ASTraNet at SemEval-2026 Task 13: Not All Code Looks the Same: Multi-View Structural and Semantic Detection of Machine-Generated Code

Ruwad Naswan*, **Dipit Saha***, **Md. Rafat Kabir***, **Nabiha Tahseen***
Bangladesh University of Engineering and Technology
{ruwad45678, sdipit099, rafatkabir054, ntahseen03}@gmail.com

Abstract

The growing adoption of large language models for code generation poses challenges for code quality, security, and authorship verification—particularly when test conditions involve unseen programming languages, generators, or application domains. We present our system, which combines three code-pretrained transformer encoders (CodeT5p-220M, CodeBERT, UniXcoder) with a structure-first Flow-Augmented AST (FA-AST) encoder implemented as a Gated Graph Neural Network. On Subtask A our best single model achieves macro F1 of 0.559; a post-competition layered rank-fusion ensemble across all three encoders raises this to 0.643. On Subtask C we obtain 0.585 officially; a three-stage ensemble combining neural probabilities with LightGBM-based features and class-priority routing raises this to 0.652. Our contributions include a language-agnostic structural detector, a diversity-driven rank-fusion strategy exploiting low inter-model correlation for binary classification, and a meta-learner stacking pipeline for multi-class detection under distribution shift.

1 Introduction

SemEval-2026 Task 13 (Orel et al., 2026b) presents a large-scale benchmark for detecting machine-generated code across diverse programming languages, generators, and application domains. It poses three main challenges: (1) *Distribution shift*—test data contain unseen languages (Go, PHP, C#, C, JavaScript) and unseen domains (research and production code) that never appear during training; (2) *Class complexity*—Subtask C introduces hybrid and adversarial categories alongside human and machine labels, requiring fine-grained distinctions; (3) *Structural diversity*—surface-level token cues that work well within a single language often fail to transfer across languages and domains (Orel et al., 2025b,a).

To address these challenges, we present a system that trains three code-pretrained encoders—CodeT5p-220M, CodeBERT, and UniXcoder—with attention pooling, contrastive learning, and focal loss, alongside a structure-first FA-AST encoder implemented as a GGNN. Our contributions are:

- A language-agnostic, structure-first detector based on canonicalized FA-ASTs and a GGNN that generalizes across programming languages by relying on structural topology rather than surface tokens.
- A layered rank-fusion ensemble for Subtask A that exploits low inter-model correlation (Spearman r=0.35–0.57) through diversity-weighted rank combination and cascading rescue layers, raising macro F1 from 0.559 to 0.643.
- A three-stage ensemble for Subtask C combining neural probabilities, LightGBM over StarCoder2-derived features, and class-priority routing, raising macro F1 from 0.585 to 0.652.

2 Background

Task Description. SemEval-2026 Task 13 (Orel et al., 2026b) provides large-scale datasets for three subtasks. Subtask A is a binary classification problem with 500K training samples (238K human-written, 262K machine-generated) drawn from algorithmic problem-solving contexts in C++, Python, and Java. Subtask C scales to 900K training samples and introduces two additional categories—*hybrid* code (partially written or completed by LLMs) and *adversarial* code (generated to mimic human style)—for a total of four classes. All subtasks are evaluated using macro F1, and test

* Equal contribution.

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 3209–3217

sets deliberately include unseen programming languages (Go, PHP, C#, C, JavaScript) and unseen domains (research and production code), making cross-lingual and cross-domain generalization the central challenge. Further details are described in Orel et al. (2026a).

Related Work. Detecting AI-generated text has been studied extensively for natural language, with approaches ranging from fine-tuned classifiers (Orel et al., 2025b) to zero-shot methods based on perplexity and statistical properties. For code specifically, Orel et al. (2025a) introduced the CoDet-M4 benchmark and demonstrated that cross-lingual and cross-domain generalization remain open problems even for strong transformer baselines. More recently, Xu et al. (2025) showed that contrastive learning on code representations can effectively distinguish LLM-generated code from human-written code. Stylometric and AST-based features have also been explored for code authorship attribution (Caliskan-Islam et al., 2015; Kalgutkar et al., 2019), though primarily in human-vs-human settings. We build on these ideas and adapt them to the human-vs-machine detection problem under distribution shift.

3 System Overview

Our approach builds on the observation that token-level and structure-level representations capture complementary aspects of code authorship. We train three code-pretrained encoders—CodeT5p-220M (Wang et al., 2023), CodeBERT (Feng et al., 2020), and UniXcoder (Guo et al., 2022)—using the same frozen-encoder pipeline with attention pooling, contrastive learning, and focal loss (Section 3.1). A Flow-Augmented AST (FA-AST) encoder implemented as a GGNN (Figure 1) operates on canonicalized syntax trees to capture structural signatures that persist across languages. For Subtask A we combine the three encoders via layered rank fusion (Section 3.4); for Subtask C we fuse neural and non-neural signals through meta-learner stacking (Section 3.5).

3.1 Transformer Encoders

Frozen-encoder pipeline. We train three code-pretrained encoders—CodeT5p-220M (Wang et al., 2023), CodeBERT (Feng et al., 2020), and UniXcoder (Guo et al., 2022)—using a shared architecture. Each input is tokenized and passed through the frozen encoder, producing token embeddings

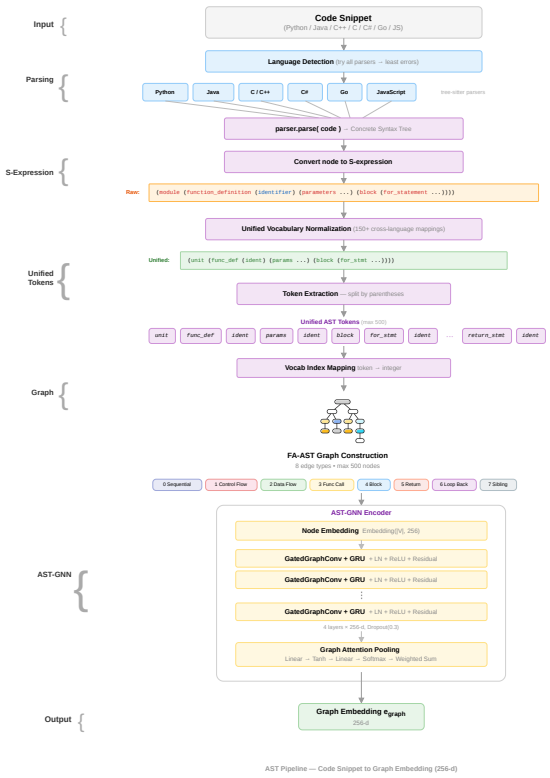

Figure 1: Flow-Augmented AST (FA-AST) Construction and GGNN-Based Structural Encoding Pipeline

$\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_n] \in R^{n \times d}$ where d=768. We apply attention pooling to obtain a fixed-length representation:

$$\alpha_i = \frac{\exp(\mathbf{w}^\top \mathbf{h}_i)}{\sum_{j=1}^{n} \exp(\mathbf{w}^\top \mathbf{h}_j)}, \quad \mathbf{r} = \sum_{i=1}^{n} \alpha_i \, \mathbf{h}_i \quad (1)$$

where $\mathbf{w} \in R^d$ is a learnable query vector. A projection head $g(\cdot)$ maps the pooled representation to $\mathbf{e} = g(\mathbf{r})$, trained with supervised contrastive loss to pull same-class embeddings together and push apart different classes. The classification head is trained with focal loss (γ=2) with per-class weighting, which down-weights easy examples and mitigates the class imbalance present in the training data. For Subtask A this pipeline is used standalone; for Subtask C we pass the predicted class probabilities to the ensemble stacking stage (Section 3.5). Figure 2 illustrates the complete encoder pipeline.

3.2 AST-Based Structural Features

A key observation motivating this component is that human-written and machine-generated code differ not only in surface tokens but also in structural patterns—such as nesting depth, expression

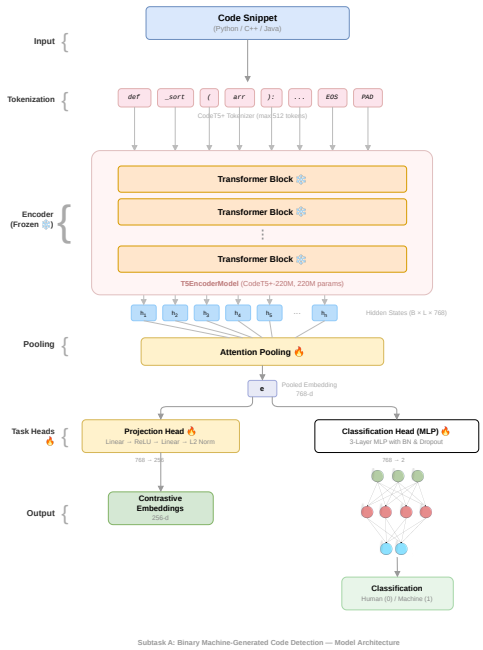

Figure 2: CodeT5p-220M Encoder Architecture with Attention Pooling and Contrastive Projection Head

complexity, identifier naming conventions, and comment density—that tend to persist across languages and domains.

For each code snippet, we parse the abstract syntax tree using Tree-sitter and extract a handcrafted metric vector $\mathbf{f} \in R^m$ comprising node-type distributions, nesting depth statistics, leaf ratios, identifier-length statistics, comment-to-code ratios, and operator/keyword frequencies. We quantify the discriminative power of each feature f_j using Cohen's d effect size between the human and machine class distributions:

$$d_j = \frac{\mu_j^{\text{machine}} - \mu_j^{\text{human}}}{s_j^{\text{pooled}}} \tag{2}$$

where s_j^{pooled} is the pooled standard deviation. Features with $|d_j| > 0.2$ were considered useful for model selection diagnostics (Section 5). Figure 3 shows the average discriminative power by metric category; MAGECODE Core (Pham et al., 2024) and GLTR Bucket (Gehrmann et al., 2019) features exhibit the largest effect sizes ($|d| > 0.37$), confirming that token-probability and rank-based signals are among the strongest discriminators between human and machine code. The metric vector is used as an auxiliary signal during analysis, while the FA-AST graph representation is encoded by the GGNN described below.

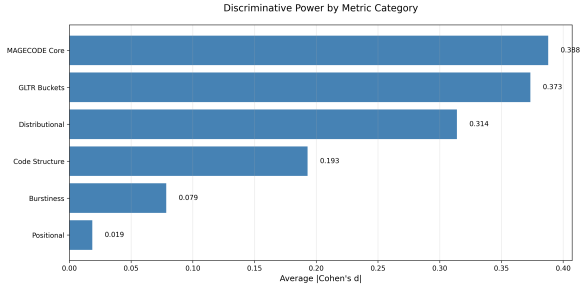

Figure 3: Effect-Size Analysis (Average |Cohen's d|) of Feature Categories for Human and Machine-Generated Code

3.3 Structure-first FA-AST Detector

This component is designed to capture topology-level differences between human-written and machine-generated code, intentionally decoupled from surface lexical signals to improve cross-language and cross-domain generalization.

As shown in Figure 1, the FA-AST pipeline augments syntax trees with flow-aware edges, yielding a graph that is then encoded with a four-layer GGNN.

Parsing and Error Handling. Following prior work on flow-augmented ASTs (Wang et al., 2020), we parse each snippet with Tree-sitter for seven languages (Python, Java, C, C++, C#, Go, JavaScript) and canonicalize node types into a shared vocabulary. Since code snippets in the dataset may be incomplete, ambiguous, or stylistically atypical, Tree-sitter occasionally produces *error nodes*—placeholder AST nodes inserted by the parser to recover gracefully from syntactic constructs it cannot fully resolve. To minimize their influence, we attempt parsing with all seven language grammars and retain the parse that yields the fewest error nodes, returning immediately if any parse is error-free. Error nodes are not discarded; they are mapped to a reserved `error` token in the shared vocabulary, preserving dataset coverage while attenuating their structural influence.

This strategy does not introduce systematic bias between human and machine classes. Parse failures stem from surface syntactic ambiguity—truncated snippets, mixed-language constructs, or unusual formatting—rather than from authorship signal. The `error` mapping therefore ensures that structurally corrupt subtrees contribute uniformly attenuated signal rather than class-specific noise.

Canonicalization and Cross-Language Consistency. After parsing, node types are mapped to a shared cross-language vocabulary of 137 unified labels covering 267 language-specific source types (e.g., `function_definition`, `method_declaration`, and `func_literal` all resolve to `func_def`). This canonicalization substantially reduces language-induced structural divergence: syntactically equivalent constructs across Python, Java, and C++ collapse to the same node type, exposing structural regularities that the GGNN can exploit. Residual cross-language differences—arising from genuinely distinct language semantics rather than surface notation—are further attenuated by the GGNN's message-passing scheme, which aggregates multi-hop structural context and is robust to local variation in node distribution. Additionally, node-level inputs are deliberately minimal—canonical node IDs, node degree, and coarse position indices—and identifier names are normalized to prevent lexical leakage, further insulating the model from parser-specific artifacts.

Graph Construction and Encoding. We then augment the AST with eight edge types—bidirectional parent–child links that let context flow both up and down the tree and seven unidirectional types (control-flow, data-flow, call/return, block, loop-back, and sibling) whose directions follow their natural semantics (e.g., control flows forward, returns flow backward, loop-backs re-enter headers). The resulting graph $G = (V, E)$ is encoded with a four-layer GGNN. At each layer t, node states are updated via gated message passing:

$$\mathbf{m}_v^{(t)} = \sum_{e \in \mathcal{E}} \sum_{u \in \mathcal{N}_e(v)} \mathbf{W}_e \, \mathbf{h}_u^{(t-1)} \tag{3}$$

$$\mathbf{h}_v^{(t)} = \mathrm{GRU}\Big(\mathbf{h}_v^{(t-1)},\ \mathbf{m}_v^{(t)}\Big) \tag{4}$$

where $\mathcal{E}$ is the set of edge types, $\mathcal{N}_e(v)$ are the neighbors of node v under edge type e, and $\mathbf{W}_e$ is an edge-type-specific weight matrix that learns to weight structural signals differently per relation (e.g., distinguishing control-flow context from parent–child hierarchies). The hidden state $\mathbf{h}_v^{(t)}$ encodes the structural neighborhood of node v up to t hops, accumulating topology-level context that is largely invariant to surface token identity. After four rounds of message passing, node embeddings are pooled via attention (Equation 1) and classified with an MLP head. This pipeline achieves a macro F1 of 0.493 on Subtask A.

3.4 Ensemble Strategy (Subtask A)

Post-competition, we trained CodeBERT and UniXcoder in the same frozen-encoder pipeline alongside CodeT5p-220M and built a five-layer cascading ensemble. The core intuition is simple: no single model is reliably correct near the decision boundary, so we use model agreement as a confidence signal—overriding cautious base predictions when independent models concur.

To quantify agreement, we convert each model's raw probability into a *percentile rank* within the test set, making scores comparable across models. The Spearman rank correlation between CodeT5p and UniXcoder (r=0.35) and between CodeT5p and CodeBERT (r=0.57) confirms that the models provide substantially independent signals—a prerequisite for effective ensembling.

The five layers progressively refine a single base prediction:

Layer 1 (Base prediction). We form a weighted rank score that favors CodeT5p (our strongest single model) while incorporating a small UniXcoder contribution:

$$\begin{aligned} r_c &= 0.85\, r_{\text{CodeT5p}} + 0.15\, r_{\text{UniXcoder}} \\ \hat{y} &= 1[r_c \geq \tau] \end{aligned} \tag{5}$$

where τ=0.843 corresponds to CodeT5p's empirically optimal probability threshold (F1 0.633).

Layer 2 (Near-threshold rescue). Predictions just below the threshold are uncertain by definition. We flip these to positive when both models independently agree, providing a second opinion on borderline cases:

$$\tau - 0.008 \leq r_c < \tau \ \wedge\ |r_{\text{CodeT5p}} - r_{\text{UniXcoder}}| \leq 0.25 \tag{6}$$

This recovers some false negatives, pushing F1 to 0.634.

Layer 3 (Consensus overrides). When both models are highly confident in the same direction, we trust them unconditionally, overriding any intermediate score:

$$\begin{cases} \hat{y} = 1 & \text{if } r_{\text{CodeT5p}} \geq 0.8 \ \wedge\ r_{\text{UniXcoder}} \geq 0.8 \\ \hat{y} = 0 & \text{if } r_{\text{CodeT5p}} \leq 0.2 \ \wedge\ r_{\text{UniXcoder}} \leq 0.2 \end{cases} \tag{7}$$

F1 rises to 0.635.

Layer 4 (UniXcoder rescue). Because UniXcoder correlates weakly with CodeT5p (r=0.35),

its high-confidence positive predictions carry independent evidential weight. We recover false negatives where $p_{\text{UniXcoder}} \geq 0.76$ and $p_{\text{CodeT5p}} \geq 0.70$, lifting F1 to 0.638.

Layer 5 (CodeBERT rescue). We apply an analogous rescue using CodeBERT, but set a higher probability bar ($p_{\text{CodeBERT}} \geq 0.88$) reflecting its stronger correlation with CodeT5p (r=0.57)—a correlated model must be more certain to add new information. Final F1: **0.643**, up from the CodeT5p-only baseline of 0.559.

3.5 Ensemble Strategy (Subtask C)

Post-competition, we designed a three-stage ensemble to address the four-class setting of Subtask C, which benefits from combining diverse model families.

Stage 1: Orthogonal feature classifier. We train a LightGBM on features the neural encoder does not explicitly model: TF-IDF character n-grams (n=2–4, 5K features), the same 32 handcrafted stylometric features defined in Section 3.2, and 40 token-level statistical features from StarCoder2-3B (Lozhkov et al., 2024)—including perplexity, log-probability distributions, GLTR rank buckets (Gehrmann et al., 2019), and per-token entropy. For each sample x, this yields a feature vector $\mathbf{x}_{\text{lgb}} = [\mathbf{x}_{\text{tfidf}};\ \mathbf{x}_{\text{hc}};\ \mathbf{x}_{\text{sc}}] \in R^{5072}$, from which LightGBM produces class probabilities $\mathbf{p}^{\text{lgb}} \in R^4$.

Stage 2: Meta-learner stacking. We stack the neural and LightGBM probability vectors into a meta-feature vector that captures both raw predictions and inter-model agreement:

$$\mathbf{m} = \left[\mathbf{p}^{\text{nn}};\ \mathbf{p}^{\text{lgb}};\ \hat{c};\ \mathbf{H};\ a;\ \boldsymbol{\delta}\right] \in R^{17} \quad (8)$$

where $\hat{c} = [\max_y p_y^{\text{nn}},\ \max_y p_y^{\text{lgb}}]$ captures each model's confidence, $\mathbf{H} = [-\sum_y p_y \log p_y]$ for each model measures prediction entropy, $a = 1[\arg\max \mathbf{p}^{\text{nn}} = \arg\max \mathbf{p}^{\text{lgb}}]$ indicates agreement, and $\boldsymbol{\delta} = \mathbf{p}^{\text{nn}} - \mathbf{p}^{\text{lgb}}$ captures per-class disagreement. A second LightGBM trained on $\mathbf{m}$ learns when to trust each base model, producing final probabilities $\mathbf{p}^{\text{meta}} \in R^4$.

Stage 3: Class-priority post-processing. Since neural models systematically under-predict minority classes, we apply threshold-based routing to the meta-learner output:

$$\hat{y} = \begin{cases} \text{Adv} & \text{if } p_{\text{adv}}^{\text{meta}} > \tau_{\text{adv}} \\ \text{Machine} & \text{if } p_{\text{mach}}^{\text{meta}} > \tau_{\text{mach}} \\ \arg\max_y \mathbf{p}^{\text{meta}} & \text{otherwise} \end{cases} \quad (9)$$

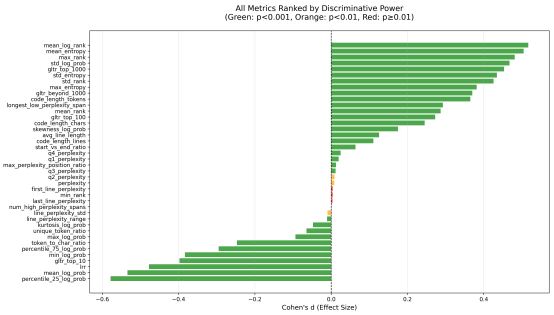

Figure 4: Ranked metric effectiveness across candidate features. Features with the largest effect sizes guided selection of complementary signals for the ensemble.

where thresholds τ_{adv} and τ_{mach} are tuned via grid search on the validation set. The priority ordering (Adversarial checked first) reflects the observation that Adversarial samples are the most under-predicted class. The metric-driven pipeline (Figure 4) guided feature selection and threshold tuning. To address class imbalance, we downsampled dominant classes and used stratified folds to prevent domain leakage. This ensemble raises the official CodeT5p-only score of 0.585 to **0.652**.

4 Experimental Setup

Data. We use the official training and validation splits for all subtasks (500K samples for Subtask A; 900K for Subtask C). No external data was used. We reserved a stratified 10% of the training set as an internal development split for hyperparameter tuning.

Training. All transformer models were fine-tuned with AdamW (learning rate 2e-5, linear warmup over 6% of steps, batch size 16) for up to 7 epochs with early stopping on validation macro F1. We also experimented with class downsampling and test-time augmentation (Wang et al., 2021). Maximum sequence length was 512 tokens.

Evaluation. All results use macro F1, the official metric.

5 Results

5.1 Official Results

Table 1 presents the official leaderboard results for our system on Subtask A and Subtask C.

5.2 Distribution Shift Analysis

Table 2 quantifies the distribution shift between validation and test for both subtasks. The gap is strik-

Task	System	Macro F1
A	FA-AST (GGNN)	0.493
A	CodeT5p-220M	0.559
A	+ downsampling + TTA	0.561
A	+ rank-fusion ens.†	0.643
C	CodeT5p-220M	0.585
C	+ meta-learner ens.†	0.652

Table 1: Test set results. †Post-competition. Subtask A ensemble fuses CodeT5p, CodeBERT, and UniXcoder via layered rank fusion. Subtask C ensemble uses meta-learner stacking with class-priority routing.

ing: on Subtask A both approaches—the CodeT5p-220M encoder and the FA-AST detector—achieve 0.980 macro F1 on the validation set, yet drop to 0.559 and 0.493 respectively on the final test set. A similar pattern holds for Subtask C, where validation performance of 0.810 falls to 0.790 on a preliminary 1K test sample released on HuggingFace and further to 0.585 on the full 500K test set. These gaps confirm that the test sets introduce substantial language and domain shifts absent from training and validation.

Subtask	Val	Test	Δ
A (CodeT5p)	0.980	0.559	–0.421
A (FA-AST)	0.980	0.493	–0.487
C (CodeT5p)	0.810	0.585	–0.225
C (1K HF test)	–	0.790	–

Table 2: Validation vs. test macro F1, illustrating massive distribution shift in both subtasks. Both Subtask A approaches achieve near-perfect validation scores yet degrade sharply on the final test set.

AST features partially recover this loss for unseen languages, where structural patterns transfer more reliably than token-level representations (see Appendix for t-SNE visualizations).

5.3 Error Analysis

Manual inspection of 100 misclassified Subtask A validation examples reveals two dominant failure modes: (1) short snippets (under 20 tokens) from unseen domains, where both token and structural features lack sufficient signal, and (2) terse competitive-programming solutions whose algorithmic style overlaps with LLM outputs.

6 Conclusion

We presented our system for SemEval-2026 Task 13, combining three code-pretrained encoders with a structure-first FA-AST detector. For Subtask A, a layered rank-fusion ensemble exploiting low inter-model correlation (Spearman r=0.35–0.57) raises macro F1 from 0.559 to 0.643. For Subtask C, meta-learner stacking with class-priority routing raises macro F1 from 0.585 to 0.652. Both results highlight that ensemble diversity—measured through rank correlation and complementary feature spaces—is more valuable than individual model strength under distribution shift. Future work could explore domain-adaptive pretraining to further close the gap between validation and test performance.

References

Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Security Symposium*, pages 255–270.

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*, pages 1536–1547. Association for Computational Linguistics.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander Rush. 2019. GLTR: Statistical detection and visualization of generated text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 111–116, Florence, Italy. 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)*, pages 7212–7225. Association for Computational Linguistics.

Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. 2019. Code authorship attribution: Methods and challenges. *ACM Computing Surveys*, 52(1):1–36.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, and 1 others. 2024. StarCoder 2 and The Stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

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, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026a. 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)*, Rabat, Morocco. Association for Computational Linguistics.

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. 2025b. 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.

Hung Pham, Huyen Ha, Van Tong, Dung Hoang, Duc Tran, and Tuyen Ngoc Le. 2024. MAGECODE: Machine-generated code detection method using large language models. *IEEE Access*.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. 2021. TENT: Fully test-time adaptation by entropy minimization. In *Proceedings of the 9th International Conference on Learning Representations (ICLR 2021)*.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2020)*, pages 261–271. IEEE.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C.H. Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Xiaodan Xu, Chao Ni, Xinrong Guo, Shaoxuan Liu, Xiaoya Wang, Kui Liu, and Xiaohu Yang. 2025. Distinguishing LLM-generated from human-written code by contrastive learning. *ACM Transactions on Software Engineering and Methodology*, 34(91):1–31.

A Additional Visualizations

A.1 Distribution Shift: t-SNE of Embeddings

Figure 5 visualizes the pooled 768-dimensional CodeT5p-220M embeddings for 1,000 training and 1,000 test samples from Subtask A using t-SNE. Training and test points occupy largely disjoint regions, confirming the massive distributional gap between splits. Figure 6 shows the same embedding space colored by programming language; unseen test languages (C#, Go, JavaScript, PHP, C) form distinct clusters with minimal overlap to the training languages (Python, Java, C++), explaining the sharp performance degradation reported in Table 2.

Figure 5: t-SNE Projection of Pooled CodeT5p-220M Embeddings(Subtask A). Illustrating Train–Test Distribution Shift

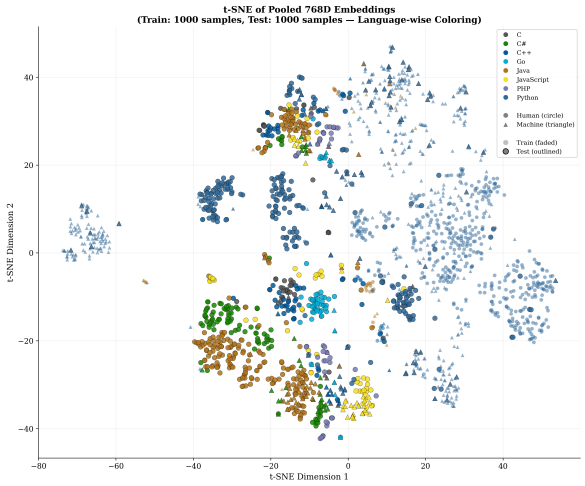

Figure 6: Language-Wise t-SNE Projection of CodeT5p Embeddings(Subtask A).

A.2 Token Length Analysis (Subtask C)

Figures 7 and 8 compare the token-length distribution of Subtask C before and after preprocessing. The raw data has a long tail with only 72.5% of samples fitting within the 512-token context window. After cleaning (removal of artifacts, comment stripping, and whitespace normalization), coverage at 512 tokens rises to 83.6%, reducing truncation-related noise during training.

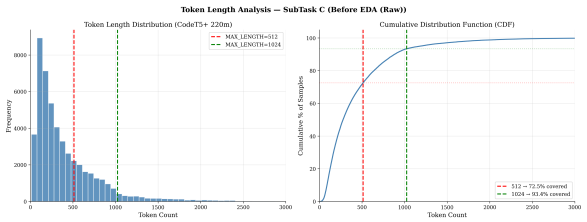

Figure 7: Token length distribution for Subtask C before preprocessing. Only 72.5% of samples fit within 512 tokens.

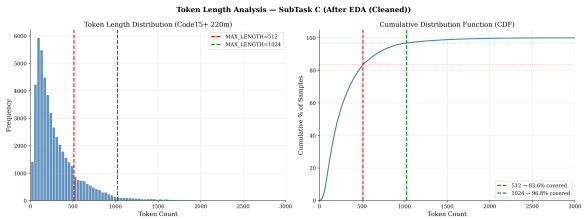

Figure 8: Token-Length Distribution for Subtask C After Preprocessing. Only 83.6% of samples fit within 512 tokens.

A.3 Supplementary Experiment: Subtask B

The raw code snippets are fed into the **CodeT5p-770M** model. The encoder is fully trained end-to-end. On top of the encoder, we attach an **Attention Pooling** layer that computes soft per-token weights over the encoder's last hidden states. Two heads operate on this pooled representation: (i) a **Projection Head** ($1024 \to 512 \to 256$, L2-normalised) used solely to compute the Supervised Contrastive loss and (ii) a **Classification Head** ($1024 \to 512 \to 256 \to 11$, with Batch Normalization and progressive dropout) trained with Focal Loss. **Focal Loss** down-weights well-classified samples, while **Supervised Contrastive Loss** shapes the embedding space so that generators form tight, separable clusters—both losses jointly address the severe class imbalance (88.4% Human). Additionally, several classes had comparatively low representation in the training dataset, such as gemma (0.39%) and bigcode (0.45%).

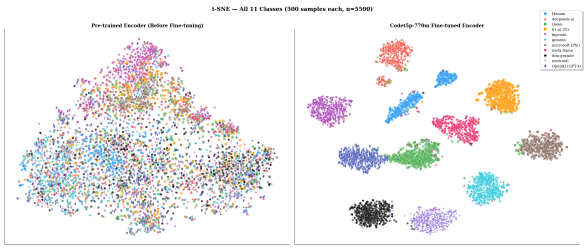

Figure 9: t-SNE of CodeT5p-770M Embeddings (Subtask B): pre-trained (left) vs. fine-tuned (right). Fine-tuning induces clear per-generator clustering.

We train on the full dataset rather than a class-balanced variant. In a separate experiment, down-sampling the Human class to 10K samples peaked at macro F1 of 0.480 (epoch 10), whereas the full dataset reached 0.577 (epoch 10)—a relative gain of 20.1%. The extra Human samples provide richer negatives for the contrastive objective, sharpening the decision boundaries between generators.

We also chose *not* to apply comment or whitespace removal. Stripping comments degraded performance in preliminary runs: commenting style and inline documentation patterns differ across generators and carry useful authorship signal.

Using the official test set released prior to the competition, our best post-competition evaluation achieved a score of **0.40759** on Subtask B.