YNU-HPCC at SemEval-2026 Task 13: Robust Machine-Generated Code Detection under Distribution Shifts

Lixian Xing, Jin Wang and Xuejie Zhang
School of Information Science and Engineering
Yunnan University
Kunming, China
xinglixian@stu.ynu.edu.cn, {wangjin, xjzhang}@ynu.edu.cn

Abstract

As Large Language Models (LLMs) become prevalent in software development, distinguishing machine-generated from human-written code is increasingly important. This paper describes the system developed by the YNU-HPCC team for SemEval-2026 Task 13, which evaluates detection under cross-language, multi-generator, and hybrid settings. Three modeling paradigms are systematically examined: encoder-based fine-tuning, feature-based machine learning, and task-specific robustness strategies. For Subtask A (Binary Detection), frozen pre-trained encoders and shallow stylometric features exhibit stronger cross-domain robustness than full fine-tuning, with indentation entropy identified as a key discriminative signal. For Subtask B (Multi-Class Attribution), a hierarchical two-stage framework is adopted to decouple humanmachine discrimination from fine-grained generator attribution, alleviating severe class imbalance. For Subtask C (Hybrid Detection), a token-level splicing augmentation strategy combined with Supervised Contrastive Learning and Focal Loss is employed to model intra-sample stylistic variation. According to the official leaderboard, our system ranked 12th out of 81 teams in Subtask A, 14th out of 34 in Subtask B, and 8th out of 32 in Subtask C.

1 Introduction

LLMs are increasingly used for code generation in competitive programming, education, and industrial development (Chen et al., 2021). Although these systems improve productivity, they raise concerns regarding authorship attribution, academic integrity, and software reliability. Automatically generated code may introduce subtle logical errors or stylistic artifacts that are difficult to detect manually. Consequently, reliable detection of machine-generated code has become an important research problem.

SemEval-2026 Task 13(Orel et al., 2026b) evaluates detection systems under diverse and heterogeneous conditions (Orel et al., 2025c,a). The task includes three subtasks: (i) binary human vs. machine detection across seen and unseen languages and domains; (ii) multi-class attribution across multiple LLM families under severe class imbalance; and (iii) hybrid and adversarial detection, where human and machine contributions coexist within a single snippet. These settings emphasize robustness to distributional variation rather than purely in-domain accuracy.

Recent studies suggest that performance degradation under unseen languages, generator families, and superficial edits remains substantial even for strong detectors (Orel et al., 2025a). Furthermore, code differs from natural language in that many tokens are syntactically constrained and exhibit low entropy, which weakens naïve likelihood-based detection signals. Therefore, the central challenge of machine-generated code detection can be framed as one of distributional robustnessthe ability to generalize across languages, domains, generator families, and intra-sample stylistic shifts.

In this paper, three modeling paradigms are systematically investigated in the shared-task setting: encoder-based fine-tuning, feature-based machine learning, and task-specific strategies for handling imbalance and hybrid modeling. For Subtask A, shallow stylometric features outperform deeper semantic representations in cross-domain evaluation. For Subtask B, a hierarchical two-stage framework is introduced to decouple majority-class filtering from fine-grained generator attribution. For Subtask C, a local style inconsistency modeling strategy based on token-level splicing is proposed to enhance robustness to intra-sample distribution shifts.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the proposed methodologies. Section 4

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 1582–1590
July 3-4, 2026 ©2026 Association for Computational Linguistics

describes the experimental setup and results. Section 5 concludes the paper. Supplementary details are included in the Appendix.

2 Related Work

Recent work on AI-generated code detection spans supervised transformer-based classifiers, code-aware zero-shot detectors, and increasingly realistic multilingual benchmarks. Early supervised methods such as GPTSniffer showed that pretrained code encoders can distinguish human-written from model-generated code in narrow settings (Nguyen et al., 2024). Subsequent zero-shot approaches adapted generic machine-generated text detection to code by introducing code-specific probability modeling, perturbation strategies, or rewriting-based similarity signals (Xianjun et al., 2023; Shi et al., 2025; Ye et al., 2025; Ashkenazi et al., 2025). More recent benchmark papers broaden the problem to unseen languages, unseen generators, humanAI coauthorship, paraphrase, and adversarial humanization, consistently showing that robust generalization remains challenging (Orel et al., 2025b,d; Guo et al., 2025; Orel et al., 2026a).

The multiclass attribution setting of Subtask B is closely related to code stylometry and source-code authorship analysis. Prior work on source-code authorship verification shows that contrastive transformer encoders can learn strong stylometric representations, while recent LLM-oriented studies demonstrate that attribution among generator families is feasible in controlled settings using encoder-style architectures over code tokens and style signals (Álvarez-Fidalgo and Ortin, 2025; Dipongkor et al., 2025; Bisztray et al., 2025). These findings motivate framing Subtask B not merely as multiclass classification, but as a stylometric attribution problem under imbalance and family-level similarity.

Pretrained code models offer complementary inductive biases for Task 13. Token-level models such as CodeBERT provide strong baselines, while GraphCodeBERT and UniXcoder incorporate data-flow or AST/comment information that may better support cross-language robustness and mixed-authorship detection (Feng et al., 2020; Guo et al., 2021, 2022). Encoder-decoder and instruction-tuned models such as CodeT5+ and Code Llama further suggest that transfer from broad code pretraining can be useful, although recent detection literature indicates that direct transfer from generic text detectors is often insufficient without code-specific adaptation (Wang et al., 2023; Rozière et al., 2023; Xianjun et al., 2023; Shi et al., 2025).

3 Methodology

Robust machine-generated code detection under heterogeneous conditions requires generalization across languages, generator families, and intra-sample stylistic shifts. To systematically examine modeling strategies under these constraints, three paradigms are investigated: (1) encoder-based fine-tuning, (2) feature-based classification, and (3) task-specific robustness strategies.

3.1 Encoder-based Fine-tuning

Pre-trained code encoders, including UniXcoder[1] (Guo et al., 2022) and GraphCodeBERT (Guo et al., 2021), are employed as backbone models. These models are specifically designed for source code representation learning. UniXcoder adopts a unified cross-modal pre-training objective over code, comments, and abstract syntax trees, and can function as an encoder, decoder, or both. GraphCodeBERT incorporates data-flow structure to enhance semantic and structural understanding of code. Given an input snippet x, contextualized representations are obtained and fed into a linear classifier. Two training settings are explored: (i) full fine-tuning of all parameters, and (ii) frozen encoder with only the classification head updated. This paradigm serves as a deep representation baseline under cross-domain evaluation.

3.2 Feature-based Classification

To reduce reliance on distribution-specific semantic representations, a feature-based paradigm is adopted. Extracted features are fed into a LightGBM classifier (Ke et al., 2017).

Shallow Stylometric Features. Surface-level statistical and formatting patterns are computed, including token counts, lexical ratios, indentation entropy, and structural depth indicators. These features aim to capture coding habits and layout consistency.

[1] We use the `microsoft/unixcoder-base-nine` variant, which extends language coverage to nine programming languages: https://huggingface.co/microsoft/unixcoder-base-nine.

AST Structural Features. Source code is parsed with Tree-Sitter[2] to obtain structural metrics, including node count, tree depth, branching factor, and node-type entropy. These features encode syntactic complexity without relying on generator-specific semantics.

Logits-Based Predictability Features. Frozen code-oriented LLMs are used as probabilistic scorers. For a token sequence $x = (t_1, \ldots, t_n)$, token-level probability distributions are computed, and summary statistics such as mean negative log-likelihood, entropy, margin, and miss rate are derived. Specifically, CodeLlama-7B (Rozière et al., 2023), Llama-3.1-8B (Team, 2024), CodeQwen-1.5-7B (Bai et al., 2023), and Nxcode-orpo-7B (Hong et al., 2024) are employed following the model selection criteria of CoDet-M4. Logit-based statistics are extracted independently for each model and concatenated to form the final predictability feature vector, thereby reducing scorer-specific bias.

Detailed definitions and calculation formulas are provided in Appendix A.

3.3 Hierarchical Two-Stage Framework (Subtask B)

To address severe class imbalance in Subtask B, a two-stage framework is adopted. Stage 1 performs binary humanmachine classification. Stage 2 applies multi-class generator attribution to samples predicted as machine-generated. This decomposition reduces the dominance of the majority class in fine-grained classification.

3.4 Local Style Inconsistency Modeling (Subtask C)

Hybrid detection requires modeling intra-sample distribution shifts. To enhance robustness, a data augmentation strategy is introduced to generate synthetic hybrid samples before training. Specifically, human-written snippet H and machine-generated snippet M are sampled from the same programming language subset to avoid cross-language artifacts. The two sequences are tokenized and truncated, and a synthetic hybrid sample is constructed as:

$$\tilde{X} = \texttt{concat}(H_{1:i}, M_{j:k}) \quad (1)$$

[2]Tree-Sitter is an open-source incremental parsing system for programming tools, capable of generating concrete and abstract syntax trees(AST) for multiple programming languages. See https://github.com/tree-sitter/tree-sitter.

where concat denotes sequence concatenation.

In addition, Supervised Contrastive Learning (Khosla et al., 2020) is applied to improve representation separability, and Focal Loss (Lin et al., 2017) is employed to mitigate class imbalance.

4 Experimental & Results

In this section, the proposed modeling paradigms are empirically evaluated on all three subtasks. Experimental settings are first described, followed by detailed performance analysis for binary detection, multi-class attribution, and hybrid detection.

4.1 Experimental Setup

Dataset. All datasets are provided by the task organizers, and no external data is used. To mitigate data scarcity in the Hybrid category of Subtask C, a token-level splicing strategy is used to generate synthetic hybrid samples before training. Human-written and machine-generated snippets are sampled within the same programming language subset. For each pair, token sequences are truncated and concatenated according to a randomly sampled split ratio $\alpha \in [0.25, 0.75]$, producing a fixed-length sequence of 512 tokens. The concatenation order is randomized to simulate diverse transition patterns. The resulting samples are labeled as Hybrid and added to the training set.

Evaluation Metrics. Macro F1-score is adopted as the primary evaluation metric for all subtasks, following the official setting. The metric computes the unweighted mean of per-class F1 scores, treating all classes equally regardless of their frequency. For class i, the F1-score is defined as the harmonic mean of precision and recall. The overall score is computed as:

$$\text{Macro-F1} = \frac{1}{C} \sum_{i=1}^{C} F_i \quad (2)$$

Hyperparameter Setting. For all deep learning models, the initial learning rate is set to $1.41e^{-4}$ with a weight decay of $1e^{-2}$. An effective batch size of 96×3 is used via gradient accumulation. Training is conducted for up to 3 epochs with a linear learning rate scheduler.

The maximum input length is fixed to 512 tokens for fine-tuning. For zero-shot inference and feature extraction of logits from large language models, the maximum length is extended to 1024 tokens.

For the LightGBM classifier, the learning rate is set to $5e^{-3}$, the maximum number of leaves is 32, and the minimum number of data points per leaf is 20. All other hyperparameters are kept at their default values.

4.2 Subtask A: Binary Machine-Generated Code Detection

Two encoder-based paradigms were evaluated using UniXcoder-base and GraphCodeBERT-base: full fine-tuning and frozen encoder with a linear classification head. The results are shown in Table 1.

Frozen encoders consistently outperform full fine-tuning. UniXcoder improves from 0.43 to 0.47 F1 when the backbone is frozen, and GraphCodeBERT achieves the best encoder-based F1 of 0.51 under this setting. In contrast, full fine-tuning leads to performance degradation.

This trend suggests that updating all encoder parameters may introduce overfitting to training-domain artifacts, reducing generalization to unseen languages and domains. Freezing the backbone preserves pre-trained representations while limiting distribution-specific adaptation, resulting in improved cross-domain robustness.

Method	F1
UniXcoder	0.43
UniXcoder(frozen)	0.47
GraphCodeBERT(frozen)	**0.51**

Table 1: Encoder-based method classification results.

Shallow	AST	Logits	F1
		✓	0.48
	✓		0.58
	✓	✓	0.54
✓			**0.69**
✓		✓	0.43
✓	✓		0.59
✓	✓	✓	0.55

Table 2: Ablation study results. Shallow features contain 21 dimensions, AST features contain 10 dimensions, and Logits-based features contain 10 dimensions.

Feature-based Machine Learning. Table 2 reports the LightGBM results under different feature combinations. The Shallow feature set alone achieves the best performance (0.69 F1), outperforming both AST (0.58) and Logits (0.48). Notably, feature fusion does not improve performance: combining Shallow with Logits leads to a substantial drop (0.43), and adding AST features reduces F1 to 0.59. These results suggest that deeper structural or model-derived representations do not enhance cross-domain robustness in this setting.

A possible explanation is that the logits features are extracted from only four frozen scorer LLMs, while the test set contains a broader set of generators. Therefore, the resulting statistics reflect relative predictability with respect to selected scorers rather than generator-invariant signals, which may limit generalization. Consequently, subsequent analysis focuses on the Shallow-only configuration.

To further interpret this behavior, SHapley Additive exPlanations (SHAP) (Lundberg and Lee, 2017) is conducted on the Shallow model (Figure 1). Indentation entropy emerges as the most discriminative feature. Higher indentation entropy is associated with machine-generated predictions, indicating more regular and hierarchical formatting patterns. In contrast, human-written code tends to have a flatter, less uniform layout. Similar trends are observed for blank-line ratio and line-length statistics, reinforcing the importance of surface-level stylistic cues.

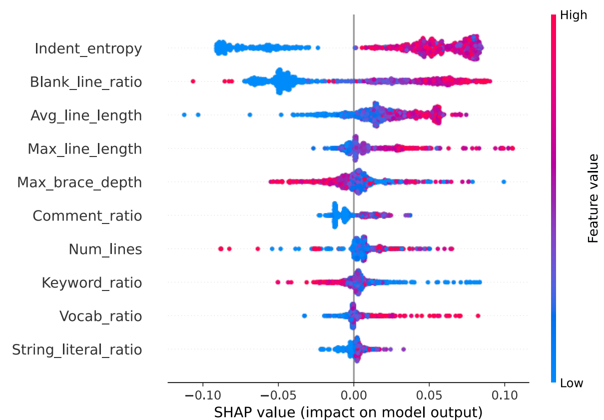

Figure 1: SHAP for LightGBM classifier.

4.3 Subtask B: Multi-Class Authorship Detection

Subtask B presents severe class imbalance: the Human class contains 442k samples, whereas certain generator classes contain as few as 2k sam-

ples. Under such skewed distributions, single-stage multi-class classification tends to bias predictions toward the majority class.

To mitigate this effect, a hierarchical two-stage framework is adopted. In Stage 1, a binary classifier distinguishes Human-written from Machine-generated code. In Stage 2, samples predicted as machine-generated are passed to a fine-grained classifier to identify the generator family (10 classes).

As shown in Table 4, the two-stage framework slightly improves performance over the single-stage baseline (0.39 vs. 0.37 F1). The Stage 1 classifier achieves an F1 score of 0.97, effectively separating most human-written samples from those generated by the generator. However, generator attribution in Stage 2 remains challenging due to subtle inter-generator similarities and residual imbalance. These results suggest that hierarchical decoupling alleviates majority bias but does not fully resolve the difficulty of fine-grained attribution.

Method	F1
GraphCodeBERT	0.97

Table 3: Stage 1 binary classification results on validation.

Method	F1
Single-stage	0.37
Two-stage	0.39
+Focal Loss	0.39
+MultiTask(lang head)	0.39

Table 4: Two-stage and Single-stage final classification results.

4.4 Subtask C: Hybrid Code Detection

Detecting *Hybrid* code (partially machine-generated) and *Adversarial* code presents a unique challenge due to the subtle semantic boundaries between classes. We use UniXcoder as our backbone. We integrated SupCon to enforce tighter clustering of same-class representations in the embedding space, alongside Focal Loss to prioritize hard-to-classify examples. To address the scarcity of hybrid training samples, we implemented a Token-level Splicing strategy. This involves randomly concatenating truncated human and machine code segments during training to simulate diverse transition patterns found in real-world hybrid code.

As shown in Table 5, the baseline model achieved an F1 score of 0.57. Incorporating SupCon and Focal Loss resulted in a significant performance boost to 0.60. After applying data augmentation, it improved further to 0.62.

Method	F1
UniXcoder	0.57
+SupCon,Focal	0.60
+Data Augmentation	**0.62**

Table 5: Task-C classification results.

5 Conclusion

This paper presented the YNU-HPCC system for SemEval-2026 Task 13 and systematically evaluated multiple modeling paradigms under heterogeneous detection settings. Across the three subtasks, different forms of distributional variation were encountered, including cross-domain shifts, severe class imbalance, and intra-sample stylistic mixture. Our findings indicate that robustness in machine-generated code detection does not solely depend on increasing representational depth. Instead, shallow stylometric modeling, hierarchical decomposition, and mixed-distribution exposure each play important roles under their respective task constraints. While fine-grained generator attribution remains challenging, especially under extreme imbalance, the proposed strategies consistently mitigate distribution-induced degradation. Future work will explore more principled domain generalization techniques and generator-invariant representations to improve robustness in realistic deployment scenarios further.

6 Acknowledgments

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant Nos.61966038 and 62266051. The authors would like to thank the anonymous reviewers for their constructive comments.

References

David Álvarez-Fidalgo and Francisco Ortin. 2025. Clave: A deep learning model for source code authorship verification with contrastive learning and

transformer encoders. *Information Processing & Management*, 62(3):104005.

Maor Ashkenazi, Ofir Brenner, Tal Furman Shohet, and Eran Treister. 2025. Zero-shot detection of llm-generated code via approximated task conditioning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 187–204. Springer.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. ArXiv:2309.16609 [cs].

Tamas Bisztray, Bilel Cherif, Richard A Dubniczky, Nils Gruschka, Bertalan Borsos, Mohamed Amine Ferrag, Attila Kovacs, Vasileios Mavroeidis, and Norbert Tihanyi. 2025. I know which llm wrote your code last summer: Llm generated code stylometry for authorship attribution. In *Proceedings of the 18th ACM Workshop on Artificial Intelligence and Security*, pages 28–39.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. ArXiv:2107.03374 [cs].

Atish Kumar Dipongkor, Ziyu Yao, and Kevin Moran. 2025. Reassessing code authorship attribution in the era of language models. *arXiv preprint arXiv:2506.17120*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Hanxi Guo, Siyuan Cheng, Kaiyuan Zhang, Guangyu Shen, and Xiangyu Zhang. 2025. Codemirage: A multi-lingual benchmark for detecting ai-generated and paraphrased source code from production-level llms. *arXiv preprint arXiv:2506.11059*.

Jiwoo Hong, Noah Lee, and James Thorne. 2024. ORPO: Monolithic Preference Optimization without Reference Model. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 11170–11189. Association for Computational Linguistics.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised Contrastive Learning. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2017. Focal Loss for Dense Object Detection. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2999–3007. IEEE Computer Society.

Scott Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. ArXiv:1705.07874 [cs].

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. *J. Syst. Softw.*, 214:112059.

Daniil Orel, Dilshod Azizov, and Preslav Nakov. 2025a. CoDet-M4: Detecting Machine-Generated Code in Multi-Lingual, Multi-Generator and Multi-Domain Settings. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, volume ACL 2025 of *Findings of ACL*, pages 10570–10593. Association for Computational Linguistics.

Daniil Orel, Dilshod Azizov, and Preslav Nakov. 2025b. Codet-m4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10570–10593.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026a. Aicd bench: A challenging benchmark for ai-generated code detection. *arXiv preprint arXiv:2602.02079*.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026b. 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. 2025c. Droid: A Resource Suite for AI-Generated Code Detection. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 31263–31289. Association for Computational Linguistics.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. 2025d. 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.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR*, abs/2308.12950. ArXiv: 2308.12950.

Yuling Shi, Hongyu Zhang, Chengcheng Wan, and Xiaodong Gu. 2025. Between lines of code: Unraveling the distinct patterns of machine and human programmers. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1628–1639. IEEE.

Llama Team. 2024. The Llama 3 Herd of Models. *CoRR*, abs/2407.21783. ArXiv: 2407.21783.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 1069–1088.

Yang Xianjun, Zhang Kexun, Chen Haifeng, Petzold Linda, Wang William Yang, and Cheng Wei. 2023. Zero-shot detection of machine-generated codes. *arXiv preprint arXiv:2310.05103*.

Tong Ye, Yangkai Du, Tengfei Ma, Lingfei Wu, Xuhong Zhang, Shouling Ji, and Wenhai Wang. 2025. Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 968–976.

Appendix

A Detailed description of handcrafted features

In this appendix, we provide precise definitions and calculation methodologies for the feature sets used in Subtask A.

A.1 Shallow Features

Shallow features capture surface-level stylometric patterns and are extracted using regex-based tokenization and simple string analysis. The complete list of shallow features is summarized in Table 8.

Indentation Entropy Calculation. To quantify the consistency of indentation, we calculate the entropy of leading space counts for non-empty lines:

$$H(\texttt{Indent}) = -\sum_{i \in I} p(i) \log p(i) \quad (3)$$

where I is the set of unique indentation lengths (e.g., 2 spaces, 4 spaces), and $p(i)$ is the probability (frequency) of indentation length i in the snippet.

A.2 AST Features

To enable language-specific AST parsing, we first employ a lightweight Language Identification (LID) model comprising a TF-IDF vectorizer and a Linear Support Vector Machine (LinearSVC). The model achieves 94% validation accuracy, with misclassifications primarily confined to syntactically similar languages like C and C++. Subsequently, we utilize tree-sitter to parse the code into ASTs. Despite the occasional conflation of C and C++ by the LID model, we empirically observed zero parsing failures. This robustness is attributed to the high degree of grammatical backward compatibility (e.g., a C++ parser successfully processing C code), ensuring that structural metrics remain valid even under dialect misclassification. After parsing, we traverse the tree to extract structural metrics summarized in Table 6.

A.3 Logit Features

These features are derived from the output probability distributions of a frozen Large Language Model (e.g., CodeLlama-7B). Let the input token sequence be denoted as $x = (x_1, \ldots, x_T)$. The detailed definitions of the derived statistics are summarized in Table 7.

Feature Name	Description
Ast_total_nodes	Total number of nodes in the syntax tree.
Ast_unique_node_types	Number of distinct node types (e.g., `function_definition`, `identifier`).
Ast_node_type_entropy	Shannon entropy of the distribution of node types.
Ast_max_depth	Maximum depth of the tree from root to leaf.
Ast_avg_depth	Average depth of all nodes in the tree.
Ast_leaf_ratio	Ratio of leaf nodes to total nodes.
Ast_branching_mean	Average number of children per non-leaf node.
Ast_node_per_token	Ratio of Ast_total_nodes to the shallow token count.
Ast_if_loop_count	Counts of structural control nodes (parsed, no regex).
Ast_call_binary_expr	Ratio of function calls and binary expressions to total nodes.

Table 6: AST-Based Feature Definitions

Feature Name	Description
Mean_true_logprob	Average log-probability of the true token y_t given context $x_{<t}$: $\frac{1}{T}\sum_{t=1}^{T} \log P(y_t \mid x_{<t})$.
Mean_nll	Mean negative log-likelihood: $-\frac{1}{T}\sum_{t=1}^{T} \log P(y_t \mid x_{<t})$.
Std_true_logprob	Standard deviation of the true-token log-probabilities.
Entropy_topk_raw	Entropy of the Top-K ($K = 20$) probability distribution: $-\sum_{i \in \text{Top-}K} p_i \log p_i$.
Entropy_topk_norm	Entropy of the Top-K distribution normalized by Top-K mass.
Maxprob_mean	Average probability of the most likely token: $\frac{1}{T}\sum_{t=1}^{T} \max_i P(i \mid x_{<t})$.
Margin_mean	Average difference between Top-1 and Top-2 probabilities: $\frac{1}{T}\sum_{t=1}^{T}(p_t^{(1)} - p_t^{(2)})$.
Topk_mass_mean	Average cumulative probability of the Top-K tokens: $\frac{1}{T}\sum_{t=1}^{T}\sum_{i \in \text{Top-}K} p_i$.
True_rank_mean	Average rank of the true token y_t in the sorted vocabulary.
Miss_rate	Fraction of tokens where the true token y_t is not in the Top-K predictions.

Table 7: LLM Probability-Based Feature Definitions

Category	Feature Name	Description
Counts	Num_chars	Total number of characters in the code.
	Num_lines	Total number of lines (N_{lines}).
	Num_tokens	Total number of tokens (N_{tokens}) extracted via regex `\w+`.
Ratios	Blank_line_ratio	Count of empty lines / N_{lines}.
	Comment_ratio	Count of comment patterns (//,#,/*) / N_{lines}.
	Numeric_literal_ratio	Count of numeric literals / N_{tokens}.
	String_literal_ratio	Count of string literals / N_{tokens}.
	Operator_ratio	Count of operators (e.g., +, -, *, =) / N_{tokens}.
	Keyword_ratio	Count of reserved keywords / N_{tokens}.
	Vocab_ratio	Vocabulary size (unique tokens) / N_{tokens}.
	Avg_line_length	$N_{\text{chars}}/N_{\text{lines}}$.
	Max_line_length	Length of the longest line in the snippet.
	Indent_entropy	Shannon entropy of indentation levels (see Eq. 1).
	Max_brace_depth	Maximum nesting depth of { or }.
Control Flow	Cnt_if, Cnt_for, Cnt_while	Raw counts of `if`, `for`, `while`.
	Cnt_return, Cnt_break, Cnt_continue	Raw counts of `return`, `break`, `continue`.
	Cnt_continue	Raw count of `continue` statements.

Table 8: Shallow Feature Definitions