MALTO at SemEval-2026 Task 13: Detecting Human, AI, and Hybrid Code via Hard Negative Mining and Curriculum-Driven Ensembles

Hüseyin Arda Arslan* **Evren Ayberk Munis*** **Timofei Khudonogov***
Mert Akgün* **Murat Beşli*** **Ayhan Meherrem***
Claudio Savelli† **Flavio Giobergia†**
Politecnico di Torino, Italy
* {firstname.lastname}@studenti.polito.it
† {firstname.lastname}@polito.it

Abstract

The rapid advancement of Large Language Models (LLMs) has significantly impacted software engineering, posing challenges for determining the origin and authenticity of source code. This paper presents the MALTO team's submission for SemEval-2026 Task 13, explicitly focusing on Subtask B (Authorship Attribution among 11 classes) and Subtask C (Hybrid Code Detection). To address severe class imbalance and the complex boundaries of mixed human-machine code, we propose a unified framework that leverages an ensemble of UniXcoder and CodeT5. Our approach integrates a robust Tree-sitter-based Universal Canonicalization strategy, Data Augmentation, and a novel 3-Phase Curriculum Training schedule enhanced by Hard Negative Mining. Specifically, UniXcoder's cross-modal representations excel at distinguishing among semantically overlapping LLM families (Subtask B), whereas CodeT5's identifier-aware architecture is superior at detecting subtle structural anomalies in hybrid and adversarial snippets (Subtask C). By aggregating these complementary strengths, our soft-voting ensemble overcomes the limitations of individual models, demonstrating strong robustness against imbalanced distributions and effectively discriminating between purely human, purely machine, hybrid, and adversarial code snippets.

1 Introduction

As Large Language Models (LLMs) like GPT-4, Codex (Chen et al., 2021), Code Llama (Rozière et al., 2023), and StarCoder (Li et al., 2023) become increasingly integrated into software development, the ability to trace the provenance of code has become a critical necessity. Distinguishing between code written entirely by humans, code generated by specific AI models, or a hybrid combination of both is essential for enforcing intellectual property rights, ensuring academic integrity, and preventing the injection of AI-generated vulnerabilities.

SemEval-2026 Task 13 (Orel et al., 2026a) challenges participants to build systems that can detect machine-generated code under diverse conditions by evaluating generalization to unseen languages, generator families, and code application scenarios. This setting is closely connected to the broader challenge of navigating the rapidly expanding LLM ecosystem, where incomplete or inconsistent metadata about model provenance, training data, task specialization, licensing, and evaluation practices can hinder transparent model comparison and reproducibility (Pierri et al., 2025). In the context of code authorship attribution, such opacity further motivates robust detection systems that do not rely solely on declared model metadata, but instead learn discriminative patterns directly from source-code artifacts. The primary challenges in these tasks include extreme class imbalance, where minority AI generators are vastly underrepresented, and the subtle semantic overlap in Subtask C, where hybrid or adversarial codes masquerade as human-written logic. To address these issues, we introduce a unified, end-to-end framework[1]. Rather than relying solely on standard fine-tuning, our methodology introduces several key contributions:

- **Hard Negative Mining Strategy:** We isolate and up-weight the top 20% highest-loss training samples, forcing the models to focus on the most ambiguous decision boundaries.
- **3-Phase Curriculum Training:** A scheduled training regimen that progressively shifts from a heavily weighted distribution targeting hard samples to a natural data distribution, concluding with full combined dataset training.
- **Universal Canonicalization & Augmentation:** Utilizing abstract syntax trees (ASTs)

[1]The source code can be found at https://github.com/MALTO/machine-generated-code for reproducibility.

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2413–2420

via Tree-sitter, we normalize identifiers, flatten layouts, and aggressively augment hard samples to ensure models learn semantic intent rather than stylistic artifacts.

- **Custom Pooling & Ensembling:** We implement a custom classification head concatenating mean and max pooling to capture both global context and local salient features, combined in a weighted ensemble of UniXcoder (Guo et al., 2022) and CodeT5 (Wang et al., 2021).

Ultimately, our proposed unified ensemble strategy demonstrates strong empirical performance, yielding a Macro-F1 score of 0.405 on Subtask B and 0.616 on Subtask C on the hidden test set.

2 Challenge Description

The SemEval-2026 Task 13 evaluates the robustness of models in detecting machine-generated code under diverse conditions.

The challenge encompasses multiple tracks evaluated on the AICD (Orel et al., 2026b) Benchmark. Our methodology is optimized for the following two domains:

- **Subtask B (Authorship Attribution):** An 11-class classification problem where the goal is to attribute a code snippet to either a Human author or one of 10 major LLM families. The dataset comprises 500K training samples, 100K validation samples, and is evaluated on a massive hidden test set of approximately 500K samples. The data distribution for this task is intrinsically long-tailed. While Human-written code dominates the dataset (442K samples), snippets from minority AI generators are severely underrepresented (e.g., 2K for Gemma, 2K for BigCode, 10K for OpenAI).

- **Subtask C (Fine-Grained Classification):** A 4-class structural task requiring the categorization of code into Human, Machine, Hybrid, and Adversarial. The dataset contains 900K training samples, 200K validation samples, and is similarly evaluated against a 500K hidden test set. It reflects severe real-world imbalances: purely Human-written (485K) and Machine-generated (210K) examples form the vast majority, while the nuanced Hybrid (85K) and Adversarial (118K) classes constitute the minority.

Given the severe class imbalances described above, relying on accuracy would disproportionately favor majority classes. Therefore, the official primary evaluation metric for all subtasks is the Macro-F1 score. This metric computes the F1 score for each class independently and averages them, treating all classes equally. It heavily penalizes models that fail to perform well for minority classes (e.g., Hybrid, Adversarial, or minority LLMs).

3 Related Work

Recent advancements in code representation learning emphasize that treating source code as simple sequence of tokens is insufficient for capturing complex semantic and behavioral properties (Guo et al., 2022). Multi-class authorship attribution requires models to go beyond surface-level patterns to identify specific LLM "fingerprints." Foundational bimodal models, such as CodeBERT (Feng et al., 2020), established the efficacy of joint pre-training on code and natural language. This was further refined by cross-modal architectures like UniXcoder (Guo et al., 2022), which map code into shared semantic spaces to prioritize functional intent over syntactic form, a process crucial for generalizing across diverse generator families and programming languages.

Detecting hybrid and adversarial code presents a distinct challenge due to the subtle semantic overlap between human logic and machine-refined snippets. Underrepresented minority classes often reside in ambiguous regions of the feature space. Identifier-aware architectures, notably CodeT5 (Wang et al., 2021), address this by explicitly modeling variable dependencies and structural invariants. Furthermore, techniques such as hard negative mining have proven essential in refining decision boundaries by forcing models to focus on high-loss, ambiguous instances during training.

To counteract stylistic noise and intentional obfuscation, code canonicalization has become a standard preprocessing step. By utilizing Abstract Syntax Trees (ASTs) via tools like Tree-sitter, researchers can normalize identifiers and flatten layouts. This structural normalization ensures that models remain robust against superficial formatting changes, focusing instead on the underlying algorithmic logic (Bui et al., 2021), which is foundational for the detection of both purely machine-generated and complex hybrid code.

4 Methodology

Our unified pipeline is designed to tackle both Subtask B and C. It comprises advanced data preprocessing, hard negative mining, a specialized 3-phase curriculum training loop, and a multi-model ensemble strategy as illustrated in Figure 1.

4.1 Data Preprocessing and Augmentation

To prevent models from relying on superficial shortcuts (e.g., idiosyncratic formatting or comments), we developed a Universal Canonicalization pipeline using *Tree-sitter*. For supported languages (e.g., Python, Java, C++, Go, PHP), the canonicalizer parses the code into an AST and applies the following transformations:

1. **Identifier Renaming:** Variables, functions, and class types are mapped to generic, sequentially numbered identifiers (e.g., `VAR_0`, `FUNC_1`, `TYPE_0`). Language-specific keywords are strictly preserved. While this renaming process destroys semantic naming clues (e.g., humans and AI often use different naming conventions), the resulting gain in structural robustness outweighs the loss of semantic heuristics.

2. **Layout Flattening:** All inline and block comments are stripped, and whitespace/indentation is strictly normalized depending on the target language.

This canonicalization fuels our Augmentation Pipeline. During preprocessing, training examples are probabilistically duplicated. A standard sample is augmented (replaced by its canonicalized version) with a baseline probability. However, if a sample is identified as a "hard negative", it is augmented with a much higher probability (up to 1.0), thereby significantly increasing the density of challenging representations in the training space.

4.2 Custom Model Architecture

To enhance feature extraction from the pre-trained transformers, we modify the default sequence classification head of both UniXcoder and CodeT5 (Wang et al., 2021). Instead of relying solely on the `[CLS]` token or on standard mean pooling, our custom architecture applies attention-mask-aware Mean Pooling and Max Pooling to the final hidden states. The resulting vectors are concatenated into a unified representation. This ensures the classifier captures both the global contextual meaning (Mean) and the most salient local syntactical triggers (Max) before passing through the final linear layers.

4.3 Hard Negative Mining

To address the overlaps between classes, such as distinguishing a fine-tuned BigCode snippet from Human code (Subtask B) or isolating Hybrid logic (Subtask C), we employ a pre-training Hard Negative Mining step. First, an initial model performs inference over the entire training dataset. We calculate the cross-entropy loss for each sample and extract the top 20% of indices with highest loss. These instances are flagged as *hard negatives*. In downstream tasks, particularly in Subtask C, this flag is used to linearly scale the base sample weights (e.g., applying a multiplier of $3.0\times$ to the already heightened base weights of Hybrid and Adversarial classes).

4.4 The 3-Phase Curriculum Training

To mitigate catastrophic overfitting on majority classes, we propose a 3-Phase Curriculum Strategy (Bengio et al., 2009) to strategically manipulate learning dynamics:

- **Phase 1 (Weighted Training - 2 Epochs):** We resample the augmented data with a weighted policy. Sample weights are dynamically assigned based on class rarity and the *hard negative* flag. During this phase, the overall validation Macro-F1 score typically remains suppressed, as the model is heavily penalized by the artificial distribution and forced to focus on complex, minority boundaries. Empirical observations indicated that extending this phase beyond two epochs led to catastrophic forgetting of the natural data distribution.

- **Phase 2 (Natural Distribution - 3 Epochs):** To counteract the artificial bias of Phase 1, the weighted sampling policy is lifted, and the model is fine-tuned on the true, natural data distribution. This phase acts as a crucial recalibration step. Once the penalty weights are lifted, the model experiences a massive and rapid surge in Macro-F1 performance, successfully mapping the robust feature representations it struggled to learn in Phase 1 to the actual real-world class boundaries. We keep track of both the *latest* and the *best* models obtained after this phase.

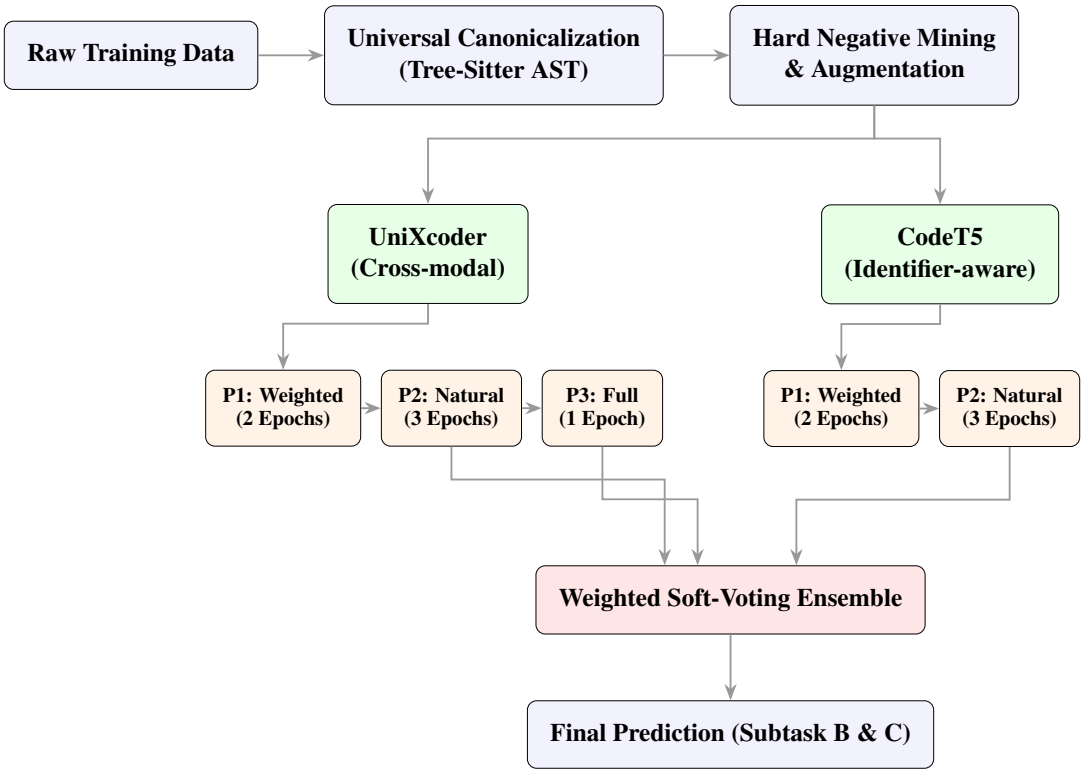

Figure 1: The proposed MALTO unified framework architecture. This diagram accurately maps the flow from code canonicalization to multi-phase temporal checkpointing, culminating in the weighted soft-voting ensemble mechanism.

- **Phase 3 (Full Training - 1 Epoch):** The training and validation datasets are concatenated. The model trains for a final, short epoch over this combined dataset to maximize data exposure, yielding a single *Phase 3 Final/Latest* checkpoint and its corresponding probability array. This phase is strictly limited to a single epoch to prevent overfitting on the concatenated data.

4.5 Ensemble Mechanism

Our final submission is powered by a weighted soft-voting ensemble mechanism that exploits both inter-model and intra-model variance. Rather than merely combining different architectures, we aggregate distinct temporal checkpoints from our training curriculum. For UniXcoder, we extract the softmax probability arrays generated across different phases, specifically, the highly calibrated *Phase 2 Best* checkpoints and the data-rich *Phase 3 Final/Latest* checkpoints. We exclude CodeT5 from Phase 3 because preliminary experiments showed early signs of overfitting on the concatenated data, making its *Phase 2 Best* checkpoint superior for generalization. This limitation also ensures computational temporal parity between the two models.

The final prediction is formulated by applying optimized scalar weights (determined empirically via grid search on the held-out validation set) to these multiple checkpoint arrays and taking the *argmax* of the summed probabilities. Specifically, for Subtask B, our predefined configuration predominantly favors UniXcoder (0.40 for Phase 2, 0.18 for Phase 3) combined with CodeT5 (0.42 for Phase 2). For Subtask C, a balanced 0.5/0.5 ensemble weighting between the models was utilized to capture structural and semantic nuances equally.

5 Experimental Setup

5.1 Datasets & Training Partitions

We utilize the official SemEval-2026 Task 13 datasets, which are built upon enriched resources and taxonomies from the Droid resource suite (Orel et al., 2025a) and the CoDet-M4 framework (Orel et al., 2025b). For our experimental setup, we strictly adhered to the official predefined data splits. During Phase 1 and Phase 2 of our curriculum, the models were exclusively trained on the official training set (500K for Subtask B, 900K for Subtask C) and evaluated continuously against the completely unseen official validation set (100K for

Subtask B, 200K for Subtask C) to monitor convergence and calibrate our final ensemble weights. In Phase 3, we concatenated the training and validation sets for one final epoch to maximize data exposure prior to inference.

5.2 Implementation Details & Computational Cost

Models were implemented using PyTorch and the HuggingFace `transformers` library. Training was conducted on single NVIDIA A100 GPUs using mixed precision (fp16) to accelerate computation. Common hyperparameters across both models and subtasks include a learning rate of $3 \cdot 10^{-5}$ (adjusted to $2 \cdot 10^{-5}$ for CodeT5-Large implementations) and a maximum sequence length of 512 tokens. To optimize memory usage, the effective batch size was maintained at 128 for UniXcoder and 32 for CodeT5 via gradient accumulation. Tree-sitter (v0.21.3) was utilized for all AST-based canonicalization processes.

The specific hyperparameter configurations, including the 20% threshold for hard negative mining and the (2-3-1) epoch distribution across curriculum phases, were empirically established to balance model convergence with strict computational budgets. The 20% cutoff was chosen to sufficiently increase the density of difficult samples without causing the model to overfit to outliers. Similarly, the phase lengths were constrained to prevent excessive training durations; Phase 1 (2 epochs) was sufficient to initially shift the decision boundaries, while Phase 3 was strictly limited to a single epoch to prevent catastrophic overfitting when training on the fully concatenated dataset.

Table 1 provides a breakdown of the computational cost and epoch distribution across the Curriculum for a single training run.

Model	**Phase 1** (2 Epochs)	**Phase 2** (3 Epochs)	**Phase 3** (1 Epoch)	**Est. Total Duration**
CodeT5	∼4.0h	∼6.0h	-	∼**10.0h**
UniXcoder	∼3.0h	∼4.5h	∼1.5h	∼**9.0h**

Table 1: Estimated training duration and epoch distribution per model on an NVIDIA A100 GPU.

6 Results and Analysis

6.1 Main Results

Our unified ensemble strategy yielded a Macro-F1 of **0.405** on Subtask B and **0.616** on Subtask C on the hidden test set.

Model	**Subtask B**	**Subtask C**
Baseline (CodeBERT (Feng et al., 2020))	0.229	0.481
CodeT5	0.387	0.600
UniXcoder	0.391	0.594
MALTO Ensemble	**0.405**	**0.616**

Table 2: Performance comparison illustrating the Macro-F1 score gains achieved by the final ensemble over individual standalone architectures and the official baseline.

As demonstrated in Table 2, the soft-voting ensemble mechanism effectively outperforms both the baseline models and the singular standalone architectures.

6.2 Model Isolation Analysis

Model Architecture Isolation: When evaluated independently, neither CodeT5 nor UniXcoder manages to capture the full spectrum of vulnerabilities inherent to AI-generated code. CodeT5's identifier-aware architecture heavily indexes on structural patterns, which makes it highly effective for isolating Hybrid logic (Subtask C), but it falls short in Subtask B, where different LLM families (e.g., Mistral vs. LLaMA) produce structurally identical but semantically distinct code. Conversely, UniXcoder captures these global semantic nuances highly effectively but is more easily bypassed by the deliberate, token-level structural obfuscations present in Adversarial code. The soft-voting ensemble succeeds by mutually masking these architectural blind spots, resulting in the final F1 score surge.

7 Conclusion

Identifying the true origin of modern source code is a highly challenging task, primarily due to severe class imbalances and deliberate obfuscation techniques found in hybrid and adversarial snippets. To address these vulnerabilities in the context of SemEval-2026 Task 13, we introduced a unified framework centered around a 3-Phase Curriculum Training schedule, robust AST-based canonicalization, and a structurally complementary soft-voting ensemble composed of UniXcoder and CodeT5.

Our methodology rapidly accelerated performance beyond standard fine-tuning constraints, yielding highly robust Macro-F1 scores of 0.405 for multi-class authorship attribution (Subtask B) and 0.616 for fine-grained hybrid classification (Subtask C), vastly outperforming the official baselines.

8 Limitations & Future Work

While our framework proved resilient, several limitations remain. First, systematic feature ablation studies are required to explicitly quantify the isolated impact of Phase 1 weighting versus the AST augmentation pipeline across distinct sub-tasks. Secondly, our methodology's reliance on Tree-sitter parsers creates a dependency on language-specific syntax definitions. To ensure zero-shot generalization to entirely unseen coding languages, future iterations must adapt the canonicalization pipeline to map code into language-agnostic Universal Abstract Syntax Trees (UAST) or Intermediate Representations (IR), similar to the structural abstractions proposed by Bui et al. (2021).

References

Orel, D., Azizov, D., Paul, I., Wang, Y., Gurevych, I., & Nakov, P. (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). Association for Computational Linguistics.

Orel, D., Paul, I., Gurevych, I., & Nakov, P. (2025). *Droid: A resource suite for ai-generated code detection*. In Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (pp. 31251-31277).

Orel, D., Azizov, D., & Nakov, P. (2025). *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 (pp. 10570-10593). Association for Computational Linguistics.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (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).

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (pp. 8696-8708).

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., & Jiang, D. (2020). *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. In Findings of the Association for Computational Linguistics: EMNLP 2020 (pp. 1536-1547).

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., ... & Wolf, T. (2023). *StarCoder: may the source be with you!*. arXiv preprint arXiv:2305.06161.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Zheng, X. E., ... & Synnaeve, G. (2023). *Code Llama: Open Foundation Models for Code*. arXiv preprint arXiv:2308.12950.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D., Kaplan, J., ... & Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv preprint arXiv:2107.03374.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., ... & Yin, J. (2021). *CodeXGLUE: A machine learning benchmark dataset for code understanding and generation*. arXiv preprint arXiv:2102.04664.

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). *Curriculum learning*. In Proceedings of the 26th annual international conference on machine learning (pp. 41-48).

Bui, N. D., Yu, Y., & Jiang, L. (2021). *InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees*. In Proceedings of the 43rd International Conference on Software Engineering (ICSE) (pp. 507-519).

Orel, D., Azizov, D., Paul, I., Wang, Y., Gurevych, I., & Nakov, P. (2026). *AICD Bench: A Challenging Benchmark for AI-Generated Code Detection*. In Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers) (EACL 2026). Association for Computational Linguistics, Rabat, Morocco.

Pierri, F., Bernasconi, A., Giobergia, F., Savelli, C., Baralis, E., Cagliero, L., & Ceri, S. (2025). *A Conceptual Map for Exploring the Landscape of Large Language Models*. IEEE Access.

A Detailed Class-Level Performance and Error Analysis

To provide deeper insights into the model's performance and address the subtle decision boundaries between generative families and hybrid structures, this section details the per-class metrics and confusion matrices evaluated on the validation sets.

A.1 Per-Class F1 Scores

Table 3 and Table 4 present the precision, recall, and F1-scores for Subtask B and Subtask C, respectively. The data highlights that while majority classes (e.g., Human-written code) achieve near-perfect recall, the minority classes benefit significantly from the ensemble's structural awareness.

Class	Precision	Recall	F1-score	Support
Human	0.9933	0.9942	0.9937	88,490
DeepSeek-AI	0.5716	0.5277	0.5488	847
Qwen	0.4256	0.4530	0.4389	1,755
01-ai	0.4198	0.4708	0.4438	650
BigCode	0.5767	0.6000	0.5881	445
Gemma	0.5749	0.7016	0.6320	372
Phi	0.6151	0.5546	0.5833	1,118
Meta-LLaMA	0.5385	0.4206	0.4723	1,695
IBM-Granite	0.7237	0.7201	0.7219	1,579
Mistral	0.4204	0.3955	0.4076	895
OpenAI	0.7169	0.7971	0.7549	2,154
Macro Avg	**0.5979**	**0.6032**	**0.5987**	**100,000**

Table 3: Classification report on the validation set for Subtask B.

Class	Precision	Recall	F1-score	Support
0 (Human)	0.9690	0.9843	0.9766	107,885
1 (Machine)	0.8657	0.8394	0.8523	46,770
2 (Hybrid)	0.8254	0.7931	0.8090	19,006
3 (Adversarial)	0.8218	0.8358	0.8287	26,339
Macro Avg	**0.8704**	**0.8632**	**0.8666**	**200,000**

Table 4: Classification report on the validation set for Subtask C.

A.2 Confusion Matrices

Figure 2 and Figure 3 visually illustrate the confusion matrices on the validation set for the given task. These matrices show the exact decision boundaries at which the ensemble excels and where certain generator families still exhibit semantic overlap.

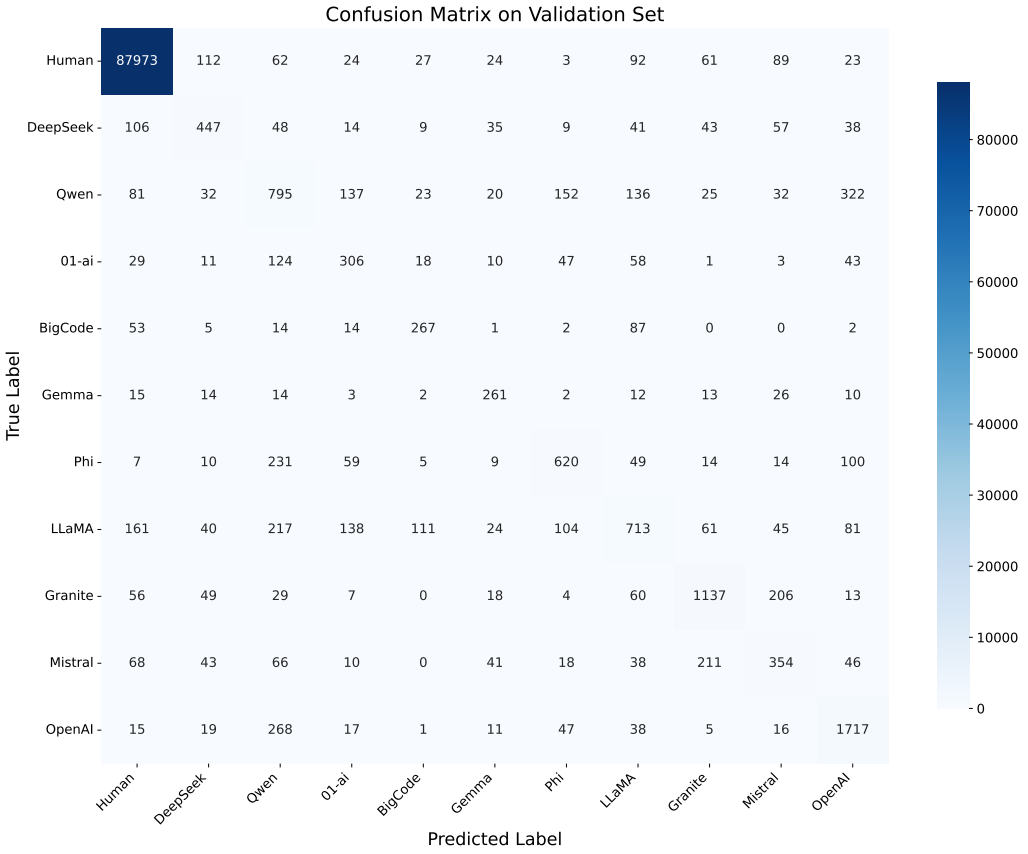

Figure 2: Confusion matrix evaluated on the Subtask B validation set.

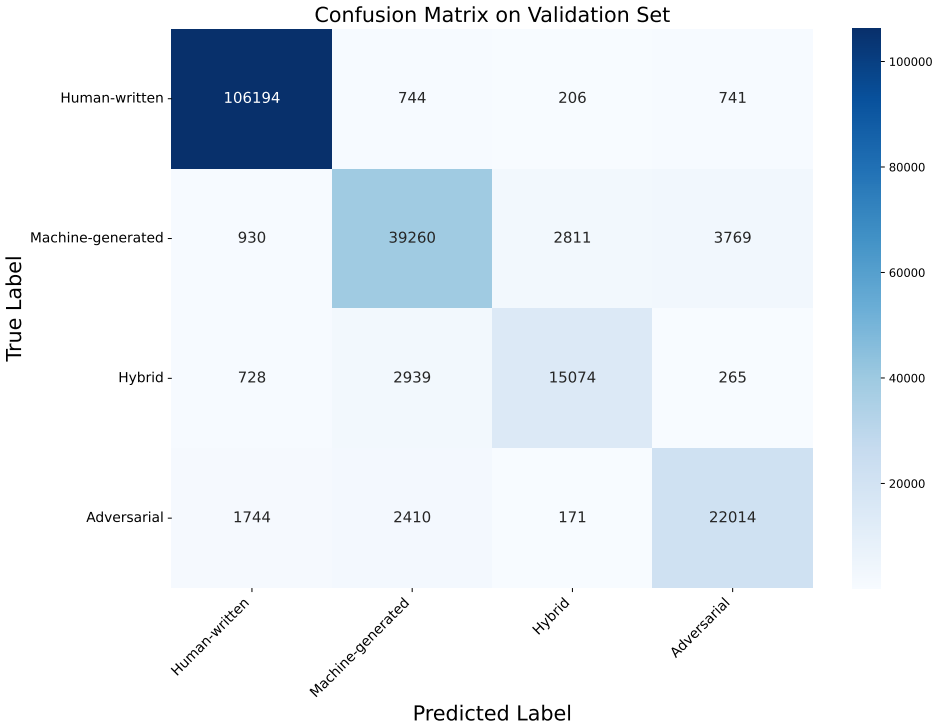

Figure 3: Confusion matrix evaluated on the Subtask C validation set.