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

Sabrina Bonfante
Politecnico di Torino
Computer Engineering
s343496

Edoardo Cotto
Politecnico di Torino
Computer Engineering
s343449

Iulian Lucian Andro
Politecnico di Torino
Computer Engineering
s346481

Francesco Papini
Politecnico di Torino
Computer Engineering
s336426

GitHub repository: https://github.com/papo1011/semeval2026

Abstract

The rapid advancement of large language models for code generation has made distinguishing machine-generated code from human-written code increasingly challenging. SemEval-2026 Task 13 addresses this problem through multiple subtasks involving binary detection, authorship attribution, and hybrid code identification across different programming languages and domains. In this paper, we present our approach for the different subtasks:

Subtask A: *We adopt a zero-shot Fast-DetectGPT approach based on analytic sampling discrepancy. Extending the original single-threshold formulation, we analyze both single and double threshold strategies to better capture the shape of discrepancy score distributions across domains. The best configuration uses Stable-Code-3B with a single threshold, achieving a Macro F1-score of 0.72.*

Subtask B: *We investigate a transformer-based approach for multi-class code authorship detection. Our system fine-tunes UniXCoder, a pre-trained model for source code understanding, to distinguish human-written code from code generated by ten different large language model families. To mitigate the severe class imbalance of the dataset, we employ class-weighted Focal Loss during training. Our approach achieves a Macro F1-score of 0.372 and an accuracy of 67.70%.*

Subtask C: *Addresses the complex challenge of hybrid and adversarial code detection. We formulated this as a 4-class sequence classification problem, fine-tuning the pre-trained UniXcoder model. To effectively manage the varying distribution of human, machine, hybrid, and adversarial samples, our system incorporates a dynamically class-weighted Focal Loss function. Evaluated on the local test set, the proposed approach achieves an Macro F1-score of 0.84 and Accuracy of 88.70%, demonstrating a strong capability to isolate fine-grained generation and obfuscation* artifacts.

1 Introduction

Recent advances in generative artificial intelligence have transformed software development practices. Large Language Models (LLMs) for code generation are now capable of producing syntactically correct and semantically meaningful source code across multiple programming languages and domains, and are increasingly integrated into real-world development environments through tools such as GitHub Copilot and ChatGPT.

The widespread adoption of these systems raises important challenges related to authorship attribution, academic integrity, plagiarism detection, software security, and trustworthiness. As machine-generated code becomes increasingly similar to human-written code, automatically identifying its origin becomes a challenging research problem.

SemEval-2026 Task 13 addresses this challenge through three complementary subtasks designed to evaluate the robustness and generalization capabilities of machine-generated code detection systems.

Subtask A focuses on binary machine-generated code detection, where the goal is to classify a code snippet as either human-written or fully machine-generated. The task evaluates system generalization across both seen and unseen programming languages and domains.

Subtask B extends the problem to multi-class authorship attribution. Instead of simply detecting whether code is machine-generated, systems must identify the specific family of large language models responsible for generating the code, including models such as DeepSeek, Qwen, OpenAI, Meta-LLaMA, and Mistral.

Subtask C further increases the complexity of the task by introducing hybrid and adversarial code

samples. In this setting, systems must distinguish among human-written, machine-generated, hybrid, and adversarially generated code snippets designed to mimic human coding styles.

2 Background

2.1 Subtask A

Zero-shot detectors distinguish machine-generated text using statistical signals from language models. DetectGPT (Mitchell et al., 2023) introduced probability curvature, but relies on costly perturbation sampling. Fast-DetectGPT (Bao et al., 2024) makes this process more efficient through analytic sampling discrepancy based on conditional probability curvature. We adapt this approach to source code and extend it by comparing single and double threshold decision rules across different code domains.

2.2 Subtask B and C

Among these models, UniXCoder (Guo et al., 2022) has demonstrated strong performance across multiple code understanding tasks by jointly modeling programming languages and natural language descriptions. Given its ability to capture semantic information beyond lexical patterns, UniXCoder represents a suitable choice for the multi-class authorship detection problem addressed in Subtask B and C.

3 System overview

3.1 Dataset

We used the official SemEval-2026 Task 13 dataset provided through Hugging Face (Orel et al., 2025a,b). The dataset is divided into three subtasks corresponding to different machine-generated code detection scenarios.

Each sample in the dataset contains the following fields: `code` (source code snippet), `generator` (source model or author of the code), `label` (target class label), `language` (programming language of the snippet).

The dataset includes code written in multiple programming languages. Both human-written and machine-generated code samples are provided. Machine-generated samples originate from several Large Language Model families such as OpenAI, Qwen, DeepSeek, Meta-LLaMA, BigCode, Gemma, Mistral, and others.

3.1.1 Subtask A

Subtask A focuses on binary machine-generated code detection. The objective is to classify each code snippet as either: human-written, machine-generated.

The training set contains approximately 500K samples, while the validation set contains 100K samples. Training data is primarily composed of algorithmic programming problems written in Python, Java, and C++. Evaluation settings additionally include unseen programming languages and unseen domains such as research and production code.

3.1.2 Subtask B

Subtask B extends the task to multi-class authorship detection. Instead of binary classification, systems must identify the author category associated with each code snippet.

The dataset includes one human class together with multiple LLM families, including:

DeepSeek-AI, Qwen, 01-ai, BigCode, Gemma, Phi, Meta-LLaMA, IBM-Granite, Mistral, OpenAI.

The training split contains approximately 500K samples, while the validation split contains 100K samples.

3.1.3 Subtask C

Subtask C focuses on hybrid code detection. The objective is to classify each snippet into one of four categories: human-written, machine-generated, hybrid, adversarial.

Hybrid samples correspond to partially AI-generated code, while adversarial samples are intentionally designed to mimic human coding behavior.

The training set contains approximately 900K samples and the validation set contains 200K samples.

3.2 Architecture

3.2.1 Subtask A

We adopt a training-free zero-shot architecture based on Fast-DetectGPT (Bao et al., 2024). Each input code snippet is processed by a code-oriented LLM, which provides token-level probability distributions through its logits.

These probabilities are used to compute the analytic sampling discrepancy score, which can be interpreted as a normalized deviation of the observed sequence from the model's expected behavior. Specifically, the score compares the log-probability of the input sequence with the expected

log-probability under the model distribution, normalized by the corresponding standard deviation. In this sense, the discrepancy score measures how many standard deviations the observed sequence lies above or below the model's expected log-probability. Since the required mean and variance can be computed directly from the model probabilities, the architecture avoids explicit perturbation generation and additional sampling steps.

The final prediction is obtained through a threshold-based decision rule.

3.2.2 Subtask B

Our approach fine-tunes UniXCoder (Guo et al., 2022) for multi-class code authorship classification. Each source code snippet is tokenized using the official UniXCoder tokenizer and encoded by the transformer backbone. The resulting sequence representation is processed by a classification layer that assigns the input to one of eleven classes, corresponding to human-written code or one of ten LLM families.

Given the highly imbalanced class distribution of the dataset, we replace the standard cross-entropy objective with a class-weighted Focal Loss (Lin et al., 2018), which increases the contribution of minority classes during training. Model selection is performed using the validation Macro F1-score, while early stopping is applied to reduce overfitting.

The prediction pipeline can be summarized as follows:

Code Snippet → UniXCoder Tokenizer → UniXCoder Encoder → Classification Head → Author Label

3.2.3 Subtask C

Similar to Subtask B, we formulate Subtask C as a 4-class sequence classification problem using the UniXCoder (Guo et al., 2022) model. The architecture processes tokenized code snippets through the transformer backbone, passing the resulting sequence representations to a classification head configured to predict one of four categories: human, machine, hybrid, or adversarial. To handle the disparate sample distributions across these four classes, we also integrate a dynamically class-weighted Focal Loss (Lin et al., 2018) function into the architecture.

The prediction pipeline follows the same structure of Subtask B.

3.3 Methodology

3.3.1 Subtask A

We first compute a discrepancy score for each sample using the analytic sampling discrepancy formulation. Threshold calibration is then performed on a balanced subset of the training set, containing the same number of human-written and machine-generated samples. The decision thresholds are selected by maximizing Macro F1-score, which is the main evaluation metric of the task.

We compare two thresholding strategies:

- **Single threshold:** this strategy follows the original Fast-DetectGPT (Bao et al., 2024) formulation. A snippet is classified as machine-generated when its discrepancy score exceeds a single calibrated threshold.
- **Double threshold:** this strategy extends the original formulation by jointly optimizing a lower and an upper threshold. A snippet is classified as machine-generated when its score falls outside the human-typical region, either below the lower threshold or above the upper threshold.

$$\text{Predic}(x) = \begin{cases} 1 \text{ (AI)} & \text{if } Z(x) > \tau_{\text{upper}} \text{ or} \\ & Z(x) < \tau_{\text{lower}}, \\ 0 \text{ (Human)} & \text{otherwise.} \end{cases}$$

Finally, we analyze the resulting score distributions to compare the behavior of the two thresholding strategies across the seen algorithmic training domain and the official mixed test setting.

3.3.2 Subtask B

Code snippets are tokenized using the UniXCoder (Guo et al., 2022) tokenizer and padded or truncated to a maximum sequence length of 512 tokens. The model is fine-tuned for multi-class classification, where each sample is assigned to one of eleven classes corresponding to human-written code or one of ten LLM families. Training is performed for 2 epochs using a batch size of 8 and mixed precision (FP16) when GPU resources are available. We optimize the network using a learning rate of 2×10^{-5} and a weight decay of 0.01.

To address the severe class imbalance present in the dataset, balanced class weights are computed from the training distribution and incorporated into a Focal Loss (Lin et al., 2018) objective with a focusing parameter of $\gamma = 2.0$. Early stopping with a

patience of 2 evaluation rounds is employed during training, monitoring the validation Macro F1-score to select the best-performing model checkpoint.

3.3.3 Subtask C

Code snippets are tokenized using the UniXcoder (Guo et al., 2022) tokenizer and padded or truncated to a maximum sequence length of 512 tokens. The model is fine-tuned for 3 epochs using a batch size of 64 and mixed precision (FP16) to maximize memory and training efficiency. We optimize the network utilizing a learning rate of 2×10^{-5} and a weight decay of 0.01.

To explicitly manage the class imbalance present in the dataset, we compute balanced class weights directly from the training distribution and apply them to a Focal Loss (Lin et al., 2018) function configured with a focusing parameter of $\gamma = 2.0$. Early stopping with a patience of 2 epochs is employed during training, continuously monitoring the validation Macro F1-score to save and select the optimal model checkpoint.

3.4 Model Selections

3.4.1 Subtask A

Model	Macro F1
Stable-Code-3B	0.72
Starcoder2-7B	0.71
CodeLlama-7B-hf	0.70
Starcoder2-3B	0.70
CodeLlama-7B-Instruct-hf	0.68
CodeLlama-7B-Python-hf	0.65
Qwen2.5-Coder-3B	0.64
Qwen2.5-Coder-7B	0.60
Qwen2.5-Coder-1.5B	0.60
Yi-Coder-9B-Chat	0.58
Yi-Coder-9B	0.58
DeepSeek-Coder-1.3B-Base	0.55
Qwen2.5-Coder-3B-Instruct	0.54
Qwen2.5-Coder-7B-Instruct	0.54
DeepSeek-Coder-6.7B-Base	0.51
DeepSeek-Coder-1.3B-Instruct	0.50
DeepSeek-Coder-6.7B-Instruct	0.50

Table 1: Model selection F1 results for zero-shot Fast-DetectGPT on Task A

Model selection was guided by the Fast-DetectGPT setting (Bao et al., 2024), where the scoring model directly affects the quality of the discrepancy score. Following the empirical observations of the original paper, we focused mainly on small language models, especially in the 1B–7B parameter range. This choice is also motivated by practical constraints, as smaller models reduce inference cost and GPU memory requirements while still providing effective probability estimates for detection.

We evaluated both models related to generator families present in the dataset and external code models not explicitly used as dataset generators. Interestingly, the best result was achieved by stable-code-3b, an external model, suggesting that generator-family overlap is not directly correlated with scoring effectiveness. We also did not observe a consistent relationship between model size and detection performance: StarCoder2-7B outperforms StarCoder2-3B, while Qwen2.5-Coder shows the opposite trend, with the 3B model performing better than the 7B variant. Overall, these results indicate that scoring effectiveness depends more on the shape and calibration of the model probability distribution over code than on size or generator-family overlap alone.

3.4.2 Subtask B and C

UniXCoder (Guo et al., 2022) was selected as the backbone model due to its strong performance on a variety of source code understanding tasks. Unlike lexical feature-based approaches, transformer architectures can capture contextual and semantic information within source code, enabling more effective modeling of programming patterns and coding styles.

The model was initialized from the publicly available `microsoft/unixcoder-base`[1] model and subsequently fine-tuned on the SemEval-2026 Task 13 dataset. To improve robustness against the highly imbalanced class distribution in Task B, we adopted a class-weighted Focal Loss (Lin et al., 2018) of the standard cross-entropy objective.

This combination allows the model to leverage rich code representations while improving learning for underrepresented author classes.

3.5 Validation Strategy

Following the shared task guidelines, Macro F1-score was adopted as the primary evaluation metric, as it equally weights all classes regardless of their frequency. In addition to Macro F1-score, we report accuracy, precision, and recall to provide a

[1] https://huggingface.co/microsoft/unixcoder-base

more comprehensive assessment of model performance.

4 Experimental results

4.1 Subtask A

Table 1 reports the performance of the evaluated code-oriented language models using the Fast-DetectGPT (Bao et al., 2024) discrepancy score. The best result is obtained by `stable-code-3b`, which achieves a Macro F1-score of 0.7219.

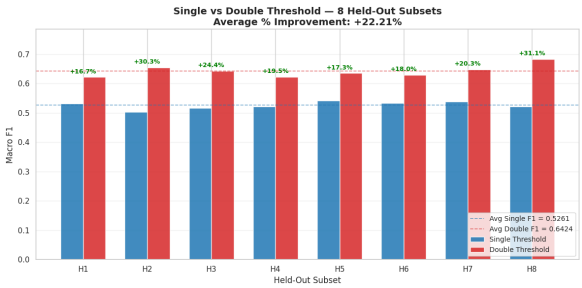

Figure 1: Comparison between single and double threshold strategies on the LeetCode training domain.

Training-domain analysis: LeetCode. We first analyze the thresholding strategies within the same domain used for calibration, namely the LeetCode-style algorithmic code domain. To make the comparison more stable, thresholds are fitted on a balanced training subset and then evaluated on multiple disjoint held-out subsets from the same training distribution. This allows us to measure whether the double-threshold strategy consistently improves over the original single-threshold rule when no domain shift is introduced.

In this setting, the double-threshold strategy clearly outperforms the single-threshold formulation. On average, the single threshold reaches a Macro F1-score of 0.5261, while the double threshold reaches 0.6424. This corresponds to an average absolute improvement of 0.1163 Macro F1 points, or a relative improvement of 22.21%.

Table 2: Average single and double threshold comparison on the LeetCode training domain.

Strategy	Macro F1	Delta	Improvement
Single threshold	0.53	–	–
Double threshold	0.64	+0.12	+22.21%

Figure 2 explains why the double threshold is effective in this setting. In the LeetCode domain, human-written code is mostly concentrated

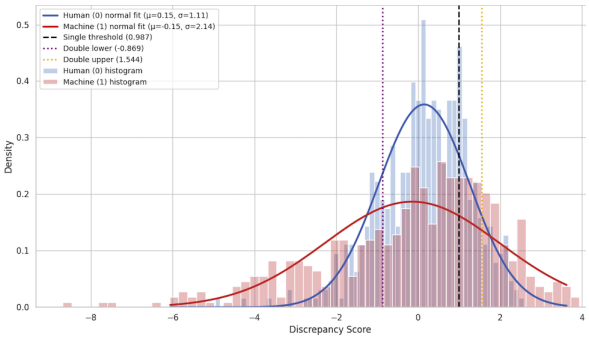

Figure 2: Double-threshold decision strategy on the LeetCode training domain using Stable-Code-3B.

in a central score region, while machine-generated code has a wider distribution with relevant mass in both tails. The upper tail captures highly predictable generated code, which is already detected by the single-threshold rule. However, the lower tail contains additional machine-generated samples with unusually low discrepancy scores. Since this lower tail is still relatively separated from the human distribution in the training domain, adding a lower threshold allows the detector to recover more machine-generated samples without introducing many false positives.

Official test-set analysis. We then evaluate the same thresholding strategies after switching to the official Task A test setting, which includes a broader and more heterogeneous code distribution. In this case, the behavior changes substantially. The single-threshold strategy achieves the best result, with a Macro F1-score of 0.7219, while the double-threshold strategy drops to 0.5799.

Table 3: Single and double threshold comparison on the official Task A test setting.

Strategy	Accuracy	Precision	Recall	Macro F1
Single threshold	0.82	0.75	0.71	0.72
Double threshold	0.64	0.59	0.62	0.58

The distribution in Figure 3 shows why the double threshold does not transfer well to the mixed test distribution. Unlike the LeetCode-domain setting, the lower tail is no longer a reliable indicator of machine-generated code. Human-written snippets from generic code domains can also receive very low discrepancy scores, producing a much stronger overlap with the machine-generated distribution. As a consequence, the lower threshold captures some additional generated samples, but it also incorrectly classifies many human-written

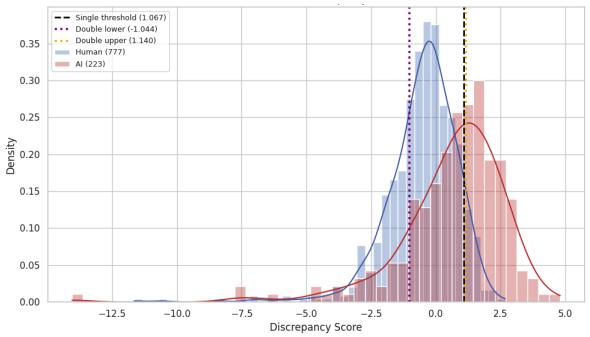

Figure 3: Single-threshold decision strategy on the official generic code test set using Stable-Code-3B.

snippets as machine-generated.

This comparison highlights that the double-threshold rule is effective when the score distributions preserve a clear separation between the central human region and both machine-code tails. However, under domain shift, the lower tail becomes less discriminative because human-written code also appears in extreme negative-score regions. For this reason, the single-threshold strategy is more robust in the official test setting: it ignores the unstable lower tail and relies only on the upper discrepancy region, leading to the best overall Macro F1-score.

4.2 Subtask B

Table 4 reports the overall performance of our approach on Subtask B. The proposed UniXCoder-based achieves a Macro F1-score of 0.372, which is the official evaluation metric of the shared task, together with an overall accuracy of 67.7%.

Table 4: Performance on Subtask B.

Metric	Score
Accuracy	0.6770
Macro Precision	0.3702
Macro Recall	0.4012
Macro F1	0.3718
Weighted F1	0.6980

The results for individual classes are reported in Table 5. The model performs particularly well on the *human* and *openai* classes, achieving F1-scores of 0.949 and 0.760, respectively. Among the remaining LLM families, *gemma* and *01-ai* obtain the strongest results, with F1-scores of 0.481 and 0.423. In contrast, performance on minority classes such as *deepseek*, *mistral*, and *ibm-granite* remains considerably lower, highlighting the difficulty of distinguishing between less represented generator families.

Table 5: Per-class performance on the Task B test set.

Class	Precision	Recall	Macro F1	Support
Human	0.9734	0.9262	0.9492	474
DeepSeek	0.0545	0.1429	0.0789	21
Qwen	0.2644	0.3151	0.2875	73
01-ai	0.3548	0.5238	0.4231	21
BigCode	0.2308	0.3000	0.2609	10
Gemma	0.4419	0.5278	0.4810	36
Phi	0.3056	0.2037	0.2444	54
Meta-LLaMA	0.3143	0.1803	0.2292	61
IBM-Granite	0.1522	0.3889	0.2188	18
Mistral	0.1212	0.2222	0.1569	18
OpenAI	0.8588	0.6822	0.7604	214
Macro Avg	0.3702	0.4012	0.3718	1000
Weighted Avg	0.7319	0.6770	0.6980	1000

Figure 4 shows the normalized confusion matrix. While human-written code is correctly recognized in most cases, substantial confusion is observed among several LLM families. In particular, Qwen samples are frequently misclassified as Phi and OpenAI, whereas Meta-LLaMA exhibits notable overlap with Qwen, IBM-Granite, and Mistral. Similar confusion patterns can also be observed between IBM-Granite and Mistral. These results suggest that different LLM families often generate code with highly similar stylistic and structural characteristics, making fine-grained authorship attribution significantly more challenging than distinguishing between human-written and machine-generated code.

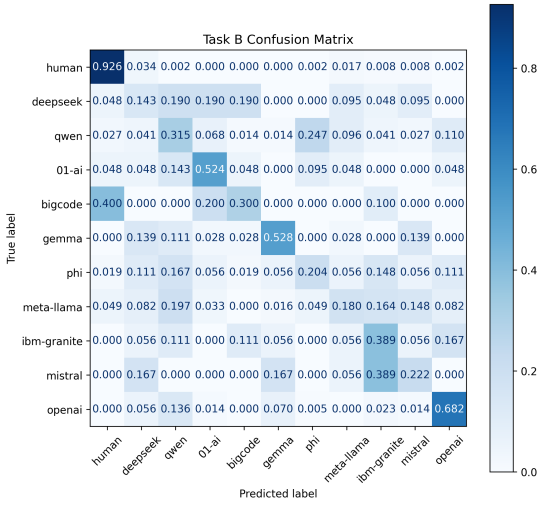

Figure 4: Confusion matrix obtained on the Subtask B test set.

Overall, the results indicate that the proposed approach effectively separates human and machine-generated code, but struggles to reliably discrim-

inate among closely related LLM families, especially in the presence of severe class imbalance.

4.3 Subtask C

Table 6 reports the overall and per-class performance of our fine-tuned UniXcoder model on the Subtask C test set. The proposed system achieved a strong overall Macro F1-score of 0.84 and a Accuracy of 88.70%, demonstrating robust capability to identify complex generation artifacts even in the presence of code modifications.

Table 6: Overall and per-class performance on the Subtask C test set.

Class	Precision	Recall	Macro F1	Support
Human	0.9865	0.9224	0.9534	554
Machine	0.8810	0.8114	0.8447	228
Hybrid	0.7115	0.8810	0.7872	84
Adversarial	0.6964	0.8731	0.7748	134
Accuracy			0.8870	1000
Macro Avg	0.8189	0.8720	0.8400	1000
Weighted Avg	0.9005	0.8870	0.8907	1000

Performance was highest on the *human* class (0.953 F1-score), followed by fully *machine-generated* code (0.845 F1-score). Despite the inherent complexity of detecting partial alterations, the model reliably identified both *hybrid* and *adversarial* samples, achieving F1-scores of 0.787 and 0.775, respectively. Notably, the high recall for both hybrid (0.881) and adversarial (0.873) categories indicates that the class-weighted Focal Loss successfully prevented the network from ignoring these underrepresented, complex categories.

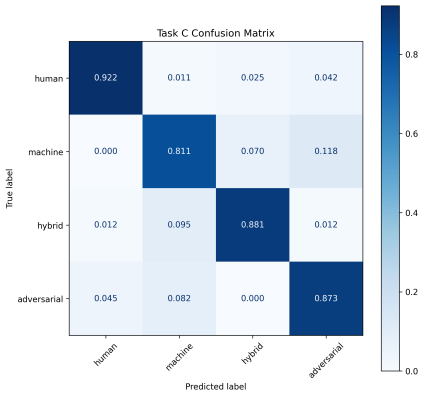

Figure 5: Normalized confusion matrix obtained on the Subtask C test set.

As illustrated in the normalized confusion matrix (Figure 5), human-written code is correctly identified in 92.2% of cases. However, the matrix reveals logical overlap among the synthetic categories. Fully machine-generated code is occasionally misclassified as adversarial (11.8%) or hybrid (7.0%). Conversely, hybrid samples show a 9.5% confusion rate with machine-generated code, and adversarial samples are sometimes mistaken for machine code (8.2%) or human code (4.5%). These patterns align with the intrinsic nature of the data, as adversarial and hybrid examples are intentionally engineered to mimic human stylometrics or blur attribution boundaries.

Overall, the results confirm that leveraging a deep semantic encoder like UniXcoder, paired with an class-weighted Focal Loss, provides a highly effective mechanism for isolating fine-grained generation and adversarial artifacts in source code.

5 Conclusion

This work presented complementary approaches for machine-generated code detection and authorship attribution across the SemEval-2026 Task 13 subtasks. For Subtask A, we adapted Fast-DetectGPT to source code and analyzed how different thresholding strategies behave across code domains. Our main finding is that threshold selection strongly depends on the target distribution. In the algorithmic LeetCode-style domain, the double-threshold strategy introduced in our work is preferable, since machine-generated code exhibits discriminative mass in both the upper and lower tails of the discrepancy-score distribution. In contrast, when moving to more generic code from production or research-oriented domains, the lower tail becomes less reliable and overlaps more with human-written samples. In this setting, the original single-threshold strategy provides a more robust decision boundary and achieves more consistent performance.

For Subtasks B and C, we relied on transformer-based classification models. By fine-tuning UniXCoder and incorporating class-weighted Focal Loss, our systems were able to learn discriminative representations of source code while partially mitigating the impact of class imbalance.

The experimental results highlight a clear difference in task difficulty. Subtask C demonstrates that transformer-based code representations are highly effective for distinguishing broad categories of code provenance, including human-written, machine-generated, hybrid, and adversarial samples. The confusion matrix shows a strong separation among the four classes, indicating that

UniXCoder captures high-level authorship and generation characteristics even in the presence of adversarial modifications.

In contrast, Subtask B proved substantially more challenging. While the model successfully identified human-written code and some generator families, performance decreased when distinguishing among individual LLM families. The confusion matrix revealed significant overlap between several generators, suggesting that different LLMs often produce code with highly similar stylistic and structural patterns. This observation highlights the increased complexity of fine-grained authorship attribution compared to broader source classification, suggesting a substantial stylistic overlap among modern LLM families.

Overall, our findings indicate that transformer-based code representations are effective for general code provenance analysis, whereas attribution to specific model families remains an open challenge. A major limitation of our approach is the severe class imbalance present in the training data, which negatively affects minority classes despite the use of class-weighted Focal Loss. Future work could explore data augmentation, contrastive learning, ensemble methods, or larger code-oriented language models to improve robustness and increase the separation between closely related LLM families.

References

Guangsheng Bao, Yanbin Zhao, Zhiyang Teng, Linyi Yang, and Yue Zhang. 2024. Fast-detectgpt: Efficient zero-shot detection of machine-generated text via conditional probability curvature. *Preprint*, arXiv:2310.05130.

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

Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal loss for dense object detection. *Preprint*, arXiv:1708.02002.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. *Preprint*, arXiv:2301.11305.

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*, pages 10570–10593, Vienna, Austria. Association for Computational Linguistics.

Daniil Orel, Indraneil Paul, Iryna Gurevych, and Preslav Nakov. 2025b. `Droid`: A resource suite for ai-generated code detection. *Preprint*, arXiv:2507.10583.