TransformerTrio at SemEval-2026 Task 13: Navigating Domain Shift and Representation Instability in Machine-Generated Code Detection

Avi Patel, Manthan Laddha, Pushti Sapovadiya, Pruthwik Mishra and Shrikant Malviya
Sardar Vallabhbhai National Institute of Technology (SVNIT), Surat, India
{u24ai071, u24ai072, u24ai101, pruthwikmishra}@aid.svnit.ac.in
shrikant@coed.svnit.ac.in

Abstract

Detecting machine-generated code is increasingly challenging due to advances in code generation models and domain variation across programming tasks. We present our submissions to SemEval-2026 Task 13, evaluating detection in three settings: binary human vs. machine classification, multi-class generator attribution, and four-way authorship classification including hybrid and adversarial cases. We compare feature-based, transformer-based, and hybrid approaches under domain shift and limited supervision. Results show that domain-specific signals often dominate model decisions, degrading generalization when training and test distributions diverge. Increasing model complexity does not consistently improve performance in low-resource or cross-domain settings and may amplify spurious correlations. These findings emphasize robustness and feature alignment over model sophistication for reliable detection.

1 Introduction

Large language models are widely used in software development, making machine-generated code detection increasingly important, particularly under domain shift, unseen languages, and hybrid authorship settings. SemEval-2026 Task 13 evaluates these challenges across multiple subtasks, languages, and generators (Orel et al., 2026b). We compare feature-based models, transformer fine-tuning, and representation-level approaches.[1] Rather than focusing on a single architecture, we analyze how modeling strategies behave under distribution shift and varying supervision.

Early work showed that stylometric and structural features (e.g., line statistics, identifier patterns, AST measures) can provide authorship signals using classical models (Li et al., 2023; Idialu et al., 2024). Deep learning methods later demonstrated improved ability to capture higher-order patterns, motivating pretrained language models for code (Vaswani et al., 2017; Tulchinskii et al., 2023). Recent systems such as GPTSniffer (Nguyen et al., 2024) and GPT-Sensor (Xu et al., 2025) train text-based language models to distinguish human and ChatGPT-generated code, primarily using CodeSearchNet-derived data (Husain et al., 2020). Although effective in constrained settings, they remain limited in domain and generator diversity. Benchmarks such as Droid and CodeT (Orel et al., 2025b,a) and CodeMirage (Guo et al., 2025) expanded evaluation to multi-domain and hybrid settings, but still exhibit limitations in adversarial and generator coverage. SemEval-2026 Task 13 further increases scale and complexity.

2 Task Description

SemEval-2026 Task 13 (Orel et al., 2026b) studies the detection and attribution of machine-generated code under distribution shift across programming languages, domains, and generators.

2.1 Subtask A: Binary Machine-Generated Code Detection

Subtask A evaluates binary classification of human-written vs. machine-generated code under distribution shift. Training data is limited to the algorithmic domain, while test data includes additional domains and unseen programming languages, requiring cross-domain generalization. Table 1 reports the language-wise data distribution.

2.2 Subtask B: Multi-Class Authorship Detection

Subtask B extends detection to eleven classes (human plus ten LLM families). Systems must attribute code to specific generators, including unseen models from known families. Table 2 shows the class distribution.

2.3 Subtask C: Hybrid Code Detection

Subtask C introduces four labels: human-written, machine-generated, hybrid, and adversarial. The

[1] All code will be released publicly: https://github.com/Avi4306/SemEval-2026-Task-13.

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 1015–1026

Language	Train	Validation	Development
C++	23K	5K	75
Python	458K	91K	303
Java	19K	4K	256
Go	–	–	60
PHP	–	–	48
C#	–	–	122
C	–	–	51
JS	–	–	85
Machine	262K	52K	936
Human	238K	48K	64
Total	500K	100K	1K

Table 1: Language-wise data distribution for Subtask A. Counts are rounded to the nearest thousand (K).

Generator	Train	Validation	Development
Human	442K	88K	474
DeepSeek-AI	4K	847	21
Qwen	9K	1.8K	73
01-ai	3K	650	21
BigCode	2K	445	10
Gemma	2K	372	36
Phi	6K	1K	54
Meta-LLaMA	8K	1.7K	61
IBM-Granite	8K	1.6K	18
Mistral	5K	895	18
OpenAI	11K	2.2K	214
Total	500K	100K	1K

Table 2: Class-wise data distribution for Subtask B.

task reflects collaborative and style-mimicking scenarios. Table 3 summarizes the label distribution.

Label	Train	Validation	Development
Human-written	485K	107K	554
Machine-generated	210K	46K	228
Hybrid	85K	19K	84
Adversarial	118K	26K	134
Total	900K	200K	1K

Table 3: Label-wise distribution for Subtask C.

Dataset Origin. The datasets used in this shared task are derived from the AICD BENCH benchmark (Orel et al., 2026a), a large-scale and challenging benchmark for AI-generated code detection that spans multiple programming languages, generators, and domains.

3 System Description

This section describes the architectures and modeling strategies used in all subtasks. Our systems combine lightweight feature-based models[2] with pretrained transformers and hybrid ensembles that aims to balance interpretability and robustness under distribution shift. The configuration details and hyperparameters are deferred to Section 4.

For each subtask, we detail the best-performing model below; others are described in Appendix A.1.

[2]Each model is named as "Model Subtask$i$" where $i \geq 1$

3.1 Subtask A: Binary Machine-Generated Code Detection

3.1.1 Model A1: Feature-Based XGBoost

Our first model is a gradient-boosted decision tree classifier (Chen and Guestrin, 2016a) trained on a small set of handcrafted features that capture stylistic and structural properties of a source code (Chen and Guestrin, 2016b).

The handcrafted features are: (1) Word length kurtosis, (2) Approximate Halstead volume computed from lexical operators and operands (Halstead, 1977), (3) Total number of lines, (4) Number of non-empty lines, and (5) Ratio of single-line comments among all comment lines.

We also explore two variants: Model A2 augments handcrafted features with frozen `ModernBERT-Base` embeddings, while Model A3 uses adversarial validation to remove distribution-specific dimensions for improved robustness. Details are provided in Appendix A.1.1 and A.1.2.

3.2 Subtask B: Multi-Class Authorship Detection

3.2.1 Model B1: Fine-Tuned Transformer

Model B1 fine-tunes a pretrained transformer on the Subtask B training data to directly predict one of the authorship classes for Subtask B. Input code snippets are tokenized using the model's native tokenizer and truncated to a fixed maximum sequence length and optimized using cross-entropy loss.

3.3 Subtask C: Hybrid Code Detection

3.3.1 Model C1: Fine-Tuned Transformer

Model C1 adapts the fine-tuning strategy from Model B1 to the four-class classification setting detailed in Table 3: human-written, machine-generated, hybrid, and adversarial code.

We additionally explore a two-stage stacking approach (Model C2), in which logit-derived features from a fine-tuned transformer are passed to an XGBoost meta-classifier. The full description is provided in Appendix A.1.3.

4 Experimental Setup

For all subtasks, we follow the official SemEval–2026 Task 13 data splits. The description of the dataset is mentioned in Section 2.

4.1 Data Preprocessing

Models operate on raw source code without normalization or language-specific preprocessing in all subtasks. Feature-based approaches extract statis-

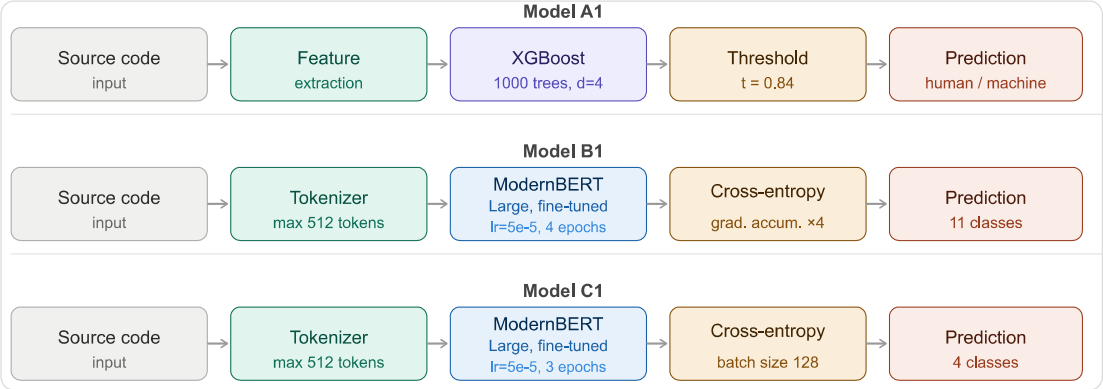


Figure 1: Overview of the primary model pipelines for all three subtasks (each pipeline flows left-to-right). **Top (Model A1):** Feature-based XGBoost pipeline for binary human vs. machine detection (Subtask A). **Middle (Model B1):** Fine-tuned `ModernBERT-Large` pipeline for 11-class generator attribution (Subtask B). **Bottom (Model C1):** Fine-tuned `ModernBERT-Large` pipeline for four-class hybrid authorship detection (Subtask C).

tics directly from the text, while transformer-based models rely on the pretrained tokenizers of their respective encoders.

For Model C2 (UniXcoder-based stacking), we additionally evaluate an ablation variant with cost-sensitive learning using weighted cross-entropy loss to address class imbalance.

4.2 Subtask A: Binary Detection

Model A1: Feature-Based XGBoost. Model A1 employs an XGBoost classifier trained on five handcrafted features. We use 1000 trees with a learning rate of 0.01, maximum depth 4, subsampling and column subsampling ratios of 0.8, and ℓ_2 regularization $\lambda = 10$.

Model A2: XGBoost with Transformer Embeddings. Model A2 augments the handcrafted features with frozen transformer embeddings extracted from `ModernBERT-Base`. Inputs are tokenized to a maximum length of 1024 tokens, and masked mean pooling is used to obtain a fixed-dimensional representation. The pooled embedding is concatenated with the handcrafted features and trained using the same XGBoost configuration as Model A1.

Model A3: Adversarially Filtered Embeddings. Model A3 extracts `UniXcoder` (`microsoft/unixcoder-base-nine`) (Guo et al., 2022) embeddings, fine-tuned for 1 epoch with a learning rate of 2×10^{-5}, batch size 64, and warmup ratio 0.1, truncating inputs to a maximum sequence length of 512 tokens. An adversarial XGBoost classifier (100 trees, depth 4, learning rate 0.1, `tree_method='hist'`) is trained with three-fold stratified cross-validation to distinguish training from test embeddings using ROC–AUC. The 50 most discriminative dimensions, identified via gain-based feature importance, are iteratively removed until the adversarial AUC falls below 0.7 or fewer than 10 dimensions remain.

4.3 Subtask B: Multi-Class Authorship Detection

Model B1: Fine-Tuned Transformer. Model B1 fine-tunes `ModernBERT-Large` for an eleven-class authorship classification task. Inputs are tokenized with a maximum sequence length of 512 tokens. Training is performed for 4 epochs using the AdamW optimizer (Loshchilov and Hutter, 2019) with a learning rate of 5×10^{-5}. A cosine learning rate schedule with a warmup ratio of 0.1 is applied. We use a staged batch size strategy, increasing from 32 in the first epoch to 64 in later epochs, combined with gradient accumulation of 4 steps, resulting in a larger effective batch size for improved stability and efficiency. Early stopping with a patience of 3 epochs is used based on validation macro-F1.

4.4 Subtask C: Hybrid Detection

Model C1: Fine-Tuned Transformer. Model C1 fine-tunes a single pretrained transformer for four-class classification. Inputs are truncated to 512 tokens. Training uses AdamW optimization for 3 epochs with a learning rate of 5×10^{-5} and effective batch size 32.

Model C2: Logit-Level Meta-Classifier. Model C2 uses a fine-tuned `UniXcoder-Base` (3 epochs, max length 512, AdamW with $lr = 5 \times 10^{-5}$, weight decay 0.001, warmup 0.1, batch size 64 with gradient accumulation 2, bf16, and gradient checkpointing). From the output logits, we construct features comprising raw logits,

softmax probabilities, logit statistics (mean, std, max, min), uncertainty measures (confidence, entropy, top-2 gap), and class ranks (19 features for 4 classes), which are standardized. An XGBoost meta-classifier (200 estimators, depth 3, $lr \approx 0.05$, with subsampling and ℓ_1/ℓ_2 regularization) is trained on these features.

4.5 Evaluation Metric

All models are evaluated using macro-averaged F1 scores, following the official SemEval–2026 Task 13 evaluation metric.

5 Results

5.1 Main Quantitative Findings

Table 4 reports test macro F_1 scores for all systems across Subtasks A, B, and C. Feature-based Model A1 performs best in Subtask A, Model B1 achieves the highest macro F_1 in Subtask B, and Model C1 slightly outperforms Model C2 in Subtask C.

Model	Subtask A	Subtask B	Subtask C
Model A1	**0.6724**	–	–
Model A2	0.6031	–	–
Model A3	0.2906	–	–
Model B1	–	**0.3997**	–
Model C1	–	–	**0.6337**
Model C2	–	–	0.6251

Table 4: Test-set macro F_1 scores for submitted models.

5.2 Subtask A: Threshold Sensitivity Analysis

Table 5 shows the effect of the selection of decision thresholds on the performance of Subtask A. For both Model A1 and Model A2, selecting an optimal threshold on the development set yields substantial improvements over the default threshold of 0.5. Model A3 remains weak in comparison.

Model	Threshold	Macro F_1
Model A1	0.50	0.6160
Model A1	**0.84**	0.6724
Model A2	0.50	0.3480
Model A2	**0.95**	0.6031
Model A3	0.50	0.2906

Table 5: Impact of decision threshold on Subtask A.

5.3 Subtask A: Binary Detection

For Subtask A, feature-driven modeling proves effective. Model A1 that relies on handcrafted lexical, structural, and statistical features with XGBoost, achieves the strongest test performance. Model A2 that augments these features with contextual embeddings from `ModernBERT-Large` improves on simpler baselines but does not surpass Model A1, suggesting contextual embeddings may be sensitive to train–test distribution shifts.

Model A1: Language-wise Performance. Table 8 shows variability across languages for Model A1. Performance is relatively stable on seen languages (Python, C++), but inconsistent on unseen ones: Go performs well, while PHP and JavaScript perform poorly. Full per-class classification reports and language-level breakdowns are provided in Appendix A.4 and Appendix A.5.

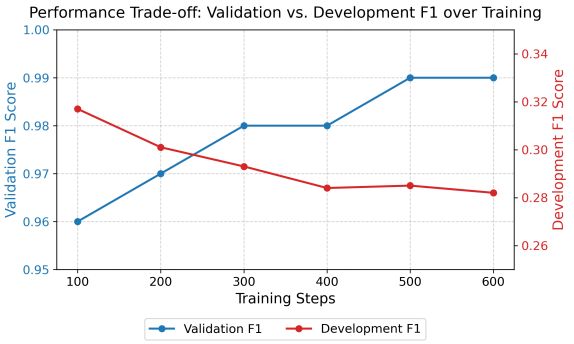
Model A2: Language-wise Performance. Despite greater representational capacity, Model A2 scores below Model A1 (0.6031 vs. 0.6724), requiring threshold tuning to 0.95 to avoid near-total collapse toward the machine class. Language-wise (Table 13), this tuning improves Python and C but substantially degrades Go and C++, indicating that frozen embeddings introduce distributional biases that a single global threshold cannot resolve across languages. This is consistent with the representation instability analysis in Section 5.3.1.

5.3.1 Representation Instability and Adversarial Filtering

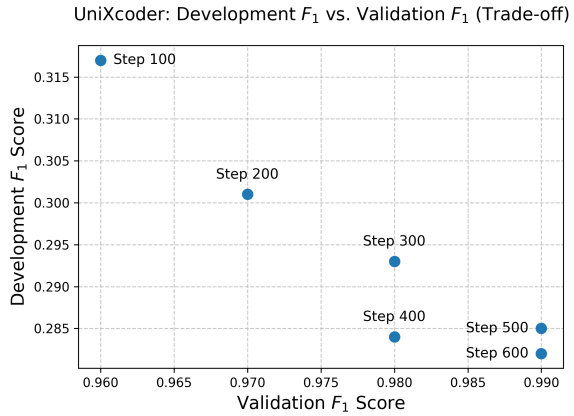
To analyze generalization during continued training, we fine-tuned `UniXcoder-Base` for one epoch. While in-domain validation F_1 increased monotonically, development set comprising both in-domain and out-of-domain samples performance consistently declined (Figure 2). This inverse relationship, also observed with `ModernBERT` embeddings, confirms a sharp ID–OOD generalization trade-off consistent with prior research (Teney et al., 2023).

To better understand this degradation, we examine the stability of transformer representations across domains using pretrained `ModernBERT-Large`, which provides a 1024-dimensional embedding space. For each embedding dimension, we compute correlations with the binary class label on the training and development data, finding that roughly 40% of dimensions exhibit correlation sign reversals across domains. Dimensions predictive of human-written code during training often become indicative of machine-generated code at test time, suggesting that contextual embeddings encode domain-specific and potentially spurious signals that undermine OOD generalization.

Model A3 applies adversarial validation to remove distribution-specific dimensions. An adversarial XGBoost classifier achieves ROC–AUC 0.94, decreasing to 0.87 after iterative removal, at which point the minimum-dimension constraint was reached. However, reduced separability does



(a) Validation and dev F_1 scores as a function of training steps.



(b) Development vs. validation F_1 across training checkpoints.

Figure 2: Fine-tuning `UniXcoder-Base` performance trade-off: validation F_1 rises while development F_1 falls.

not improve performance (macro F_1 0.2906), suggesting that domain-specific information is diffusely encoded. The adversarial process is leakage-free and uses no test labels; further details are provided in Appendix A.11 and Appendix A.10.

5.4 Subtask B: Multi-Class Authorship Detection

Model B1 achieves a macro F_1 of 0.3997 on the official test set. Per-class results (Appendix A.12) reveal a strong imbalance between majority and minority classes: Human (F_1 = 0.98) and OpenAI (F_1 = 0.75), comprising 68.8% of development support, are classified reliably, while the remaining nine LLM families achieve F_1 scores between 0.07 and 0.52. The weakest classes (DeepSeek-AI, BigCode, Mistral) have the smallest support ($\leq$21 samples), suggesting that poor recall on these classes reflects extreme class imbalance rather than a fundamental representational limitation.

Model B1: Generator-wise Performance. Precision generally exceeds recall for majority classes and falls below recall for minority classes, indicating systematic over-prediction of common generators and under-prediction of rare ones.

5.5 Subtask C: Hybrid Detection

Both systems perform competitively, with Model C1 (0.6337) slightly outperforming Model C2 (0.6251), indicating that single-stage fine-tuning is sufficient for the four-class label space. The large gap between development scores (C1: 0.8565, C2: 0.8254) and test scores reflects substantial distribution shift.

Model C1: Language-wise Performance. Table 17 shows variability across languages for Model C1. Performance is consistently higher for languages with larger training support (Python, Java), while low-resource languages suffer: PHP performs worst (F_1 = 0.554) and C best (0.896). Full per-class metrics are provided in Appendix A.14.

Model C2: Language-wise Performance. Model C2 follows a similar language-wise ordering (Table 19), though absolute scores differ. The most frequent errors are AI→Adversarial (19 cases) and AI→Hybrid (13 cases), confirming that the core ambiguity lies between the three machine-generated categories rather than in human vs. machine discrimination. Full per-class metrics and the confusion matrix are provided in Appendix A.15.

6 Conclusion

We provide a systematic comparison of feature-based, transformer-based, and representation-level approaches for SemEval-2026 Task 13 across all three subtasks, analyzing their interaction with supervision and domain alignment. In Subtask A, feature-based XGBoost models outperform embedding-based variants, while naive integration of frozen or fine-tuned embeddings worsens out-of-distribution generalization. We observe instability in embedding–label correlations across domains, and adversarial filtering reduces distributional separability without restoring predictive accuracy. For Subtask B, fine-tuning a large pretrained transformer yields limited macro-F_1, reflecting challenges from class imbalance and data sparsity. In Subtask C, end-to-end fine-tuning consistently surpasses representation stacking, indicating that direct supervision is more effective in hybrid authorship settings. Overall, robustness under domain shift requires more than representation-level distribution matching; future work should explore hybrid adversarial and task-aware fine-tuning or causal feature selection to better balance robustness and discrimination.

Acknowledgments

The authors acknowledge the use of the Aurora High-Performance Computing (HPC) Cluster at Sardar Vallabhbhai National Institute of Technology (SVNIT), Surat, for the research reported in this work.

References

Tianqi Chen and Carlos Guestrin. 2016a. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA. Association for Computing Machinery.

Tianqi Chen and Carlos Guestrin. 2016b. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794. ACM.

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, Dublin, Ireland. Association for Computational Linguistics.

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. *Preprint*, arXiv:2506.11059.

Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *Preprint*, arXiv:1909.09436.

Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M. Atlee, and Mei Nagappan. 2024. Whodunit: Classifying code as human authored or gpt-4 generated - a case study on codechef problems. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 394–406, New York, NY, USA. Association for Computing Machinery.

Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. 2018. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7167–7177, Red Hook, NY, USA. Curran Associates Inc.

Ke Li, Sheng Hong, Cai Fu, Yunhe Zhang, and Ming Liu. 2023. Discriminating human-authored from chatgpt-generated code via discernable feature analysis. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 120–127.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. *Preprint*, arXiv:1711.05101.

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. *Journal of Systems and Software*, 214:112059.

Shuaicheng Niu, Jiaxiang Wu, Yifan Zhang, Zhiquan Wen, Yaofo Chen, Peilin Zhao, and Mingkui Tan. 2023. Towards stable test-time adaptation in dynamic wild world. In *Internetional Conference on Learning Representations*.

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.

Damien Teney, Yong Lin, Seong Joon Oh, and Ehsan Abbasnejad. 2023. Id and ood performance are sometimes inversely correlated on real-world datasets. *Preprint*, arXiv:2209.00613.

Eduard Tulchinskii, Kristian Kuznetsov, Laida Kushnareva, Daniil Cherniavskii, Sergey Nikolenko, Evgeny Burnaev, Serguei Barannikov, and Irina Piontkovskaya. 2023. Intrinsic dimension estimation for robust detection of ai-generated texts. In *Advances in Neural Information Processing Systems*, volume 36, pages 39257–39276. Curran Associates, Inc.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. 2021. Tent: Fully test-time adaptation by entropy minimization. In *International Conference on Learning Representations*.

Yanwen Wang, Mahdi Khodadadzadeh, and Raúl Zurita-Milla. 2025. On the use of adversarial validation for quantifying dissimilarity in geospatial machine learning prediction. *GIScience & Remote Sensing*, 62(1).

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Griffin Thomas Adams, Jeremy Howard, and Iacopo Poli. 2025. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2526–2547, Vienna, Austria. Association for Computational Linguistics.

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 Trans. Softw. Eng. Methodol.*, 34(4).

A Appendix: Additional Implementation Details and Analysis

This appendix provides low-level implementation details, feature analyses, and exploratory discussions that support reproducibility and further investigation.

A.1 Additional Model Descriptions

This section provides full descriptions of the non-primary models summarized in Section 3.

A.1.1 Model A2: XGBoost with Transformer-Based Embeddings

Our second model combines the same handcrafted features from Model A1 with contextual embeddings extracted from `ModernBERT-Base` model (Warner et al., 2025). The transformer encoder is kept frozen and used as a feature extractor.

To obtain a fixed-length representation from the transformer, we apply mask-aware mean pooling over token embeddings:

$$\mathbf{h} = \frac{1}{\sum_i m_i} \sum_{i=1}^{T} m_i \mathbf{e}_i, \tag{1}$$

where $\mathbf{e}_i$ is the embedding of the i-th token and m_i is the attention mask indicating valid (non-padding) tokens.

The pooled transformer embedding is concatenated with the handcrafted features and fed into the XGBoost model, as in Model A1.

A.1.2 Model A3: Adversarially Filtered Transformer Embeddings

Model A3 explores whether distribution-specific signals present in transformer embeddings can be identified and removed using adversarial validation (Wang et al., 2025), with the goal of improving robustness under domain shift.

Embedding Extraction. We fine-tune `UniXcoder` (`microsoft/unixcoder-base-nine`) (Guo et al., 2022) on the Subtask A training data for 1 epoch using AdamW with a learning rate of 2×10^{-5}, batch size 64, a warmup ratio of 0.1, and a maximum sequence length of 512 tokens. The fine-tuned model is then used as a feature extractor: each code snippet is represented by a single pooled embedding vector extracted from the encoder.

Adversarial Validation. We apply adversarial validation (Wang et al., 2025) by training XGBoost to distinguish training from test embeddings. An ROC–AUC significantly above 0.5 identifies a measurable distribution shift, indicating that the classifier can successfully identify a sample's origin based on domain-specific features.

Iterative Dimension Dropping. To reduce reliance on distribution-specific latent dimensions, we iteratively remove embedding dimensions that contribute most to adversarial separability. At each iteration, we:

1. Train an adversarial XGBoost classifier on the current embeddings.
2. Compute feature importances from the trained model.
3. Remove the top-k most important dimensions.

This process continues until the adversarial AUC drops below a predefined threshold (0.7), or until too few dimensions remain to support stable training.

Final Classification. After adversarial filtering, the remaining embedding dimensions are standardized and used to train a final XGBoost classifier for the Subtask A prediction task.

Discussion. This model did not produce competitive F1 performance. We include this model to provide insights into the trade-offs between domain

invariance and predictive performance.

A.1.3 Model C2: Logit-Level Stacking Ensemble

Model C2 employs a two-stage stacking architecture. In the first stage, a single fine-tuned transformer model is applied to the validation and test sets to extract and cache raw prediction logits.

In the second stage, logit-derived features are constructed, including raw logits, softmax probabilities, summary statistics of the per-model logit vector, confidence and entropy-based uncertainty measures, top-two probability margins, and rank features. An XGBoost meta-classifier is trained on the validation logits and logit-derived features to produce the final predictions.

A.2 Model A1 and A2: Preprocessing and Normalization

All scalar code-level features were z-score normalized using mean and standard deviation computed on the training split. Comment-related features were extracted using regular-expression-based heuristics. Empty lines and whitespace-only lines were excluded from line-based feature computations. The single-line comment ratio was defined relative to the total number of detected comment lines.

A.3 Model A1: Feature-Level Analysis

Feature Importance. Table 6 reports the top normalized feature importances derived from Model A1 for Subtask A. Importance values were computed using the absolute contribution of each feature to the final decision function and normalized to sum to one.

Feature	Importance
Single-line comment ratio	0.7968
Total lines of code	0.0596
Non-empty lines	0.0575
Halstead volume	0.0444
Word-length kurtosis	0.0418

Table 6: Top feature importances for Subtask A (Model A1).

The dominance of the single-line comment ratio suggests that stylistic cues related to documentation and commenting behavior are strong indicators for Subtask A, while structural and complexity-based features (e.g., Halstead volume) provide complementary but comparatively weaker signals.

Label	Precision	Recall	F_1	Support
Human (0)	0.8711	0.8610	0.8660	777
Machine (1)	0.5345	0.5561	0.5451	223
Macro Avg	0.7028	0.7085	0.7055	1000
Weighted Avg	0.7960	0.7930	0.7944	1000

Table 7: Classification report for Model A1 at threshold 0.84.

Language	F1 (Thr=0.84)	Seen/Unseen
Python	0.6577	Seen
Java	0.4545	Seen
C++	0.6190	Seen
C	0.5185	Unseen
C#	0.4828	Unseen
JavaScript	0.3600	Unseen
Go	0.7097	Unseen
PHP	0.2000	Unseen

Table 8: Macro F_1 by programming language for Subtask A (Model A1, threshold = 0.84) on Development set.

A.4 Model A1: Classification Report (Threshold = 0.84)

Language-wise Performance. Table 8 shows variability across languages. Performance is relatively stable on seen languages (Python, C++), but inconsistent on unseen ones: Go performs well, while PHP and JavaScript perform poorly. This suggests that identifying domain-invariant features could improve generalization, but extracting such features remains challenging.

A.5 Model A1: Classification Report (Threshold = 0.5)

Table 9 presents the classification report for Model A1 at the default threshold of 0.5 on the test set.

Label	Precision	Recall	F_1	Support
Human (0)	0.9122	0.6152	0.7348	777
Machine (1)	0.3718	0.7937	0.5064	223
Macro Avg	0.6420	0.7045	0.6206	1000
Weighted Avg	0.7917	0.6550	0.6839	1000

Table 9: Classification report for Model A1 at threshold 0.5.

A.6 Model A1: Language-wise Performance at Default Threshold

Table 10 reports macro F_1 scores across programming languages for Model A1 using the default decision threshold of 0.5 on the development set.

Performance remains highly variable across languages even at the default threshold, with strong results on Python and Go but persistent weakness on PHP, reinforcing the impact of domain and language-specific variation.

Language	F1 (Thr=0.50)	Seen/Unseen
Python	0.6404	Seen
Java	0.4118	Seen
C++	0.4561	Seen
C	0.3784	Unseen
C#	0.4615	Unseen
JavaScript	0.5823	Unseen
Go	0.6842	Unseen
PHP	0.1622	Unseen

Table 10: Macro F_1 by programming language for Subtask A (Model A1, threshold = 0.5) on the development set.

A.7 Model A2: Classification Report (Threshold = 0.5)

Label	Precision	Recall	F_1	Support
Human (0)	0.9615	0.1931	0.3215	777
Machine (1)	0.2571	0.9731	0.4067	223
Macro Avg	0.6093	0.5831	0.3641	1000
Weighted Avg	0.8045	0.3670	0.3405	1000

Table 11: Classification report for Model A2 at threshold 0.5 on development set.

A.8 Model A2: Classification Report (Threshold = 0.95)

Table 12 shows that increasing the decision threshold to 0.95 substantially improves class balance and macro F_1, correcting the strong bias toward machine predictions observed at lower thresholds.

Label	Precision	Recall	F_1	Support
Human (0)	0.8320	0.8031	0.8173	777
Machine (1)	0.3880	0.4350	0.4101	223
Macro Avg	0.6100	0.6190	0.6137	1000
Weighted Avg	0.7330	0.7210	0.7265	1000

Table 12: Classification report for Model A2 at threshold 0.95 development set.

A.9 Model A2: Language-wise Performance Analysis

Table 13 reports macro F_1 scores across programming languages for Model A2 under two decision thresholds (0.50 and 0.95) on the development set.

Higher thresholding (0.95) improves performance on Python but degrades performance on several other languages (e.g., Go and C++), indicating that threshold tuning introduces language-dependent trade-offs.

A.10 Subtask A: Stable-Dimension Adversarial Analysis

To further analyze representation instability, we restrict embeddings to dimensions whose correlation with the class label did not exhibit sign reversal between training and development data, and whose absolute change in correlation magnitude is at most 0.3. Figure 3 visualizes the resulting per-dimension correlations on the training set versus the development set. Adversarial validation performs on this restricted feature set still yields a high ROC–AUC of 0.9413, indicating that substantial domain-specific information remains even among ostensibly stable dimensions.

Language	F1 (Thr=0.50)	F1 (Thr=0.95)
PHP	0.1250	0.0909
Java	0.3065	0.2432
C++	0.3291	0.2222
C	0.3600	0.4444
C#	0.3944	0.3529
Python	0.4830	0.5473
Go	0.5600	0.2105
JavaScript	0.5614	0.4286

Table 13: Language-wise macro F_1 scores for Model A2 at different decision thresholds on the development set.

A.11 Exploratory Discussion: Practical Extensions of Model A3

A natural extension of Model A3 is to make inference explicitly aware of out-of-distribution (OOD) domain shifts and to adapt the contribution of latent features accordingly. One practical approach is to incorporate a lightweight domain-shift detector operating on intermediate representations (Lee et al., 2018). Such a module could estimate the degree to which an input deviates from the training distribution and use this signal to down-weight or suppress latent features that are empirically unstable across domains before producing the final prediction.

A more ambitious extension involves test-time or online adaptation using a two-model framework. In this setting, an auxiliary adaptation model is trained to identify domain shift from intermediate representations and to generate updates to the parameters of a separate prediction model. The prediction model then performs inference using these adapted weights, while remaining fixed with respect to task supervision. The adaptation signals could be learned from unlabeled target-domain data and guided by simple distributional alignment objectives over latent feature statistics, without requiring access to ground-truth labels (Wang et al., 2021; Niu et al., 2023). Compared to static feature masking, this decoupled adaptation mechanism may enable more targeted and stable adjustment of parameters that remain predictive under domain shift.

While these strategies were not explored in the present work, they represent practical directions for bridging the gap between identifying domain-generalizable latent features and actively exploiting this information at inference time to improve ro-

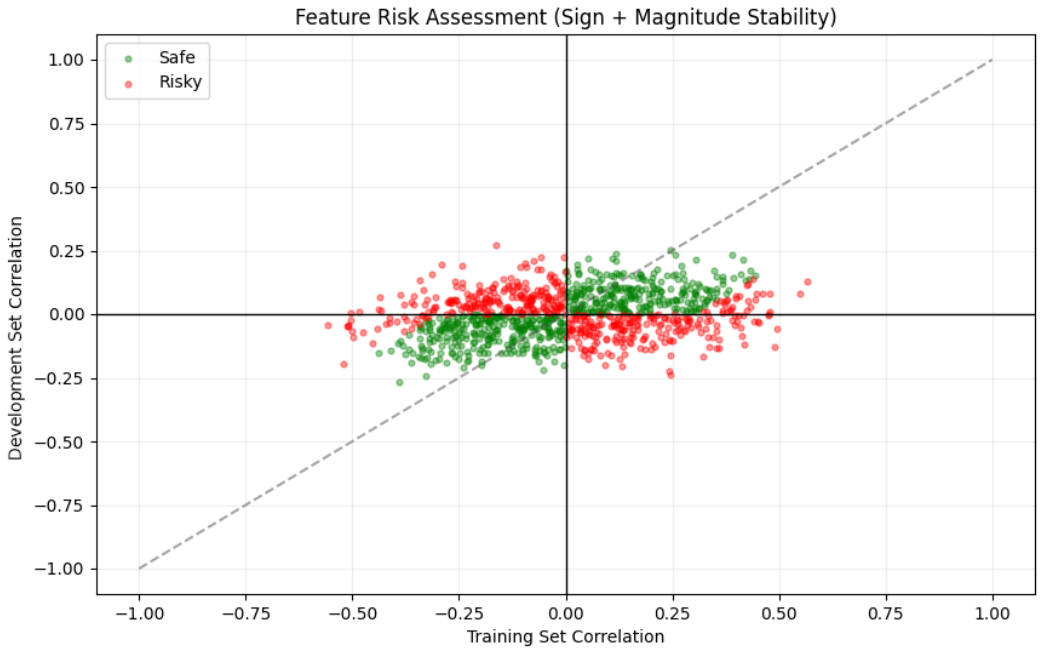


Figure 3: Per-dimension correlation with the class label on the training set versus the development set, computed after filtering to embedding dimensions with stable correlation signs and limited magnitude drift. Substantial divergence remains between training and test correlations, indicating persistent representation instability under domain shift.

bustness under domain shift.

A.12 Model B1: Classification Report

Table 14 presents the per-class precision, recall, and F_1 scores for Model B1 on the test set.

Label	Precision	Recall	F_1	Support
0	0.96	1.00	0.98	474
1	0.06	0.10	0.07	21
2	0.26	0.32	0.28	73
3	0.48	0.52	0.50	21
4	0.17	0.20	0.18	10
5	0.64	0.44	0.52	36
6	0.25	0.22	0.24	54
7	0.31	0.20	0.24	61
8	0.23	0.44	0.30	18
9	0.17	0.39	0.24	18
10	0.86	0.66	0.75	214
Macro Avg	0.40	0.41	0.39	1000
Weighted Avg	0.73	0.71	0.71	1000

Table 14: Classification report for Model B1 on Subtask B.

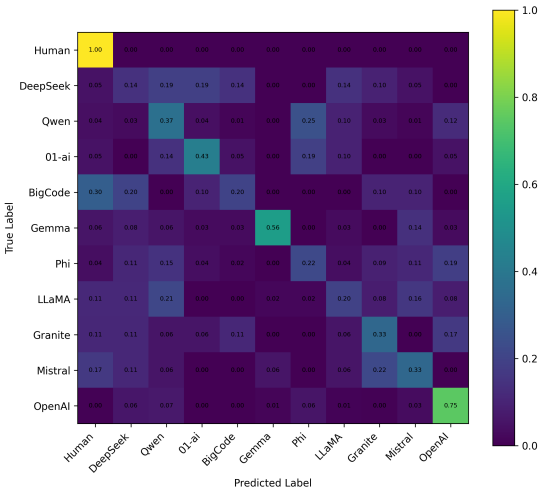


Figure 4: Row-normalized confusion matrix for Model B1 on Subtask B. Each row sums to 1, highlighting systematic confusion among generator classes and the dominance of majority classes.

Error Analysis. Three distinct performance tiers emerge from Table 14.

High-performing classes. Human (class 0, F_1 = 0.98, support 474) and OpenAI (class 10, F_1 = 0.75, support 214) benefit from the two largest support counts. The gap between precision (0.86) and recall (0.66) for OpenAI indicates that the model correctly identifies many true OpenAI samples but also assigns the class to a non-trivial fraction of other generators' output.

Moderate-performing classes. 01-ai (class 3, F_1 = 0.50) and Gemma (class 5, F_1 = 0.52) are the strongest among the minority LLM families. Gemma has the highest precision (0.64) in this tier, though its recall (0.44) remains low given its limited support (36 samples).

Low-performing classes. DeepSeek-AI (class 1, F_1 = 0.07, support 21) is the weakest, with near-zero precision (0.06), indicating that most positive predictions for this class are false positives. IBM-Granite (class 8) and Mistral (class 9), each with only 18 development samples, achieve F_1 scores of 0.30 and 0.24 respectively; their recall values (0.44 and 0.39) exceed their precision, consistent with a model that broadly predicts these classes rather than discriminating them sharply.

Overall pattern. The weighted average F_1 (0.71) substantially exceeds the macro average (0.39), confirming that the aggregate score is dominated by the two large classes. The remaining nine classes contribute 31.2% of total support yet account for the majority of the macro-F_1 penalty, suggesting that class-weighted loss, oversampling, or few-shot adaptation would be the most effective avenues for improving performance on underrepresented generators.

A.13 Subtask B: Epoch-wise Performance Analysis

Effect of Training Epochs. Table 15 reports the macro F1 scores obtained by ModernBERT-Large on Subtask B across different fine-tuning epochs. All models were trained with identical hyperparameters and a fixed maximum sequence length, varying only the number of training epochs.

Epoch	Macro F_1
1	0.3771
2	0.3915
3	0.3997
4	0.3997

Table 15: Epoch-wise performance of ModernBERT-Large on Subtask B (test set).

The performance improves consistently from epoch 1 to epoch 3, after which it saturates. This suggests that the model converges early for Subtask B and additional training does not yield further gains.

A.14 Model C1: Classification Report

Table 16 presents per-class precision, recall, and F_1 scores for Model C1 on the development set.

A.15 Model C2: Classification Report and Error Analysis

Table 18 and Figure 6 present the per-class results and confusion matrix for Model C2 on the development set, and Table 19 reports the language-wise macro F_1 scores.

Label	Precision	Recall	F_1	Support
Human	0.9632	0.9910	0.9769	554
AI	0.8559	0.8333	0.8444	228
Hybrid	0.8148	0.7857	0.8000	84
Adversarial	0.8268	0.7836	0.8046	134
Macro Avg	0.8652	0.8484	0.8565	1000
Weighted Avg	0.9080	0.9100	0.9087	1000

Table 16: Classification report for Model C1 on Subtask C (development set).

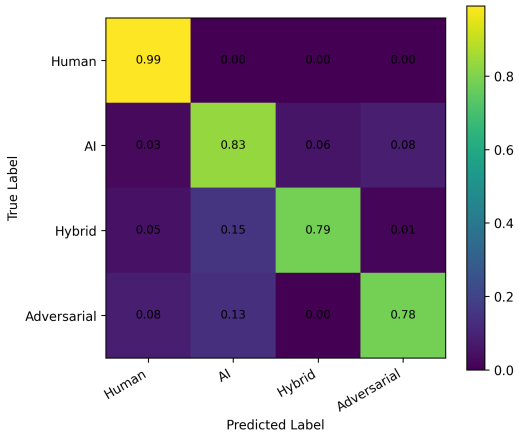


Figure 5: Row-normalized confusion matrix for Model C1 on Subtask C. The matrix highlights confusion between AI, Hybrid, and Adversarial classes, indicating ambiguity among machine-generated categories.

Error Analysis. Model C2 classifies Human code most reliably (precision 0.939, recall 0.978), benefiting from the large class support (554 samples) and distinctive stylistic signals. The principal failure modes are concentrated in the minority classes.

AI misclassification. Of the 228 AI samples, 19 are predicted as Adversarial and 13 as Hybrid, together accounting for 14% of AI errors. This pattern suggests that the XGBoost meta-classifier conflates uncertainty-bearing AI logit signatures with those of the boundary categories, especially when the transformer's confidence is distributed across multiple classes.

Hybrid misclassification. Of the 84 Hybrid samples, 17 are predicted as AI and 6 as Human. The AI confusion mirrors the symmetric AI→Hybrid pattern (13 cases), indicating that the model struggles to commit to either label for mixed-authorship code. No Hybrid sample is predicted as Adversarial, consistent with the structural difference between partially human-edited code and adversarially disguised machine output.

Adversarial misclassification. Of the 134 Adversarial samples, 18 are predicted as Human and 17 as AI. Human confusion (13%) indicates that style-mimicking attacks remain partially effective

Language	Macro F_1
PHP	0.5541
Go	0.7488
C++	0.8011
C#	0.8252
JavaScript	0.8344
Java	0.8610
Python	0.8899
C	0.8956

Table 17: Language-wise macro F_1 for Model C1 on Subtask C (development set).

Label	Precision	Recall	F_1	Support
Human	0.9393	0.9783	0.9584	554
AI	0.8186	0.8114	0.8150	228
Hybrid	0.8000	0.7143	0.7547	84
Adversarial	0.8115	0.7388	0.7734	134
Macro Avg	0.8424	0.8107	0.8254	1000
Weighted Avg	0.8830	0.8860	0.8838	1000

Table 18: Classification report for Model C2 on Subtask C (development set).

even after logit-level re-scoring; these samples likely exhibit surface statistics indistinguishable from human-authored code. AI confusion (13%) reflects cases where the attack succeeds in removing human stylistic markers without fully adopting an AI signature recognizable by the meta-classifier.

Language-wise trends. C achieves the lowest macro F_1 (0.622), possibly due to the small per-label sample counts in that language amplifying label noise. PHP (0.689) and JavaScript (0.737) are also below average, consistent with the cross-language variation observed for Model C1. Python benefits from the largest training support and achieves the highest score (0.862).

Overall, Model C2's error distribution closely parallels that of Model C1, indicating that the logit-level stacking stage does not substantially reorder the difficulty of individual examples. The primary bottleneck remains the ambiguity between AI, Hybrid, and Adversarial categories rather than the choice of final classifier.

A.16 Subtask C: Model Configuration and Imbalance Analysis

Effect of Weighted Loss. Table 20 compares UniXcoder performance with and without cost-sensitive learning via weighted cross-entropy.

Configuration	Macro F1
UniXcoder (3 epochs, unweighted)	0.5791
UniXcoder (3 epochs, weighted)	0.5978

Table 20: Impact of class-weighted loss on Subtask C performance (test set).

Weighted loss improves F_1 by 1.87%, confirming its effectiveness for the four-way classification task.

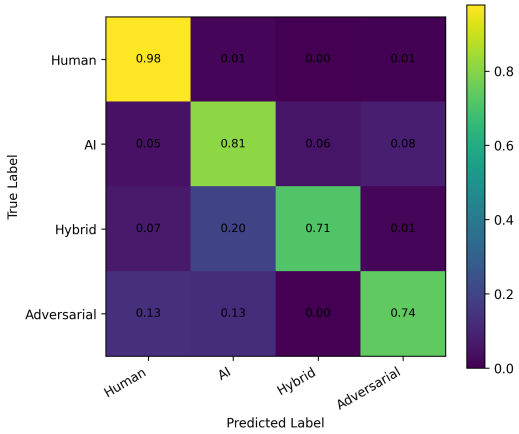


Figure 6: Row-normalized confusion matrix for Model C2 on Subtask C (development set). The matrix highlights confusion between AI and Adversarial classes, as well as overlap between AI and Hybrid categories.

Language	Macro F_1
C	0.6217
PHP	0.6887
JavaScript	0.7372
Go	0.7644
C++	0.7800
C#	0.8306
Java	0.8331
Python	0.8618

Table 19: Language-wise macro F_1 for Model C2 on Subtask C (development set).

Effect of Token Length, Model Size, and Training Duration. Table 21 presents an ablation study using ModernBERT variants in Subtask C, analyzing the effects of token length, model scale, and number of fine-tuning epochs.

Model (Tokens)	Epochs	Macro F1
ModernBERT-Base (512)	1	0.5987
ModernBERT-Base (1024)	1	0.5983
ModernBERT-Large (512)	1	0.5998
ModernBERT-Large (512)	3	0.6337
ModernBERT-Large (512)	4	0.6336

Table 21: Ablation study of ModernBERT configurations on Subtask C (test set).

Increasing model capacity and training duration yields substantial improvements, while extending the token length from 512 to 1024 provides negligible gains. Performance plateaus beyond three epochs, consistent with early convergence.