Pixel Phantoms at SemEval-2026 Task 13: Exploring Classical and Neural Approaches for AI-Generated Code Detection

Janani Hariharakrishnan
Sri Sivasubramaniya Nadar College of Engineering
janani2210181@ssn.edu.in

Jithu Morrison S
Sri Sivasubramaniya Nadar College of Engineering
jithumorrison2210564@ssn.edu.in

Angel Deborah S
Sri Sivasubramaniya Nadar College of Engineering
angeldeborahs@ssn.edu.in

Rajalakshmi S
Sri Sivasubramaniya Nadar College of Engineering
rajalakshmis@ssn.edu.in

Abstract

This paper describes our system for SemEval-2026 Task 13, Subtask A: detecting whether a given code snippet is AI-generated or human-written. We explored a range of approaches from classical machine learning baselines using TF-IDF representations to fine-tuned transformer models pre-trained on code, specifically CodeBERT and GraphCodeBERT. Our experiments revealed a notable degradation in model performance when CodeBERT was trained beyond an optimal number of steps, indicating that continued training within an epoch leads to overfitting or representation drift. GraphCodeBERT, by contrast, yielded our best submission with a macro F1 score of 0.36866. Our findings highlight the sensitivity of code-specific transformers to training duration and suggest that early checkpoint selection is critical for this task.

1 Introduction

The rapid proliferation of large language models capable of generating syntactically correct and semantically plausible code has raised important questions around code authenticity, academic integrity, and software provenance. Distinguishing AI-generated code from human-written code is a challenging task, as modern code generators can closely mimic the stylistic and structural patterns of human programmers.

SemEval-2026 Task 13, Subtask A, addresses this problem directly by providing a binary classification dataset of code snippets labeled as either AI-generated or human-written. Following the official task overview released by the organizers, the goal is to build a system that accurately classifies unseen code samples (Orel et al., 2026).

Our approach involved two main directions. First, we implemented classical TF-IDF-based baselines using Logistic Regression and Linear Support Vector Classifiers (LinearSVC), exploring both character-level n-gram and word-level feature representations. Second, we fine-tuned transformer models that are pre-trained on code - CodeBERT and GraphCodeBERT - using the Hugging Face Transformers library. A key observation from our experiments was that CodeBERT's performance consistently declined when trained past early checkpoints within an epoch, with performance degrading at steps 5,000, 13,000, and 63,000 in sequence. GraphCodeBERT, which incorporates structural code information through data flow graphs, achieved our best result.

All experiments were run on Google Colab with GPU support. We report results across multiple submissions and discuss the patterns we observed.

In addition to overall performance, we compared error patterns between lexical and structural models to better understand where each representation helped most across snippet length, comment density, and stylistic variation.

2 Background and Related Work

AI-Generated Content Detection. Detecting machine-generated content has been studied extensively in natural language, where methods range from probability-based analysis and local surprisal cues to discriminative classifiers built on model or corpus statistics (Gehrmann et al., 2019; Mitchell et al., 2023; Hans et al., 2024).

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2040–2045

For code, the problem becomes more structurally constrained but no less challenging: Orel et al. (2025b) introduced DROID, a large-scale benchmark for AI-generated code detection spanning multiple programming languages and generator families, and found that fine-tuned transformer models consistently outperformed classical n-gram baselines on out-of-distribution generators. Their companion work (Orel et al., 2025a) further demonstrated that cross-lingual transfer between structurally similar languages - such as Java and C# - can substantially improve detection performance, while models trained on Python alone generalize poorly to lower-resource languages (Orel et al., 2025b,a). These findings motivate our choice to evaluate both lexical (TF-IDF) and structure-aware (GraphCodeBERT) representations: if surface-level lexical cues are sufficient for seen generators but fail on unseen ones, structural representations may provide the robustness needed for real-world deployment.

Code Representation Learning. Transformer models pre-trained on code corpora have demonstrated strong performance on tasks such as code summarization, code search, and bug detection. CodeBERT (Feng et al., 2020) extends the BERT architecture with training on bimodal data (code and natural language) from GitHub. GraphCodeBERT (Guo et al., 2021) extends this further by incorporating data flow graphs during pre-training, providing the model with structural understanding of how variables are defined and used. Other code-specific models and benchmarks, such as CodeT5, Code2vec, and CodeXGLUE, further support systematic evaluation of code understanding approaches (Wang et al., 2021; Alon et al., 2019; Lu et al., 2021). Recent pretraining efforts like PLBART, UniXcoder, and InCoder further demonstrate the breadth of code-focused representation learning and generation paradigms (Ahmad et al., 2021; Guo et al., 2022; Fried et al., 2022). These models are natural candidates for code classification tasks.

TF-IDF for Code. Character-level and token-level TF-IDF representations have been used effectively for code authorship attribution and software classification tasks (Caliskan-Islam et al., 2015; Frantzeskou et al., 2006). Character n-grams are particularly effective for capturing programming idioms and stylistic patterns at a sub-token level, while word-level n-grams can capture identifier usage patterns.

Our work situates itself in the intersection of code understanding and AI content detection, building on these established representations while identifying important failure modes when fine-tuning code transformers on limited compute budgets and under submission-time constraints.

Compared to prior SemEval tasks centered on text or multilingual classification, this task emphasizes code-specific artifacts such as indentation, delimiter usage, and identifier naming conventions, which make structural modeling particularly relevant in low-resource and heterogeneous settings.

3 System Overview

We developed and evaluated four distinct systems for Subtask A.

System 1 - Logistic Regression with Character TF-IDF (Light). We included this system as a fast, interpretable baseline before introducing learned code representations.

We extracted character n-gram features ($n = 3$ - 5) from code snippets using a TF-IDF vectorizer with 5,000 maximum features and trained a Logistic Regression classifier ($C = 2.0$, `liblinear` solver). This served as a lightweight, fast baseline to establish a performance floor.

System 2 - LinearSVC with Word TF-IDF (Strong TF-IDF). We replaced character n-grams with a word-level TF-IDF vectorizer using a code-aware token pattern (`[A-Za-z_][A-Za-z0-9_]+`), with 20,000 features, unigram-to-trigram n-grams, and sublinear TF scaling. A LinearSVC ($C = 1.0$, balanced class weights) was trained on this representation. A variant with 200,000 features and character n-grams ($n = 3$ - 6) was also evaluated.

System 3 - Fine-tuned CodeBERT. CodeBERT was chosen as a strong transformer baseline because its bimodal pre-training on code and natural language docstrings from GitHub makes it well-suited to capture surface-level code patterns with-

out requiring task-specific architectural changes. We fine-tuned `microsoft/codebert-base` for binary sequence classification. Code was tokenized with padding and truncation to 256 tokens. We used a batch size of 8, a learning rate of 2×10^{-5}, a weight decay of 0.01, and we trained the model for 3 epochs. Checkpoints were saved every 5,000 steps. We observed that predictions from later checkpoints (13,000 and 63,000 steps) produced lower macro F1 than predictions from earlier checkpoints, suggesting progressive performance degradation with continued training.

System 4 - Fine-tuned GraphCodeBERT (Best System). We selected GraphCodeBERT as our primary neural system because its data-flow-aware pre-training encodes how variables are defined, used, and passed between functions - structural properties we hypothesized would differ systematically between AI-generated and human-written code. We fine-tuned `microsoft/graphcodebert-base` using the same hyperparameter setup as CodeBERT (256 tokens, batch size 8, $lr = 2 \times 10^{-5}$, 3 epochs) but with checkpoints saved every 2,000 steps. To avoid repeated tokenization, we cached the tokenized dataset to disk using Hugging Face's `DatasetDict.save_to_disk` API. GraphCodeBERT's data-flow-aware pre-training enabled it to capture structural patterns in code more effectively, and this system produced our best submission.

Across these systems, we prioritized reproducibility by keeping preprocessing and metric computation consistent. This allowed us to attribute performance differences to representation choices rather than evaluation variance.

4 Experimental Setup

Data. We used the official train, validation, and test parquet files provided for Subtask A of SemEval-2026 Task 13 (Orel et al., 2026). All three splits contained a `code` column with source code snippets and (for train/validation) a `label` column for binary classification (0 = human-written, 1 = AI-generated).

Classical Models. For TF-IDF systems, we processed the training data in batches of 5,000 samples using `scipy.sparse.vstack` to manage memory. The vectorizer was fit only on the training set and applied to validation and test sets via `transform`. Final evaluation used macro F1 and accuracy on the validation set.

Transformer Models. Fine-tuning was performed using the Hugging Face `Trainer` API. We enabled mixed-precision training (`fp16=True`) and gradient checkpointing to reduce memory usage. Training was run on Google Colab with a T4 GPU. The slow CodeBERT run (13 hr) used a max sequence length of 512 tokens; the faster run (4 hr) reduced this to 256 tokens with a batch size of 8.

Checkpoint Evaluation. A key part of our experimental procedure was evaluating predictions from multiple checkpoints of the CodeBERT model - specifically `checkpoint-5000`, `checkpoint-13000`, and `checkpoint-65000` - using a separate inference script that loaded each checkpoint and generated test predictions.

Submission. Test predictions were saved as CSV files with columns `ID` and `label` and submitted to the competition platform. Macro F1 on the test set was the primary metric.

We also monitored training and validation curves to detect divergence early and to identify unstable optimization behavior. When signs of overfitting appeared, we compared checkpoints using the same inference pipeline and preprocessing setup to ensure a fair comparison.

5 Results

Table 1 summarizes our main experimental runs and their macro F1 scores on the leaderboard. We focus our analysis on the strongest system, GraphCodeBERT, and use the remaining runs primarily as controlled comparisons.

Key Finding: CodeBERT Checkpoint Degradation. A striking pattern across our submissions is that CodeBERT-based predictions worsened as training progressed. The best CodeBERT result (F1 = 0.32976) corresponds to an earlier checkpoint. At `checkpoint-13000` (Run R6), the score drops to 0.28188, and at `checkpoint-65000` (Run R7), it further drops to 0.27664. This monotonically decreasing pattern across checkpoints suggests that the model was overfitting to the training distribution as training continued. We hypothesize

Run	Model	F1
R1	GraphCodeBERT	**0.36866**
R2	CodeBERT (early checkpoint)	0.32976
R3	Logistic Regression + character TF-IDF	0.32854
R4	LinearSVC + word TF-IDF	0.28933
R5	LinearSVC + word TF-IDF	0.28281
R6	CodeBERT (mid checkpoint)	0.28188
R7	CodeBERT (late checkpoint)	0.27664

Table 1: Results on the SemEval-2026 Task 13 Subtask A leaderboards.

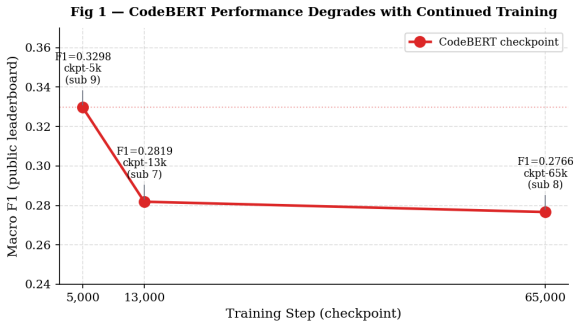

Figure 1: CodeBERT performance degrades with continued training.

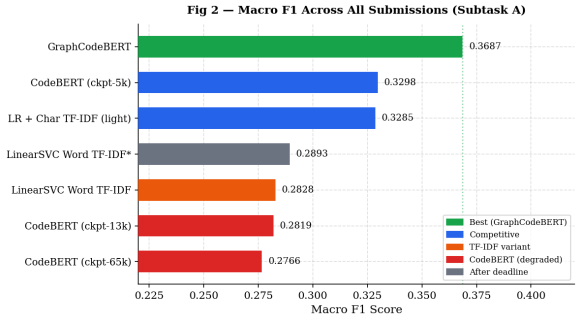

Figure 2: Macro F1 across all submissions (Subtask A).

that the high variability of code style in the training set makes extended fine-tuning prone to memorizing surface-level artifacts rather than learning generalizable patterns.

GraphCodeBERT vs. CodeBERT. GraphCodeBERT significantly outperformed CodeBERT at its best (0.36866 vs. 0.32976), an absolute gain of 0.03890 macro F1, and it also exceeded our strongest TF-IDF baseline by 0.04012. This margin suggests that the data flow graph information incorporated during GraphCodeBERT's pre-training provides a richer structural representation for this task. In our qualitative error analysis, that advantage was most visible on longer snippets and on examples with denser variable interactions or comments, where purely lexical cues were less reliable.

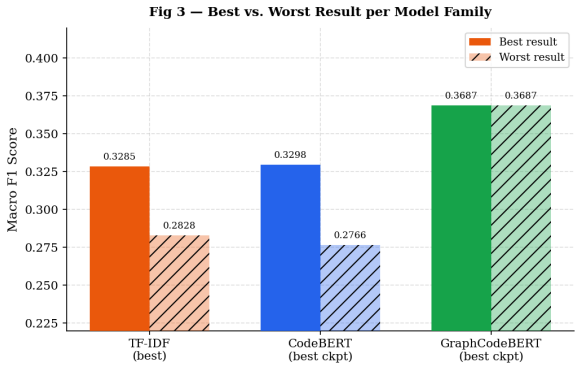

Figure 3: Best vs. worst result per model family.

TF-IDF Baselines. The lightweight Logistic Regression character TF-IDF model achieved 0.32854, competitive with our best CodeBERT submission. This is a notable result - a fast classical model with minimal tuning matched a large pre-trained transformer. The LinearSVC word TF-IDF models performed worse (0.28 - 0.29), potentially due to vocabulary mismatch or the word-token pattern failing to capture relevant code-specific patterns.

These findings suggest that surface-level lexical cues remain strong signals for AI-generated code, at least in the current benchmark dataset. Combining lexical and structural representations remains a promising practical next step, although we did not evaluate ensembles due to shared-task time and compute constraints.

5.1 Error Analysis

To better understand model failures, we performed a qualitative analysis of validation set predictions from our best system (GraphCodeBERT) and the lightweight TF-IDF baseline. We observed three recurring error patterns. First, both models struggled with very short code snippets (under 5 lines), where there is insufficient context to distinguish stylistic patterns. AI-generated code at this length

often resembles terse, idiomatic human code, making the classification boundary ambiguous. Second, the TF-IDF model tended to misclassify heavily commented code as human-written, likely because comments introduce natural-language tokens that shift the character n-gram distribution away from typical AI-generated patterns. GraphCodeBERT was less affected by this, suggesting its structural representations are more robust to comment density.

Third, we noticed that false positives (human code labeled as AI-generated) often involved highly templated or boilerplate code - such as getter/setter methods or standard exception handlers - which structurally resemble AI output. This suggests that template-heavy code is an inherent ambiguity in this task, and that future systems may benefit from explicitly modeling code originality or entropy.

6 Conclusion

We presented our system for SemEval-2026 Task 13 Subtask A, exploring TF-IDF baselines and fine-tuned code transformers for the task of distinguishing AI-generated from human-written code. Our contribution is empirical rather than architectural: we compare established lexical baselines and existing code-specific transformers under a shared evaluation setup rather than proposing a new model architecture or learning paradigm. Our best system used GraphCodeBERT and achieved a macro F1 of 0.36866 on the test set. A key finding was the consistent degradation of CodeBERT performance with continued training - earlier checkpoints (around 5,000 steps) outperformed later ones (13,000 and 65,000 steps) - highlighting the importance of early stopping and checkpoint selection when fine-tuning on this type of task.

Although our best result was competitive among our own submissions, it remains below the strongest systems on the shared-task leaderboard, indicating that substantial room for improvement remains. Future work could explore ensemble methods combining TF-IDF features with transformer representations, as the classical baseline proved surprisingly competitive. It would also be valuable to investigate why extended CodeBERT training hurts performance specifically on this task and to study when structural cues are most helpful.

Finally, collecting more balanced data across programming languages and generator families would likely improve generalization in real-world settings.

Ethical Considerations

This work aims to detect AI-generated code, which has legitimate applications in academic integrity, code auditing, and software provenance tracking. However, such systems could also be misused - for example, to flag human-written code incorrectly or to unfairly penalize developers who use AI tools as productivity aids. Like all classifiers, our models carry inherent biases. The training data may over-represent particular programming languages, coding styles, or AI code generators, limiting generalizability. Models may also perform differently across demographic groups of programmers or across programming languages not well represented in the training data. We encourage responsible deployment of such systems with human oversight, especially in high-stakes contexts such as academic evaluation.

We also note that evaluation on leaderboard data does not guarantee robustness against adaptive adversaries. Any deployment should include periodic auditing and transparency about model limitations.

A Additional Discussion and Limitations

Our main paper emphasizes the empirical comparison between lexical baselines and code-focused transformers. The strongest evidence we have for CodeBERT degradation comes from the checkpoint trajectory itself: macro F1 falls from 0.32976 at the early checkpoint to 0.28188 at the mid checkpoint and 0.27664 at the late checkpoint, a total drop of 0.05312. Under our single learning-rate setting (2×10^{-5}) and fixed batch size, this monotonic decline is more consistent with overfitting than with simple random variance. We therefore view lower learning rates, fewer update steps, and stronger regularization as the most plausible interventions, though confirming that claim requires targeted ablations and full training-versus-validation curves.

As a natural extension of this work, a TF-IDF+transformer ensemble could combine complementary lexical and structural signals. We did not include such experiments in the final

submission because the shared-task deadline and our available Colab GPU budget forced us to prioritize single-model runs first. Even so, the strength of both GraphCodeBERT and the lightweight TF-IDF baseline suggests that the two representations capture partially complementary evidence.

Finally, GraphCodeBERT improved over the best CodeBERT run by 0.03890 macro F1 and over the lightweight TF-IDF baseline by 0.04012, indicating a structural advantage in practice. In our manual error review, the clearest gains appeared on longer snippets, comment-heavy samples, and examples with variable reuse and more complex control flow.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 40:1–40:29.

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

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shen, 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*.

Georgia Frantzeskou, Stephen G. MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. 2006. Source code author identification based on n-gram author profiles. *Artificial Intelligence Applications and Innovations*, pages 508–515.

Daniel Fried, Abhijit Jain, Iz Beltagy, Dan Orlanski, Daniel Khashabi, Wen-tau Yih, Andreas Stolcke, Huaiyu Zhao, and Noah A. Smith. 2022. Incoder: A generative model for code infilling and synthesis. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander M. 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.

Daya Guo, Shuai Lu, Shuo Ren, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, and Nan Duan. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shuai Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Abhimanyu Hans, Avi Schwarzschild, Valeriia Cherepanova, Micah Goldblum, Jonas Geiping, and Tom Goldstein. 2024. Binoculars: Zero-shot way to distinguish between human-written and machine-generated text. *arXiv*, arXiv:2401.12070.

Sheng Lu, Daya Guo, Shuo Ren, Nan Duan, Neel Sundaresan, Vishal Udandarao, Yuning Ma, Abderrahmane Zghidi, Param Raja, and Anjana Krishna. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv*, arXiv:2102.04664.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D. Manning, and Chelsea Finn. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. *Proceedings of the 40th International Conference on Machine Learning*.

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. 2026. Semeval-2026 task 13: Detecting machine-generated code with multiple programming languages, generators, and application scenarios. Official SemEval-2026 Task 13 overview page.

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.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.