Segmentation Fault at SemEval-2026 Task 13: A Regularization-First Approach with Generator-Based Out-of-Distribution Splits for Detecting AI-Generated Code

Lakshmi Priya Swaminatha Rao, Dhannya Santhakumari Madhavan, Sreya Kodeswaran, Nithila R, Kanmani R
Department of Computer Science and Engineering
Sri Sivasubramaniya Nadar College of Engineering, Chennai, India
{lakshmipriyas,dhannyasm,nithila2410205,
kanmani2410090,sreya2410187}@ssn.edu.in

Abstract

This paper describes the system for SemEval-2026 Task 13 Subtask A on detecting AI-generated code. The proposed approach fine-tunes CodeBERT-base with a generator-based out-of-distribution validation split, holding out some model families to better match the test conditions. Heavy regularization like data augmentation, dropout, weight decay, and label smoothing is applied, training on a balanced 100K sample subset with early stopping based on validation trend monitoring. Four approaches are experimented with - logistic regression, UniXcoder, vanilla CodeBERT, and the final regularized system. The results suggest that data quality and evaluation design play an important role in OOD generalization, and the final system achieves 0.439 macro F1 score on the test dataset, which is a 62% improvement over the initial approaches. Further analysis of the validation-test gap that is still present attributes it to human code source homogeneity, which is still a challenge and requires future work.

1 Introduction

SemEval-2026 Task 13 (Orel et al., 2026) addresses the detection of machine-generated code which is a challenge intensified by the rapid evolution of Large Language Models (LLMs) (Pan et al., 2024). Participation in Subtask A involves distinguishing human-written from AI-generated code across a diverse set of programming languages. The training set provided by the task had seen languages like Python, Java and C++, while the evaluation phase had unseen languages like Go, PHP, C#, C and JavaScript to test out-of-distribution (OOD) generalization. This task becomes important for two key reasons: first, in educational settings, it is necessary to distinguish synthetic code in education platforms (Becker et al., 2023) because if not, it might affect students downstream when they enter the workplace. There is also a concern on open-source repositories, as it can raise copyright and licensing questions, since LLM outputs are trained on existing code.

The main strategy shifts the focus from model scale to evaluation integrity and data regularization. A significant generalization gap was observed between validation and test performance. This is not because of a flaw in the standard random split, which is designed for in-distribution evaluation, but because the test setting involves unseen generator families. Under the standard split, models are validated on the same generators seen during training, leading to high validation scores that do not reflect performance on unseen generators in test data.

To better match the test conditions, we introduce a generator-based out-of-distribution (OOD) validation split, where entire generator families are held out during training. This creates a more challenging but realistic evaluation setup for model selection.

Through participation in this task, it was observed that raw data volume appears to be less significant than data diversity and evaluation rigor in our experimental setting. The system achieved a 62% relative improvement in Test F1 (0.27 to 0.439) by training on a fraction of the available data but with significantly higher regularization. The final system achieved a Macro-F1 score of 0.439 on the hidden test set, placing 55th on the official leaderboard. Qualitatively, the system excels at identifying structural AI patterns across both seen and unseen languages, but struggles with short, boilerplate snippets that lack distinct stylistic markers. The code is available at: https://github.com/Sreya889/Semeval-Task-13-Subtask-A.

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2108–2113

2 Background

2.1 Task Definition

SemEval-2026 Task 13 is a binary classification task: given a code snippet, determine whether it was written by a human (label 0) or generated by an AI model (label 1). The task covers multiple programming languages including Python, Java, C++, and JavaScript, and systems are evaluated using macro F1 score, which averages the F1 score for each class separately. This means a system cannot simply predict one label for everything and score well.

2.2 Dataset

The dataset (Orel et al., 2026, 2025b,a) contains around 600K labeled samples - 500K for training, 100K for validation, and 500K test samples with labels withheld. The AI - generated samples were generated by 35 different AI models from nine families: Qwen, Phi, CodeLlama, Llama, DeepSeek, Yi, StarCoder, Granite, and CodeGemma, ranging from small models like Llama-3.2-1B to large ones like Llama-3.3-70B-Instruct. The wide range of generators is deliberate - the real challenge is handling AI models that were not present during training, which reflects real-world conditions where new models are released continuously.

2.3 Why Detection Is Difficult

The main issue with AI-generated text detection is that models trained on one generator's outputs tend to fail on others. Rather than learning general features of AI-generated text, these models pick up on surface-level patterns specific to the generators they were trained on - things like formatting habits or naming conventions (Sadasivan et al., 2025). Code makes this worse. Unlike natural language, code is tightly constrained by syntax. Standard patterns like loops, conditionals, and library calls appear frequently in both human and AI code, which reduces the useful signal available to a classifier. Short snippets are particularly difficult since there is simply not enough content to identify meaningful differences.

2.4 Pre-trained Models

The system is built on CodeBERT (Feng et al., 2020), a RoBERTa-based model pre-trained on CodeSearchNet (Husain et al., 2019) across six programming languages. Experiments also included UniXcoder (Guo et al., 2022), which additionally incorporates abstract syntax tree information during pre-training. Despite this extra structural knowledge, UniXcoder produced the same test performance as CodeBERT under the experimental conditions, which indicated the problem was not the model choice but the training setup.

3 System Overview

Four training approaches were implemented, where each addresses the limitations found in the previous attempt. All systems perform binary classification of code snippets as human-written (label 0) or AI-generated (label 1).

3.1 Model 1: TF-IDF Logistic Regression (Baseline)

This baseline extracts TF-IDF features from raw tokens and trains a logistic regression classifier (Alemerien et al., 2024). This system requires no GPU and serves to establish a benchmark against which other neural approaches are compared.

3.2 Model 2: UniXcoder Fine-Tuning

The second system fine-tunes UniXcoder-base by adding a classification head over the `[CLS]` token representation. Stratified sampling across both generator and label dimensions is used to ensure proportional representation of all 35 generators in training. Despite this, the system failed to generalise to the test set and produced a macro F1 score lower than the baseline. This is attributed to the model memorizing generator-specific surface patterns rather than learning the features that distinguish human code from AI code at a deeper level. Since UniXcoder was trained under different data volumes and split conditions from the final system, direct comparisons with Model 4 should be interpreted cautiously.

3.3 Model 3: Vanilla CodeBERT

To determine whether the failure in Model 2 was architecture-specific or data-volume-related, CodeBERT-base is fine-tuned on the full available training data (500K samples) with standard hyperparameters and no additional regularization.

This system produced identical results to Model 2. The similarity in results of two different architectures trained on different data volumes suggested that simply increasing model capacity or training data may not be sufficient to resolve the generalization issue in this setting. However, since these

factors were not isolated through controlled experiments, this observation should be interpreted with caution. The high validation scores reflected memorization of generator patterns rather than actual generalization, due to the presence of unseen generators in the test set, while the training and validation data contained the same 35 generators.

3.4 Model 4: Regularized CodeBERT with OOD Validation

The final system attempts to resolve the generalization failure identified in Models 2 and 3 through three design choices.

3.4.1 Generator-Based OOD Validation Split

To better align validation with the test setting, a generator-based split is constructed by holding out three model families exclusively for validation:

- **Training (22 generators):** Qwen, Phi, CodeLlama, StarCoder, Granite, CodeGemma
- **Validation (13 generators):** Llama, DeepSeek, Yi

No generator from the validation set appears in training. This directly simulates the test environment of unseen generators, allowing meaningful model selection during training. The held-out families were chosen to represent different model scales (1.3B to 70B parameters) and training paradigms.

Generator and Language Distribution: Before balancing, the dataset has 47.7% human-written code, with the remaining samples distributed across 35 AI generators, each contributing approximately 1–3%. The dataset is also heavily skewed toward Python (91.46%), with smaller proportions of C++ (4.68%) and Java (3.86%).

After balancing, the training set is constructed with an equal proportion of human and AI samples (50% each). The AI portion is uniformly distributed across the selected training generators, with each contributing approximately 2.38% of the total dataset. The language distribution remains almost unchanged after balancing (Python: 91.46%, C++: 4.70%, Java: 3.84%), since balancing is performed at the label and generator level and not at the language level.

In the generator-based OOD split, all samples from three families (Llama, DeepSeek, Yi) are held out for validation, ensuring that no generator overlap exists between training and validation.

3.4.2 Data Augmentation

Stochastic augmentation (Yu et al., 2022) is applied to training samples using three transformation types designed to prevent the model from relying on generator-specific surface signals - variable renaming (Wen et al., 2025), comment removal, and formatting normalization. These are applied stochastically to 80% of training samples. Because they operate at the lexical level and do not rely on language-specific parsers, they are computationally cheap and broadly applicable across programming languages.

Before

```
def calculateSum(
    listOfNums):
  totalValue = 0
  for item in listOfNums
      :
    totalValue += item
  return totalValue
```

After

```
def var_0(var_1):
  var_2 = 0
  for var_3 in var_1:
    var_2 += var_3
  return var_2
```

Table 1: Variable renaming: original identifiers (left) replaced with generic placeholders (right).

3.4.3 Model Regularization

Strong regularization (Ahmadian et al., 1998) is applied to CodeBERT-base beyond standard fine-tuning practice. Dropout is set to 0.3 (versus the usual 0.1), forcing the model to learn more distributed representations. Label smoothing with factor 0.15 replaces hard binary targets with soft targets, avoiding overconfident predictions. Weight decay of 0.02 further penalizes large weights. Training is stabilized through gradient clipping (0.5) and learning rate warmup (10% of steps), and length-based grouping which can encourage memorization is disabled.

Training is conducted on a reduced, balanced subset of 100K samples (50K human, 50K AI). Although validation F1 shows slight improvement beyond step 1,000, the gains are marginal while training loss continues to decrease. We therefore select step 1,000 as a conservative checkpoint to limit overfitting.[1]

4 Results

4.1 Overall Performance

The final system achieved 0.439 macro F1 on the test set, placing 55th on the leaderboard among

[1]Python 3.10, PyTorch 2.1.0, HuggingFace Transformers 4.36.0, NVIDIA T4 (Google Colab)

81 submissions. Table 2 shows how all four systems performed. Models 2 and 3 both scored 0.99 on validation but only 0.27 on the test. This happened with two different models and two different data sizes, so neither was the cause. Model 4 used a generator-based OOD split with stronger regularization and scored 0.439 on the test, showing that validation under the standard split reflects in-distribution performance, but not performance on unseen generators.

S.no	Model	Data	Val F1	Test F1
1	Logistic Reg.	100K	0.920	0.340
2	UniXcoder	150K	0.990	0.270
3	CodeBERT	500K	0.990	0.270
4	CodeBERT + OOD	100K	0.965	**0.439**

Table 2: Comparison of all systems on the official test set. Val F1 for System 4 is on the OOD split; for Systems 2–3 it is on the provided random split

4.2 Ablation Study

The improvement from Model 3 to Model 4 results from several changes applied together, including the generator-based OOD validation split, training on a smaller balanced dataset, and stronger regularization. Since these components were modified simultaneously, their individual contributions were not isolated experimentally.

However, some clear observations can still be made. First, the OOD validation split provides a validation setting that better matches the test condition. In Models 2 and 3, validation F1 is very high (0.99), but test F1 drops to 0.27, showing poor generalization. In Model 4, the validation F1 (0.965) is closer to the test F1 (0.439), suggesting that the new split helps to better simulate the test conditions.

Second, training on a smaller balanced subset (100K samples) appears to improve generalization compared to using the full dataset. This likely reduces the model's tendency to learn generator-specific patterns, although this effect is not isolated.

Data augmentation (variable renaming, comment removal, and formatting normalization) is used as part of the regularization strategy to reduce reliance on superficial patterns. While its individual impact is not measured, it likely contributes to the overall improvement.

Finally, as shown in Table 3, validation F1 improves only slightly after step 1,000 (0.9649 to 0.9662), while training loss continues to decrease, indicating diminishing returns and the onset of overfitting. This trend is further illustrated in Figure 1.

Step	Samples	OOD Val F1	Train Loss
1,000	48K	0.9649	0.336
2,000	96K	0.9662	0.316

Table 3: Checkpoint comparison on the OOD validation split. At step 2,000, training loss kept falling while validation F1 showed only marginal improvement — a sign of overfitting.

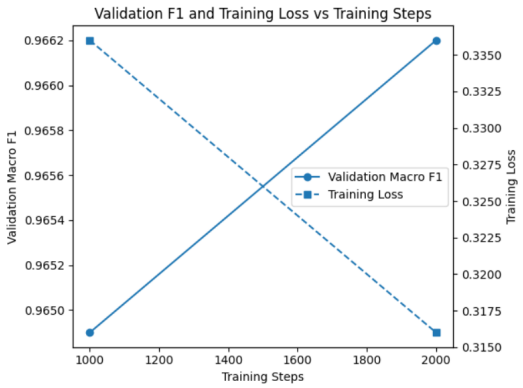

Figure 1: Validation macro F1 and training loss as a function of training steps. While training loss decreases from 0.336 to 0.316 between steps 1000 and 2000, validation F1 improves only slightly (0.9649 to 0.9662), indicating diminishing returns.

5 Conclusion

Participation in SemEval-2026 Task 13 (Subtask A) highlights that overfitting to generator-specific stylistic signatures appears to be a key challenge in this setting. By recognizing the difference between in-distribution validation and the out-of-distribution test setting, a transition was made to a robust OOD validation strategy. The findings demonstrate that training on a smaller, highly regularized subset of 100,000 samples using aggressive identifier obfuscation is significantly more effective than training on massive, unregularized datasets.

The system achieved a Macro-F1 score of 0.439, representing a 62% relative improvement over the initial unregularized baselines. The approach proved capable of generalizing across seen languages (Python, Java, C++) and unseen languages (Go, PHP, C#, C, and JavaScript). While the model excels at identifying structural patterns in complex logic, it continues to struggle with short, boilerplate snippets that lack distinct stylistic markers. Future

research should focus on integrating Abstract Syntax Tree (AST) features to further isolate code logic from surface-level syntax.

6 Limitations

The variable renaming uses regex rather than a language-specific parser, which can occasionally produce syntactically invalid output. The remaining gap between validation (0.965) and test (0.439) F1 suggests that the OOD split does not fully capture the test distribution. This is attributed partly to homogeneity in human code sources. The work was also limited to CodeBERT-base due to compute constraints; larger models or ensembles may perform differently. We note that all evaluated models are of similar scale, so conclusions about model capacity remain limited and should be explored in future work.

We analyze the effect of sequence truncation at 256 tokens. Approximately 43.94% of training samples and 45.62% of validation samples exceed this length and are truncated during tokenization. The mean sequence lengths are 386.9 and 420.1 tokens for training and validation respectively, with maximum lengths exceeding 37K tokens.

This indicates that a substantial portion of the data is truncated, which may lead to loss of long-range contextual information. However, increasing the maximum sequence length was not feasible due to computational constraints.

We did not conduct experiments with models of substantially different capacity due to computational and time constraints during the competition. Therefore, the effect of model scale on performance in this setting remains inconclusive in this work. The similar performance observed between CodeBERT and UniXcoder applies only to models of comparable scale and architecture, and should not be interpreted as evidence that model capacity is unimportant. This remains an open direction for future work involving both smaller and larger code models.

Acknowledgments

Thanks are extended to the organizers of the shared task for providing the dataset and evaluation framework. Computational resources were provided by Google Colab.

References

H. Ahmadian, J.E. Mottershead, and M.I. Friswell. Regularisation methods for finite element model updating. *Mechanical Systems and Signal Processing*, 12(1):47–64, 1998. ISSN 0888-3270. doi: https://doi.org/10.1006/mssp.1996.0133. URL https://www.sciencedirect.com/science/article/pii/S0888327096901338.

Khalid Alemerien, Aram Al-Ghareeb, and Malek Zakarya Alksasbeh. Sentiment analysis of online reviews: A machine learning based approach with tf-idf vectorization. *Journal of Mobile Multimedia*, 20 (5):1089–1116, 2024. doi: 10.13052/jmm1550-4646.2055.

Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard - or at least it used to be: Educational opportunities and challenges of ai code generation. SIGCSE 2023, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394314. doi: 10.1145/3545945.3569759. URL https://doi.org/10.1145/3545945.3569759.

Zhangyin Feng et al. 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, 2020.

Daya Guo et al. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pages 7212–7225. Association for Computational Linguistics, 2022.

Hamel Husain et al. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Daniil Orel, Dilshod Azizov, and Preslav Nakov. CoDet-m4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10570–10593, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-256-5. URL https://aclanthology.org/2025.findings-acl.550/.

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

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 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, June 2026. Association for Computational Linguistics.

Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. Assessing ai detectors in identifying ai-generated code: Implications for education. ICSE-SEET '24, page 1–11, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704987. doi: 10.1145/3639474.3640068. URL https://doi.org/10.1145/3639474.3640068.

Vinu Sankar Sadasivan, Aounon Kumar, Sriram Balasubramanian, Wenxiao Wang, and Soheil Feizi. Can ai-generated text be reliably detected?, 2025. URL https://arxiv.org/abs/2303.11156.

Jin Wen, Qiang Hu, Yuejun Guo, Maxime Cordy, and Yves Le Traon. Variable renaming-based adversarial test generation for code model: Benchmark and enhancement. 35(1), December 2025. ISSN 1049-331X. doi: 10.1145/3723353. URL https://doi.org/10.1145/3723353.

Shiwen Yu, Ting Wang, and Ji Wang. Data augmentation by program transformation. *Journal of Systems and Software*, 190: 111304, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2022.111304. URL https://www.sciencedirect.com/science/article/pii/S0164121222000541.

A Hyperparameters and Configuration

Hyperparameter	Value
Learning rate	1.5×10^{-5}
Batch size	16
Gradient accum. steps	3 (effective: 48)
Dropout	0.3
Weight decay	0.02
Label smoothing	0.15
Augmentation rate	0.80
Max sequence length	256 tokens
Warmup ratio	0.10
Mixed precision	fp16
Optimizer	AdamW
Training stopped at	Step 1,000

Table 4: Final system hyperparameters

Held-out generators: **Llama:** Llama-3.1-8B, Llama-3.1-8B-Instruct, Llama-3.2-1B, Llama-3.2-3B, Llama-3.3-70B-Instruct. **DeepSeek:** deepseek-coder-1.3b-base, deepseek-coder-1.3b-instruct, deepseek-coder-6.7b-base, deepseek-coder-6.7b-instruct. **Yi:** Yi-Coder-1.5B, Yi-Coder-9B, Yi-Coder-1.5B-Chat, Yi-Coder-9B-Chat.