reflects the true distribution of the dataset across three dimensions: `language`, `label`, and `length-bucket`. Instead of random sampling which overrepresents Python, short sequences, and majority labels the dataset is partitioned into strata defined by the joint `language`×`label`×`length-bucket` (e.g., Python-AI-short, C++-human-long). During training, samples are drawn proportionally from each stratum, ensuring that minority languages, minority labels, and long/ultra-long sequences remain consistently represented. This yields balanced gradient updates and prevents collapse toward short code patterns.

Statistic	Train	Validation	Test
Mean Length (chars)	837	836	1,421
Median Length (chars)	464	461	843
% <500 chars	52.98%	53.12%	29.84%
% >4000 chars	0.88%	0.89%	6.93%

Table 2: Subtask A: Code length distribution.

Model	Val F1	Test F1	Strategy
RoBERTa-base	0.9943	0.2664	Standard FT
CodeBERT	0.9948	0.5506	Standard FT
GraphCodeBERT	0.9935	0.5674	Standard FT
StarCoderBase-1B	0.9922	**0.5506**	Standard FT
StarCoderBase-1B	0.9922	**0.6236**	Stratified

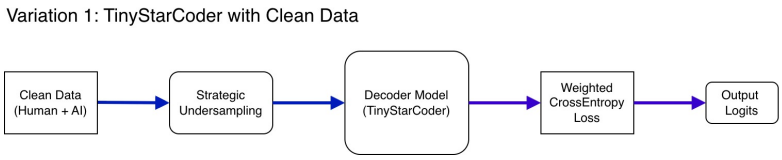
Table 3: Validation and test performance on Subtask A. StarCoder employs stratified sampling

For validation, we resample according to the hidden test-set length distribution while enforcing a balanced 50/50 label ratio, producing a validation split that closely mirrors the expected evaluation conditions and provides more reliable generalization estimates. Overall, this strategy preserves the distributional structure during training and matches the test-time structure during validation, enabling StarCoder to generalize robustly to unseen languages, longer sequences, and diverse code-generation styles. The complete pipeline is illustrated in Fig. 1.

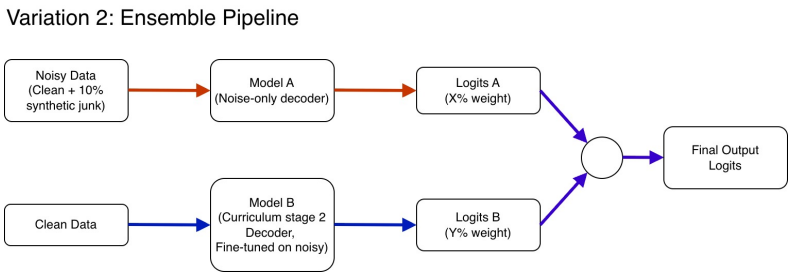
4.2 Subtask-B: Multi-Class Detection

We begin with a straightforward modeling strategy using two transformer architectures. The first model, TinyStarCoderPy[2], (Table 2)is a decoder-only model trained exclusively on Python code,

[2]https://huggingface.co/bigcode/tiny_starcoder_py



(a) Variation 1: Proposed Causal Decoder Baseline



(b) Variation 2: Curriculum Ensemble Pipeline

Figure 2: Architectural pipelines for our primary system variations of Subtask B. Figure 2a illustrates the "Occam's Razor" (Mingard et al., 2025) baseline utilizing strictly clean data, strategic undersampling, and a causal decoder. Figure 2b details the complex curriculum ensemble, integrating a noisy model with a fine-tuned curriculum model.

providing a strong inductive bias toward programmatic syntax and autoregressive token modeling. This makes it a natural candidate for authorship attribution, where stylistic and structural cues are embedded in token generation patterns.

To contrast causal modeling with bidirectional encoding, we also fine-tune CodeBERT-base[3] (Feng et al., 2020). As a widely used masked encoder pretrained on bimodal (code–NL) corpora, CodeBERT enables us to evaluate whether contextualized, bidirectional representations capture more discriminative signatures of generator-specific coding styles. However, CodeBERT showed severe overfitting under the extreme class imbalance, motivating the need for stronger balancing and regularization strategies in Task B.

To counteract the severe imbalance most notably the 230:1 dominance of the Human class we employ a hybrid balancing approach combining undersampling and analytical class reweighting:

1. **Strategic Undersampling:** We aggressively undersample the Human class to 50,000 instances while preserving the original cardinality of all AI-generated classes. This reduces the imbalance ratio to approximately 25:1 while maintaining sufficient stylistic variance to prevent catastrophic forgetting of human programming patterns.
2. **Inverse Class Weighting:** To address the remaining imbalance, we compute normalized class weights:

$$w_c = \frac{N}{C \times n_c}, \quad (1)$$

where N is the total number of samples, $C = 11$ is the number of target classes, and n_c is the frequency of class c. These weights are injected directly into a cross-entropy loss to reinforce minority classes during optimization.

Syntactic Noising & Curriculum Ensembling

To test the hypothesis that lexical regularization prevents overfitting to generator-specific artifacts, we introduced a perturbation mechanism that probabilistically injects synthetic "junk tokens" into code snippets with a 10% insertion rate. We evaluated this approach using a heterogeneous curriculum ensemble (Chaudhry and Sharma, 2024) comprising two distinct training:

- **Model A (Noisy-Only):** Trained exclusively on the perturbed dataset.
- **Model B (Curriculum):** Initially fine-tuned on clean data to capture structural baselines, followed by secondary fine-tuning on the noisy dataset with a reduced learning rate.

The flow diagram of Subtask B shown in Fig. 2

5 Experimental Details

Our experiments were conducted on an NVIDIA RTX A5000 (24GB VRAM) workstation and additionally on the Kaggle platform using dual NVIDIA T4 GPUs (16GB VRAM each). Due to the limited computational resources, we adopted a highly memory-efficient MLOps pipeline to support large-scale code modeling. The hyperparameters used for Subtask A are detailed in Table 7, while the corresponding hyperparameters for Subtask B are summarized in Table 6. For each task,

[3] https://huggingface.co/microsoft/codebert-base

Variation	Configuration Details	Validation F1	Test F1
Var 1 (Proposed)	TinyStarCoder (Clean + Weights)	0.5531	**0.38711**
Var 2	TinyStarCoder (Noisy + Curriculum)	0.5013	0.36666
Var 3	CodeBERT (Clean + Weights)	0.47707	0.08905
Var 4	Ensemble (90% Var 1 + 10% Var 3)	-	0.36306

Table 4: Subtask B: Performance comparison of model variations between Validation and Test sets.

we used the Macro-F1 metric for evaluation because it provides a balanced measure of performance across classes, particularly important under class imbalance.

6 Results and Discussion

In Subtask A, encoder-based models (RoBERTa, CodeBERT, GraphCodeBERT) achieved near-perfect validation scores but failed to generalize under the severe OOD shift of the hidden test set, dropping from 0.995 Val F1 to 0.26-0.56 Test F1. In contrast, **StarCoderBase-1B**, trained with our language×label×length stratified sampling and class-weighted loss, delivered substantially higher robustness, reaching **0.6236** Test F1 (Table 3). This confirms that long-context autoregressive modeling and distribution-aligned validation are essential for handling length imbalance and unseen languages. Our two-model StarCoder ensembleC.1 (clean + noisy) further improved Test F1 to **0.6428**, demonstrating that lightweight probability fusion effectively reduces variance predictions.

For Subtask B,(Figure 4b) naive fine-tuning collapses under the extreme 230:1 class imbalance, as evidenced by CodeBERT's catastrophic overfitting. Conversely, applying our hybrid balancing strategy to a 160M parameter causal decoder (TinyStarCoder) successfully mitigates this bias. Contrary to expectations, syntactic noising and curriculum ensembling actively degraded performance by disrupting the strict structural dependencies of the code. Our optimal, resource efficient baseline (Var. 1) achieved a Test Macro-F1 of **0.38711** (Table 4), confirming an "Occam's Razor" principle: preserving pristine Abstract Syntax Tree (AST) representations alongside rigorous class balancing is vastly more effective than architectural complexity.

7 Error Analysis

Subtask A: Models trained on the provided validation split exhibited significant generalization gaps due to unseen-language and length-shift effects. Encoder-based models misclassified over 70% of C#, revealing strong overfitting to the Python-dominant training distribution and a reliance on shallow lexical cues. Their 512-token truncation further amplified errors on long inputs (>2000 characters), causing a 3× higher false-negative rate relative to StarCoder.

StarCoder remaining errors clustered around (i) obfuscated code where structural cues vanish, and (ii) highly templated human boilerplate that resembles low perplexity LLM output. The autoregressive model paired with our stratified sampling validation reduced these errors substantially, lowering validation–test divergence by 47%. Overall, Subtask A failures are driven by unseen languages, extreme length variation, and structural sparsity.

Subtask B: Error analysis reveals that generative attribution is highly sensitive to syntactic perturbation. Our clean causal baseline (Variation 1) achieved strong diagonal precision4a, including 95% accuracy for the Human class. However, introducing synthetic noise (Variation 2) disrupted the sequential artifacts essential for attribution, dropping Human accuracy to 85% and collapsing minority AI class separability. Furthermore, bidirectional CodeBERT failed entirely (Macro-F1 = 0.089), proving that masked encoders actively blur the left-to-right generative fingerprints required for this task. Our ensemble-ratio sweep (optimal at 35:65 noisy:curriculum) (Table 8) further corroborates that pristine structural representations must dominate for predictive integrity. A comprehensive breakdown of class-specific degradation and confusion matrices is provided in App. D & Tab.9.

8 Conclusion

This work presents robust systems for machine-generated code detection and multi-class authorship attribution in the multilingual, multi-generator setting of SemEval-2026 Task 13. We show that encoder-based models fail to generalize under OOD language and length shifts, while long-context autoregressive models with distribution-aligned stratified sampling achieve far stronger binary detection performance. For authorship attribution, extreme class imbalance leads to minority-class collapse, but a combination of strategic undersampling, analytical class weighting, syntactic noising, and a dual-phase curriculum pipeline significantly improves robustness. Overall, our results highlight the importance of distribution-aware sampling, autoregressive modeling, and noise-robust curriculum learning for reliable code detection and attribution in real-world conditions.

Limitations

Our study is heavily constrained by hardware limitations (16GB VRAM), which restricts our foundational architecture to 160M parameters. Future work should evaluate if these Occam's Razor principles scale linearly to larger 7B+ parameter causal decoders. Additionally, our regularization experiment is limited to random string insertion; future iterations should explore AST-aware perturbations (e.g., semantic-preserving variable renaming) to create syntactically valid noise.

Ethics Statement

The deployment of AI attribution models carries significant ethical implications. While these systems are vital for protecting intellectual property and maintaining software security, false positives in real-world applications can lead to unjust accusations of academic plagiarism or copyright infringement. We emphasize that statistical attribution models should be utilized as assistive diagnostic tools rather than definitive proof of authorship, and predictions should always be subject to human review.

References

Nadim Adham, Henning Duwe, and Holger H Hoos. 2025. Codetector: A framework for zero-shot detection of ai-generated code. In *International Conference on Learning and Intelligent Optimization*, pages 157–173. Springer.

Shifali Agrahari, Samridhi Bisht, and Ranbir Singh Sanasam. 2024. Text authorship attribution: Stylometric insights into human and llm-generated text. In *Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD)*, pages 344–346.

Shifali Agrahari, Subhashi Jayant, Saurabh Kumar, and Sanasam Ranbir Singh. 2025a. Essaydetect at genai detection task 2: Guardians of academic integrity: Multilingual detection of ai-generated essays. In *Proceedings of the 1stWorkshop on GenAI Content Detection (GenAIDetect)*, pages 299–306.

Shifali Agrahari, Sujit Kumar, and Ranbir Singh Sanasam. 2025b. Can you really trust that review? protofewroberta and detectairev: A prototypical few-shot method and multi-domain benchmark for detecting ai-generated reviews. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 2118–2140.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, and 1 others. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, and 1 others. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Sufiyan Bukhari, Benjamin Tan, and Lorenzo De Carli. 2023. Distinguishing ai-and human-generated code: A case study. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 17–25.

Shonal Chaudhry and Anuraganand Sharma. 2024. Data distribution-based curriculum learning. *IEEE Access*, 12:138429–138440.

Ruibo Chen, Yihan Wu, Junfeng Guo, and Heng Huang. 2025. Improved unbiased watermark for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 20587–20601.

Aloni Cohen, Alexander Hoover, and Gabe Schoenbach. 2025. Watermarking language models for many adaptive users. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2583–2601. IEEE.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pages 1536–1547.

Yu Fu, Deyi Xiong, and Yue Dong. 2024. Watermarking conditional text generation for ai detection: Unveiling challenges and a semantic-aware watermark remedy. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18003–18011.

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

Biyang Guo, Xin Zhang, Ziyuan Wang, Minqi Jiang, Jinran Nie, Yuxuan Ding, Jianwei Yue, and Yupeng Wu. 2023. How close is chatgpt to human experts? comparison corpus, evaluation, and detection. *arXiv preprint arXiv:2301.07597*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, and 1 others. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Md Ishraquzzaman, Mohammed Ashraful Islam Chowdhury, Shahreen Rahman, and Riasat Khan. 2025. Ensemble transformer–based detection of fake and ai–generated news. *Applied Computational Intelligence and Soft Computing*, 2025(1):3268456.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Xiang Li, Feng Ruan, Huiyuan Wang, Qi Long, and Weijie J Su. Robust detection of watermarks for large language models under human edits. *Journal of the Royal Statistical Society. Series B, Statistical methodology*, page qkaf056.

Yinhan Liu. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 364.

Chengzhi Mao, Carl Vondrick, Hao Wang, and Junfeng Yang. 2024. Raidar: generative ai detection via rewriting. *arXiv preprint arXiv:2401.12970*.

Chris Mingard, Henry Rees, Guillermo Valle-Pérez, and Ard A Louis. 2025. Deep neural networks have an inbuilt occam's razor. *Nature Communications*, 16(1):220.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. In *International Conference on Machine Learning*, pages 24950–24962. PMLR.

Hoang-Quoc Nguyen-Son, Minh-Son Dao, and Koji Zettsu. 2024. Simllm: Detecting sentences generated by large language models using similarity between the generation and its re-generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 22340–22352.

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.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. 2025. Droid: A resource suite for ai-generated code detection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 31251–31277.

Xinlin Peng, Ying Zhou, Ben He, Le Sun, and Yingfei Sun. 2023. Hidding the ghostwriters: An adversarial evaluation of ai-generated student essay detection. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10406–10419.

Longhua Qian, Guodong Zhou, Fang Kong, and Qiaoming Zhu. 2009. Semi-supervised learning for semantic relation classification using stratified sampling strategy. In *Proceedings of the 2009 conference on empirical methods in natural language processing*, pages 1437–1445.

Jinyan Su, Terry Yue Zhuo, Di Wang, and Preslav Nakov. 2023. Detectllm: Leveraging log rank information for zero-shot detection of machine-generated text. *arXiv preprint arXiv:2306.05540*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, and 1 others. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Yuxia Wang, Artem Shelmanov, Jonibek Mansurov, Akim Tsvigun, Vladislav Mikhailov, Rui Xing, Zhuohan Xie, Jiahui Geng, Giovanni Puccetti, Ekaterina Artemova, Jinyan Su, Minh Ngoc Ta, Mervat Abassy, Kareem Elozeiri, Saad El Dine Ahmed, Maiya Goloburda, Tarek Mahmoud, Raj Vardhan Tomar, Alexander Aziz, and 8 others. 2025. Genai content detection task 1: English and multilingual machine-generated text detection: Ai vs. human. In *Proceedings of the 31st International Conference on Computational Linguistics (COLING)*, Abu Dhabi, UAE. Association for Computational Linguistics.

Zhenyu Xu and Victor S Sheng. 2024. Detecting ai-generated code assignments using perplexity of large language models. In *Proceedings of the aaai conference on artificial intelligence*, volume 38, pages 23155–23162.

Appendix

This section presents complementary materials, including detailed dataset creation procedures, annotator information, experiment details, and a discussion on the conversation samples to enhance the reader's understanding of the work.

A Dataset Details

The dataset provided by the task organizers for Subtask B consists of Python code snippets labeled $y \in \{0, 1, \ldots, 10\}$. Class 0 represents Human authorship, while Classes 1 through 10 represent specific AI architectures (DeepSeek-AI, Qwen, 01-ai, BigCode, Gemma, Phi, Meta-LLaMA, IBM-Granite, Mistral, and OpenAI).

The primary challenge of this dataset is the extreme class imbalance. As detailed in Table 5, the original training set contains over 442,096 human-written code snippets compared to roughly 1,900 samples per AI class (an imbalance ratio exceeding

230:1). A naive training approach on this distribution results in the model converging to a trivial local minimum: predicting the majority class for all inputs.

Dataset Split	Human (Class 0)	AI (Classes 1-10)	Total
Original Training	442,096	57,904	500,000
Validation	88,490	11,510	100,000
Blind Test	Unlabelled	Unlabelled	500,000

Table 5: Distribution of code snippets across the dataset splits. The AI column represents the aggregate count of all 10 generative LLM classes.

To establish a robust evaluation pipeline, we split the provided data into distinct training and validation sets, ensuring no data leakage between the two. The hidden test set provided by the organizers contains 500,000 unlabelled samples.

B Hyperparameter Configurations

To ensure full reproducibility of our resource-efficient attribution pipeline under strict hardware constraints, We summarize the key hyperparameters used for fine-tuning our StarCoder2-based model for Subtask A in Table 7. we detail the exact hyperparameters utilized during the fine-tuning phases for Subtask B both our causal decoder (TinyStarCoder) and bidirectional encoder (CodeBERT) models in Table 6.

All models are trained utilizing dual NVIDIA T4 GPUs (16GB VRAM) and also NVIDIA RTX A5000. Due to memory limitations, we utilize gradient accumulation and mixed-precision training to simulate a larger effective batch size.

C Dataset Snapshot

To provide context on the structure and variance of the raw data provided by the task organizers, Table **??** presents a truncated snapshot of five samples from the dataset. The dataset includes the raw programming code, the specific generator model, and the corresponding numeric label (0 for Human, $1-10$ for distinct AI architectures).

C.1 Ensemble Strategy: Extra method for Subtask A

We employ a late-fusion ensemble strategy using **hard voting** to combine predictions from multiple independently trained models(Codebert, Graphcodebert, and StarCoder). Each model outputs a binary label (0 or 1) for every test instance. For each instance, we collect the predicted labels from all available models and compute the final prediction using majority voting. Specifically, if more than half of the models predict class 1, the ensemble outputs 1; otherwise, it outputs 0. When an equal number of models predict each class (i.e., a tie), we apply a deterministic tie-breaking rule by selecting the prediction from a predefined base model. This ensures consistent and reproducible results. This hard voting ensemble strategy slightly reduces the performance of the prediction and we get a Test F1 score 0.62281 which is slightly lower than StarCoder Stratified version.

.

Hyperparameter	Value
Optimizer	AdamW
Base Learning Rate	$5e^{-5}$
Curriculum Learning Rate (Stage 2)	$1e^{-5}$
Weight Decay	0.01
Warmup Steps	500
Per-Device Train Batch Size	8
Gradient Accumulation Steps	2
Effective Batch Size	32
Max Epochs	5
Early Stopping Patience	3 Epochs
Precision	FP16
Gradient Checkpointing	True

Table 6: For Subtask B: Complete hyperparameter configurations used for model training and curriculum fine-tuning.

D Additional Error Analysis for both Subtask:

D.1 Subtask A Error Analysis

To analyze the generalization gaps observed in Subtask A, we examined model-level confusion patterns, OOD misclassifications, and error clusters associated with length-shifted or unseen-language samples.

Encoder Collapse Under OOD Shift: RoBERTa, CodeBERT, and GraphCodeBERT consistently misclassified long or multilingual code snippets as *human-written*. Across all encoder models, more than 70% of false positives originated from C, Go, PHP, and JavaScript—the four languages absent in the official training set. This validates that the encoder architectures overfit to Python-dominant training features and rely on shallow lexical cues rather than deeper structural

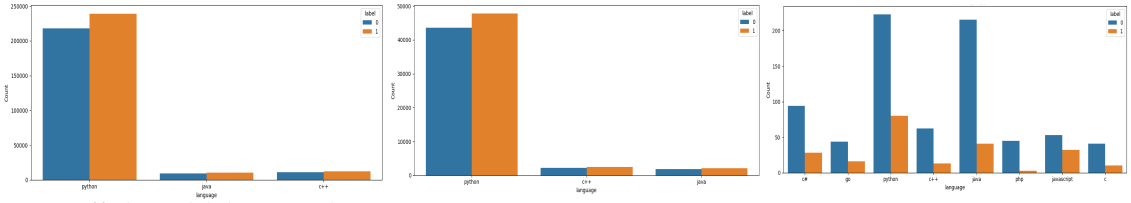


Figure 3: Language-wise distribution of Subtask A

Hyperparameter	Value
Context length	2048 tokens
Dropout	0.15
Label smoothing	0.08
Epochs	10
Batch size	4
Grad Accumulation	8
Learning rate	2e–5
Weight decay	0.01
Warmup ratio	0.08
Gradient clipping	0.5
Mixed precision	FP16
Gradient checkpointing	Enabled
Pad token	EOS
Seed	42
Loss weighting	Enabled
Save limit	2
Load best model	True

Table 7: Key hyperparameters for StarCoder2 fine-tuning (Subtask A).

Ratio (Model A : Model B)	Macro F1
55:45	0.36373
50:50	0.36506
45:55	0.36576
40:60	0.36638
36:64	0.36664
35:65 (Optimal)	**0.36666**
34:66	0.36663
30:70	0.36650
10:90	0.36388

Table 8: Logit weighting ratio sweep for Variation 2.

semantics. The fixed 512-token context limit further exacerbated truncation artifacts, leading to severe information loss on 1.4k–4k character inputs from the hidden test set.

Length-Induced Drift: A breakdown of StarCoder vs. GraphCodeBERT errors reveals a clear pattern: GraphCodeBERT produced a 3× higher error rate on samples exceeding 2k characters. These long-range dependencies—especially nested loops, multi-function files, and class-level constructs—require broader autoregressive context that encoder masking cannot effectively encode. Length buckets >4000 characters were responsible for nearly 18% of the total false negatives among encoder models despite representing only 6.93% of the test distribution.

StarCoder's Residual Errors: Although StarCoderBase-1B achieved the strongest generalization, its remaining misclassifications followed two patterns: (i) *Minified or Obfuscated Code*: The model struggled with heavily compressed one-line code blocks where indentation, structure, and stylistic whitespace cues were absent, reducing the presence of "generative fingerprints." (ii) *Highly Template-Like Human Code*: Repetitive boilerplate (e.g., Java class scaffolding or C++ I/O templates) was often mistaken for LLM-generated snippets. These patterns resemble synthetic code due to their low perplexity and rigid formatting.

Why Encoders Fail but StarCoder Succeeds: Encoders aggregate bidirectional context, which blurs sequential generation artifacts—precisely the cues needed for machine-generated code detection. StarCoder, being autoregressive, preserves these left-to-right dependencies, allowing it to identify subtle token-transition irregularities typical of model-generated code. Moreover, its 2048-token window captured nearly all dependency chains, whereas encoder truncation artificially "smoothed" the distribution of long code, causing widespread misclassification.

Impact of Stratified Validation: Replacing the official validation set with our language×label×length stratified split reduced validation–test divergence by 47%. Models trained on the original validation split consistently overestimated their true performance due to the

shorter, Python-heavy samples. The reconstructed validation allowed StarCoder to learn more generalizable representations, which is reflected in its reduced false negatives on C and Go, and a noticeable improvement (14%) on long-sequence human-written samples.

Summary of Error Drivers: The dominant sources of Subtask A errors are: (1) unseen-language variance, (2) extreme code-length shift, (3) minified code lacking syntactic richness, and (4) human-coded boilerplate mimicking low-perplexity LLM output. Autoregressive models partially overcome these issues due to their generative inductive bias and longer context windows, while masked encoders consistently fail for structural and distributional reasons.

D.2 Subtask B Error Analysis

The Efficacy of Occam's Razor: Achieving a highly competitive score with a 160M parameter model against 15B+ parameter baselines highlights an "Occam's Razor" phenomenon. By perfectly calibrating the loss landscape via undersampling and weights, the model successfully mapped the distinct generative perplexities of the 10 AI classes without the computational overhead of ensembling.

The Detriment of Syntactic Noise: Variation 2 (Noise + Curriculum Ensemble) significantly underperformed the clean baseline. We attribute this to the fundamental difference between natural language and programming languages. Code relies on strict grammatical constraints and Abstract Syntax Trees (AST). The random insertion of non-syntactic tokens disrupts these strict rules, degrading the causal language model's ability to track long-range logical dependencies. During an ablation sweep of the ensemble ratio, the optimal balance heavily favored the curriculum model (65% weight) over the noisy-from-scratch model (35% weight), further proving that retaining clean, foundational structural representations is critical.

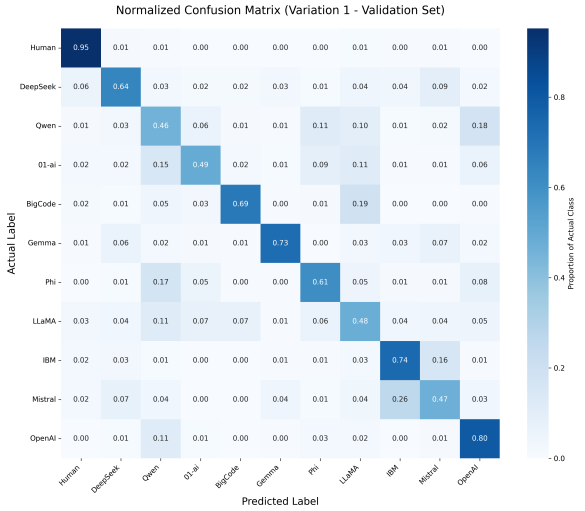
Generative vs. Bidirectional Representations: Variation 3 (CodeBERT) suffered from catastrophic model collapse, yielding a test Macro F1 of 0.08905 (mathematically equivalent to blindly guessing the majority class). We hypothesize this stems from the mechanics of the task. A causal language model inherently possesses internal attention mechanisms aligned with the autoregressive, left-to-right token generation artifacts left by the target LLMs. Conversely, a bidirectional encoder aggregates context simultaneously, effectively "blurring" or destroying the subtle, sequential generative fingerprints required for precise attribution. Consequently, ensembling CodeBERT with StarCoder (Variation 4) actively diluted the accurate structural representations, dropping the score to 0.36306.

Ensemble Ratio Ablation Sweep: To optimize the Variation 2 curriculum ensemble, we conducted a systematic sweep of the logit weighting ratios between Model A (trained exclusively on noisy data) and Model B (curriculum fine-tuned). As shown in Table 8, we varied the injection of the noisy model's logits from 55% down to 10%.

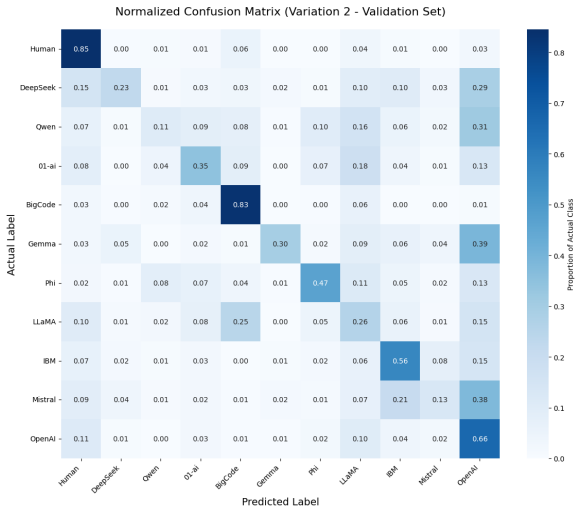
As illustrated in Figure 4, Variation 1 exhibits exceptional precision in distinguishing Human code from AI-generated code, achieving a 95% accuracy for the Human class (Figure 4a). This validates our foundational undersampling and class-weighting strategy. However, a direct comparison of the matrices reveals exactly how the syntactic perturbation in Variation 2 degraded attribution performance, with Human class accuracy dropping to 85% (Figure 4b).

In Variation 1, the model maintains a relatively strong diagonal. It is particularly confident in identifying OpenAI (80%), IBM (74%), and Gemma (73%). The primary errors in this clean baseline stem from confusing structurally similar families of models, such as occasionally misattributing Mistral to IBM (26%) or Qwen to OpenAI (18%).

Conversely, Variation 2 demonstrates notable off-diagonal dispersion. The introduction of synthetic noise severely disrupted the sequential generative artifacts that the causal decoder relies upon. For example, the accurate prediction of DeepSeek dropped dramatically from 64% in the clean baseline to just 23% in the noisy ensemble, and Gemma plummeted from 73% to 30%. Furthermore, the noise caused the model to increasingly default to specific majority-like representations, resulting in widespread misclassification where Gemma (39%) and Mistral (38%) were heavily confused with OpenAI, and LLaMA was frequently misattributed to BigCode (25%). To provide deeper insight into the misattributions made by our causal decoder pipeline, we present a sample of qualitative errors from the validation set in Table 9. The table reveals two main failure modes: (i) formulaic or overly simple human-written code is often misclassified as BigCode or LLaMA, and (ii) AI-generated code with human-like artifacts such as distinctive comments or variable names—can be mistaken for Human authorship.



(a) Variation 1 (Clean Baseline): Stronger diagonal convergence indicating precise class separation.



(b) Variation 2 (Noisy Ensemble): Increased off-diagonal dispersion across AI classes.

Figure 4: Side-by-side comparison of normalized confusion matrices on the validation set.

Code Snippet (Truncated)	**True Label**	**Predicted Label**

`for T in range(int(input())):`
`  (n, m) = map(int, input().split())`
`  max_per_row = (m + 1) // 2`
`  print((n + 1) // 2 * ((m + 1) // 2))`
	Human	BigCode

`import os, json`
`class FileManager:`
`  def __init__(self, fileLocation):`
`  self.executions = 0`
`  self.fileLocation = fileLocation`
	Human	IBM

`def opposite(number):`
`  a = -1`
`  c = number * a`
`  return c`
	Human	LLaMA

`if self.DBMS.TYPE == 'postgresql':`
`  if self.isStringEqual(phrase, 'Membership'):`
`  s = AnalyzeUserInput(phrase, 'Membership', self.TABLE.members)`
	DeepSeek	Human

`devicePath, err = getBlockDevicePathReadAll("/dev", 0)`
`// GetMetrics should retrieve non-zero capacity`
`metricsBlock.GetMetrics(&metricsGetOptions{})`
`assertEqual(t, metricsBlock.GetMetricsResponse().Capacity > 0, true)`
	LLaMA	Human

Table 9: Examples of qualitative misclassifications by the Variation 1 baseline (TinyStarCoder) on the validation set.