Königsberg at SemEval-2026 Task 13: Beyond Language Models: A Low-Resource Feature-Driven and Data-Flow Embedding Approach for Machine-Generated Code Detection

Shahir Habib and Dipankar Das
Department of Computer Science and Engineering
Jadavpur University, Kolkata, India
{shahirhabib4, dipankar.dipnil2005}@gmail.com

Abstract

The rise of Large Language Models (LLMs) has increased the need for reliable detection of machine-generated code. This paper presents a low-resource, hybrid detection framework developed for for SemEval-2026 Task 13 ,designed to operate efficiently without the computational overhead of end-to-end fine-tuning of large models. Our approach combines (i) comprehensive feature extraction pipeline that calculates interpretable software metrics capturing stylistic and structural properties of code, and (ii) we leverage the semantic capabilities of GraphCodeBERT by extracting frozen embeddings from its pre-trained encoder to model semantic and data-flow information while preserving generalizability. This fusion enables efficient detection of machine-generated code across multiple programming languages (Python, C++, Java, and Go) and improves robustness under out-of-distribution settings. This feature-driven fusion offers a competitive, computation-efficient alternative to purely LLM-based fully fine-tuned models, achieving an F1-score of 38.26.

1 Introduction

The rapid advancement of Large Language Models (LLMs) in software engineering has introduced a critical challenge: accurately distinguishing between human-written and machine-generated code. SemEval-2026 Task 13 (Orel et al., 2026b) addresses this pressing need by establishing a rigorous shared task for machine-generated code detection. Solving this problem is essential for maintaining academic integrity, ensuring software reliability, and preventing the subtle vulnerabilities often embedded in AI-generated scripts.

Our system's primary strategy relies on a dual-pronged, low-resource hybrid approach designed to avoid the prohibitive computational costs of end-to-end LLM fine-tuning. We synthesize interpretable feature engineering with deep, structural representations. First, we implemented a robust feature extraction pipeline that computes a wide array of statistical and structural software metrics directly from the source code. Second, we fused these handcrafted features with frozen, data-flow-aware embeddings extracted from the pre-trained Graph(Guo et al., 2020) model. By keeping the base model frozen, our methodology efficiently captures both surface-level stylistic markers and the underlying semantic logic without updating the heavy model weights.

Our feature analysis revealed significant disparities in traditional software metrics: human code generally exhibited higher variance in comment ratios and lines of comment, reflecting organic development processes and ad-hoc documentation. Conversely, machine-generated code often demonstrated more uniform cyclomatic and cognitive complexity , alongside distinct statistical signatures in Halstead metrics (Halstead, 1977), suggesting a highly predictable, algorithmic generation pattern. While our system effectively captured these static features, it occasionally struggled with highly iteratively prompted or human-edited AI code, where these structural boundaries blur and those of out-of-distribution (OOD) test samples.

We foster reproducibility and encourage further research into low-resource code detection bymaking our complete feature extraction and embedding fusion pipeline publicly available. The code will be released and can be accessed via our Github page.[1]

2 Related Work

Early supervised detectors (small-scale datasets). (Nguyen et al., 2023) introduced one of the first systematic attempts at binary code detection using a small 7K-sample dataset and a

[1] https://github.com/Shahir-habib/SemEval-Task-13-2026-Machine-Generation-Code-Detection

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 1288–1297
July 3-4, 2026

CodeBERT-based (Feng et al., 2020) classifier. The implementation used a pretrained CodeBERT encoder fine-tuned on binary labels (human vs. machine) and standard cross-entropy loss. Its main limitations include a tiny scale, narrow language mix (typically one language per split), and evaluation restricted to in-distribution splits. This leads to optimistic reported performance that does not transfer to unseen languages or generators.

Multi-language benchmarking (CodeMirage). (Guo et al., 2025) built a 210K multi-lingual benchmark spanning 10 languages and 10 production LLMs. This work evaluated detectors across zero-shot, fine-tuned, embedding-based, and paraphrase-augmented settings, analyzing paraphrase-robust pipelines and detector calibration under low false-positive budgets. Although broader than prior datasets, detectors still show dramatic drops in true positive rates when constrained to practical low false-positive rates. Furthermore, many detectors are saturated on the benchmark's simplified binary task, masking real-world hybrid or adversarial scenarios.

Zero-shot perturbation-based detection. Moving beyond supervised models, efforts like DetectGPT (Mitchell et al., 2023) introduced zero-shot detection leveraging the observation that LLM-generated text typically occupies negative curvature regions of the source model's log-probability function. While highly effective for natural language without requiring a fine-tuned classifier, adapting these perturbation-based metrics to source code has proven challenging. The strict syntax of programming languages means that random perturbations often break compilability, skewing probability scores and reducing detection efficacy on realistic codebases.

Unified, large-scale evaluation (AICD Bench). AICD Bench (Orel et al., 2026a) is the most comprehensive recent attempt, featuring 2M samples, 77 generators across 9 languages, and three tasks: robust binary classification, model-family attribution, and fine-grained human–machine classification. Standardized data splits create progressively more challenging out-of-distribution (OOD) folds to evaluate classical and neural detectors like UniXcoder (Guo et al., 2022). Key failures identified by the study include: (1) detector performance collapses under realistic OOD shifts, (2) fine-tuned classifiers fail to generalize to unseen generator families, and (3) hybrid/adversarial samples drastically reduce usable F1/TP rates.

Task	Split	#Samples	Languages
Task 1	Train	500K	Python, Java, C++
Task 1	Validation	100K	Python, Java, C++
Task 1	Test	1M	Python, Java, C++, C, Golang, PHP, C#, Js

Table 1: Data distribution for Subtask A (binary code classification). The test split includes additional unseen languages to evaluate robustness.

Synthesis and gap analysis for SemEval Task

- Most prior works focus on *in-distribution binary* classification or on limited OOD axes; SemEval Task 13 (Orel et al., 2026b) focuses on OOD splits across *language, generator family,* and *domain* simultaneously to expose realistic failure modes (recommended by AICD Bench).

- Evaluation metrics must include low-FPR operating points and family-attribution accuracy in addition to F1 to surface deployment-relevant weaknesses (CodeMirage, AICD Bench findings).

3 Data

3.1 Task Definition

SemEval 2026 Task 13 (Subtask A) (Orel et al., 2026b)formulates AI code detection as a **binary classification** problem. Given a source-code snippet, the system must determine whether it is human-written or AI-generated.

Input: raw code snippet in one of several programming languages.

Output:

- `0`: human-written

- `1`: AI-generated

3.2 Subtask A Data Split

Table 1 shows the data distribution for Subtask A. The test set introduces additional programming languages to explicitly evaluate cross-language generalization, following the robustness goals of AICD Bench. (Orel et al., 2026a)

The dataset is balanced between human and machine-written code and is split to induce controlled distribution shifts. (Orel et al., 2026a) AICD Bench introduces several important advances over prior resources:

- **Diversity in Splits:** significantly larger generator and language coverage than earlier datasets. (Orel et al., 2026a)
- **Explicit OOD evaluation:** benchmark splits test cross-language and cross-domain generalization.
- **Realistic difficulty:** includes challenging hybrid and adversarial cases.

3.3 Dataset Source

The SemEval data is derived from **AICD Bench**, currently the largest unified benchmark for AI-generated code detection, containing roughly **2M samples** spanning human and machine-generated code. (Orel et al., 2026a)

AICD Bench was constructed to stress-test detectors under realistic conditions and includes:

- 77 generators from 11 model families
- 9 programming languages
- Multiple code domains

Prior work showed that many detectors that perform well in-distribution degrade significantly under these settings.

3.4 Novel Aspects of Our Method

The robustness challenges identified by AICD Bench are addressed through a **novel dual-pronged low-resource hybrid framework** that fundamentally departs from conventional end-to-end fine-tuning approaches. Unlike prior methods that rely solely on large-scale model adaptation, our approach uniquely combines (i) handcrafted statistical and structural software metrics, providing interpretable surface-level signals, with (ii) frozen, data-flow-aware embeddings from GraphCodeBERT to capture deep semantic structure without additional training overhead. This principled fusion of complementary feature spaces—explicit and latent—constitutes the core innovation of our method, enabling robust, generalizable detection across heterogeneous and out-of-distribution settings while remaining computationally efficient.

4 Methodology

Our system adopts a **dual-pronged, low-resource hybrid architecture** designed to avoid the high computational cost of full end-to-end LLM fine-tuning while maintaining strong semantic sensitivity. The pipeline consists of two complementary components: (i) handcrafted statistical–structural feature modeling and (ii) frozen GraphCodeBERT semantic embeddings. The final prediction is obtained via an ensemble of both views.

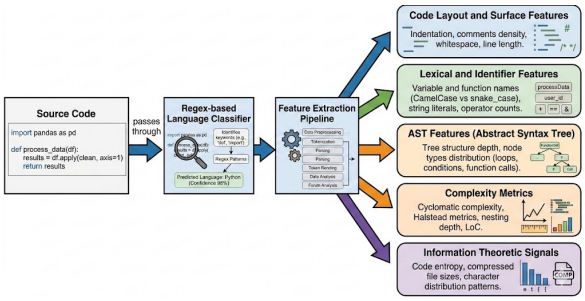

Figure 1: Overview of the proposed dual-pronged feature extraction.

Figure 1 illustrates the overall feature extraction pipeline.

4.1 Approach I: Language-Aware Statistical and Structural Feature Modeling

As a preprocessing step, we first infer the programming language of each test sample using a lightweight **regex-based language detector**. The detector relies on language-specific lexical signatures (e.g., `def`, `import`, `package`, `func`) and assigns the most likely language label.

Given the predicted language, we apply the corresponding language-specific parsing and feature extraction pipeline. This design avoids expensive multilingual parsing while ensuring syntactic correctness of downstream structural features.

4.2 Handcrafted Feature Categories

We organize our handcrafted signals into five complementary feature groups, each capturing distinct properties of source code.

4.2.1 Code Layout and Surface Statistics

These features characterize formatting regularity and stylistic surface patterns.

loc_total, **loc_comments**, **loc_blank**, **comment_ratio** , **avg_line_length** , **max_line_length**, **indentation_std_dev**

4.2.2 Lexical and Identifier Statistics

This group captures vocabulary diversity and naming conventions.

unique_identifier_ratio avg_identifier_length snake_case_ratio keyword_density hapax_legomena_count: identifiers appearing exactly once

4.2.3 AST Structural Features

These features model syntactic program structure using the abstract syntax tree (AST).

ast_node_count, **ast_max_depth**, **ast_avg_branching_factor**, **control_flow_ratio**, **exception_handling_ratio**, **return_statement_density** , **syntactic_ngram_frequency**: frequency of parent→child AST patterns (e.g., For→If)

4.2.4 Complexity and Maintainability Metrics

We compute classical software engineering metrics that quantify structural difficulty.

cyclomatic_complexity: McCabe complexity (McCabe, 1976) **halstead_volume**: information content in bits **halstead_difficulty**: estimated implementation difficulty **halstead_effort**: volume × difficulty **maintainability_index cognitive_complexity**: nesting-aware complexity metric

4.2.5 Information-Theoretic Signals

These features capture distributional uncertainty often exhibited by machine-generated code.

token_entropy: Shannon entropy of tokens **ast_node_entropy**: entropy of AST node types **perplexity_score**: proxy-model perplexity (Jelinek et al., 1977) of the code. **burstiness**: variance of perplexity across the snippet

These features are lightweight to compute and provide strong signals for stylistic artifacts often present in machine-generated code. However, they may fail when generators mimic human style or under heavy distribution shift. Please refer B

4.3 Feature Selection and Dimensionality Reduction

The full handcrafted feature space exhibits substantial redundancy due to inter-feature correlation and overlapping statistical signals. To obtain a compact and robust representation, we design a **multi-stage feature selection pipeline** that identifies the most informative subset of features.

Stage 1: Standardization and stratified evaluation. All candidate features are standardized using `StandardScaler`. We employ `StratifiedKFold` cross-validation (Kohavi, 1995) to ensure label balance during evaluation and to prevent selection bias.

Stage 2: Model-based importance estimation. We train multiple lightweight classifiers—including Logistic Regression, Random Forest, MLP, and gradient-boosted variants—under hyperparameter tuning via `RandomizedSearchCV`. Tree-based models provide intrinsic feature importance scores, while linear models capture sparse discriminative signals. **Statistical Ranking for Final Feature Selection**

For further improving the robustness and reduce model-specific bias, we complement model-based importance with **filter-based statistical ranking**. This step explicitly accounts for feature–label dependency and inter-feature redundancy.

Ranking criteria. Each candidate feature is evaluated using three independent statistics:

Chi-Square Rank(Pearson, 1900): measures dependence between each feature and the binary label, highlighting discriminative categorical or frequency-based signals. **Mutual Information (MI) Rank (Shannon, 1948):** estimates nonlinear dependency between features and labels, capturing information gain beyond linear correlation. **Correlation Rank(Pearson, 1895):** computed via Pearson correlation (absolute value) to quantify linear association with the target variable.

Rank aggregation. For every feature, we compute its rank under each criterion and aggregate the scores to obtain a *consensus importance ranking*. This multi-view ranking mitigates the weaknesses of any single statistical test.

Based on the aggregated ranking, we retain the top **15 most informative features**, which provide the best trade-off between predictive performance and dimensional efficiency. Please refer A

4.4 Approach II: Frozen GraphCodeBERT Embeddings

The second branch complements surface statistics by extracting **deep semantic representations** using GraphCodeBERT.We employ `microsoft/graphcodebert-base`, a data-flow-aware transformer pretrained on large-scale code corpora. Unlike plain token models, GraphCodeBERT incorporates semantic relationships between variables and program structure.

Embedding extraction. Code snippets are tokenized with maximum length 512, the frozen encoder produces contextual token embeddings, we apply mean pooling over the last hidden states to obtain a fixed-length vector.

Crucially, **the backbone remains frozen**. This

design avoids expensive fine-tuning on large datasets, reduces overfitting to seen generators, preserves general semantic knowledge.

Downstream classifier. The pooled embeddings are standardized and fed into lightweight classifiers (e.g., logistic regression), enabling efficient training.

4.5 Complementarity Analysis

The two branches capture orthogonal signals, with feature engineering focusing on stylistic and statistical artifacts, while GraphCodeBERT captures semantic and data-flow structure. Empirically, embedding-only models can miss shallow stylistic cues, while handcrafted features alone lack deep semantic understanding. This motivates a fusion strategy.

4.6 Ensemble Fusion

Our final system uses an **ensemble-based late fusion** of both branches. **Procedure.** The statistical-feature classifier is trained alongside the embedding-based classifier, and their prediction probabilities are combined via weighted averaging. Let p_s denote the statistical model probability and p_g the GraphCodeBERT model probability. The final prediction is defined as:

$$p_{\text{final}} = \alpha p_s + (1 - \alpha) p_g$$

where α is tuned on the validation set. where α is tuned on the validation set. **Rationale.** This low-cost ensemble improves robustness under distribution shift while keeping training efficient, directly addressing the challenges highlighted in AICD Bench.

4.7 Efficiency Considerations

The proposed design is explicitly **low-resource**: no full LLM fine-tuning, frozen transformer backbone, lightweight classical classifiers, and a precomputable feature pipeline. This makes the system scalable to the multi-million sample regime of SemEval Task 13 while maintaining competitive performance.

4.8 Implementation Details

We ensure reproducibility by explicitly describing key components of our pipeline. Perplexity is computed using a statistical n-gram language model, specifically a trigram model ($n = 3$), with add-one (Laplace) smoothing to estimate token probabilities and compute cross-entropy/perplexity.

Contextual embeddings are obtained from a frozen transformer encoder (GraphCodeBERT), producing vectors of size 768. These embeddings are mean-pooled across tokens to obtain fixed-length representations. To reduce dimensionality, we apply feature truncation by retaining only the first 25 dimensions of each embedding vector ($X = X[:, : 25]$). This is a simple, non-parametric dimensionality reduction approach without any learned projection.

For feature fusion, we use a weighted combination of features, where the fusion weight is empirically set to $\alpha = 0.7$ based on cross-validation.

Language identification is performed using a regex-based detector that captures script-level patterns . For unseen languages, we employ a fallback parsing strategy based on whitespace tokenization and Unicode normalization without requiring language-specific rules. Finally, we ensure accessibility of our implementation by providing the code for below experiments available at Github page.[2]

We evaluate both branches of our framework—(i) handcrafted features and (ii) frozen GraphCodeBERT embeddings—using a suite of standard classifiers. For the statistical feature pipeline, we train Logistic Regression (LR), Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF), CatBoost, XGBoost, and a shallow Artificial Neural Network (ANN) on the provided train/val/test data splits.

The same set of classifiers is applied to the GraphCodeBERT embedding representations to ensure a fair comparison. All models are tuned via validation data, and performance is reported on the official test split. The final system additionally evaluates the proposed ensemble fusion based on F1 because when dealing with imbalanced datasets, the F1 score becomes especially important because it balances precision and recall into a single metric. For details about preprocessing, hyperparameter tuning refer appendix C

5 Results

Table 5 reports the performance of different models on Subtask A. Overall, tree-based ensembles and boosted models perform strongly for the seen domain and languages on handcrafted features, while embedding-based models capture complementary semantic signals. The proposed ensemble achieves

[2] https://github.com/Shahir-habib/SemEval-Task-13-2026-Machine-Generation-Code-Detection

Model	Seen Domain and Language	Unseen Domain and Language
Logistic Regression	86.55	36.44
SVM	86.54	36.78
Decision Tree	88.2	**38.26**
Random Forest	90.3	37.76
XGBoost	93.7	37.17
CatBoost	93.6	37.91
ANN	90.21	–

Table 2: Performance comparison (F1 Score) of different models on Subtask A.

the best overall result. Under the official Subtask A evaluation setting, the system obtains an F1 score of **38.26**. Ultimately, our system achieved a ranking of 65 in the final leaderboard surpassing the CodeBert, highlighting both the viability of our low-resource feature-fusion approach and the intense competitiveness of the task.

Post-submission Update. After submission, we observed a modest improvement in performance by reducing the embedding dimensionality via feature truncation (retaining the first 25 dimensions). This simplified representation improved generalization while maintaining efficiency. The updated results are reported in the Appendix D for completeness.

6 Error Ananlysis and Conclusion

We presented a low-resource hybrid framework for AI-generated code detection t,where the proposed ensemble demonstrates reasonable efficiency, it does not surpass fully fine-tuned large language model (LLM) detectors. This gap is expected: end-to-end LLM systems can internally model deeper semantic, stylistic, and long-range dependencies that are only partially captured by frozen embeddings and engineered features.

Error analysis indicates that most failures occur in the following scenarios:

(i) *Highly polished AI-generated code* that closely mimics human stylistic patterns, making surface-level and structural distinctions less effective.

(ii) *Short snippets with limited structural signals*, where insufficient context reduces both statistical and semantic discriminability. For example:

Example: Short Snippet

```
int add(int a, int b)  return a + b;
```

(iii) *Cross-language out-of-distribution cases*, where handcrafted statistics fail to generalize across syntactic and stylistic differences. For example:

Example: Cross-Language Snippet (Go)

```
package main
import "fmt"
func greet(name string) string  return
"Hello, " + name
func                                main()
fmt.Println(greet("World"))
```

Example: Unusual Code Snipit OOD Case

```
p = int(input()) print(['8' * (p // 2) +
'9' * (p % 2), -1][p > 36])
```

We also observe a domain sensitivity effect: since training data is concentrated in specific domains, performance degrades when evaluated on unseen domains. We hypothesize that extending the same feature extraction and fusion pipeline to a more diverse, multi-domain training corpus would likely improve generalization and yield performance closer to the seen-domain setting.

Despite these limitations, our approach offers an attractive trade-off between computational cost, interpretability, and robustness.

Future work will explore parameter-efficient fine-tuning (e.g., adapters or LoRA), stronger domain diversification during training, improved language-agnostic normalization, and more principled fusion strategies to further close the gap with large end-to-end LLM detectors.

References

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of EMNLP*.

Daya Guo, Shaohua Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Nan Duan, and 1 others. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of ACL*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

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. *arXiv preprint arXiv:2506.11059*.

Maurice H. Halstead. 1977. *Elements of Software Science*. Elsevier North-Holland, Inc.

Frederick Jelinek, Robert L. Mercer, Lalit R. Bahl, and Janet K. Baker. 1977. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1):S63–S63.

Ron Kohavi. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 1137–1143. Stanford, CA.

Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. DetectGPT: Zero-shot machine-generated text detection using probability curvature. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 24950–24962. PMLR.

Hoang-Dung Nguyen, Quang Duc Nguyen, Cong-Duy Nguyen, Phong To, Danh Nguyen, Huy Nguyen-Gia, Long Tran, Anh Tran, An Dang-Hieu, Anh Nguyen Duc, and Tho Quan. 2023. Supervised learning models for social bot detection: Literature review and benchmark. *Expert Systems with Applications*, 238:122217.

Daniil Orel, Dilshod Azizov, Indraneil Paul, Yuxia Wang, Iryna Gurevych, and Preslav Nakov. 2026a. Aicd bench: A challenging benchmark for ai-generated code detection. *arXiv preprint arXiv:2602.02079*.

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.

Karl Pearson. 1895. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242.

Karl Pearson. 1900. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine*, 50(302):157–175.

Claude E. Shannon. 1948. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423.

Appendix

A Feature Selection

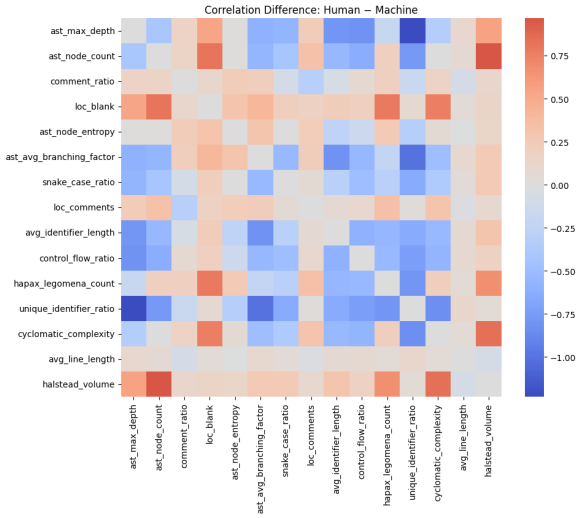

Figure 2: Correlation heatmap of the selected features.

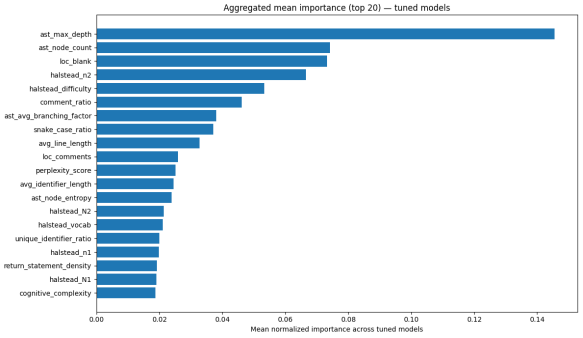

Figure 3: Distribution of aggregated mean feature importances.

Feature	Chi-Square	MI	Correlation	Combined
ast_max_depth	2	2	1	5
unique_identifier_ratio	6	7	4	17
ast_avg_branching_factor	10	5	2	17
control_flow_ratio	3	11	3	17
comment_ratio	1	12	5	18
avg_identifier_length	9	1	9	19
cyclomatic_complexity	7	6	6	19
ast_node_count	11	3	10	24
hapax_legomena_count	8	8	8	24
ast_node_entropy	13	4	16	33
loc_comments	12	13	13	38
syntactic_ngram_frequency	5	26	7	38
maintainability_index	15	17	11	43
snake_case_ratio	4	33	15	52
halstead_volume	19	16	17	52

Table 3: Feature ranking across statistical measures.

B Notable Differences in Features

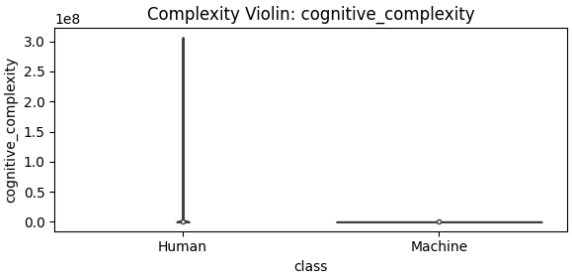

Figure 4: Cognitive complexity variation between human and AI-generated code.

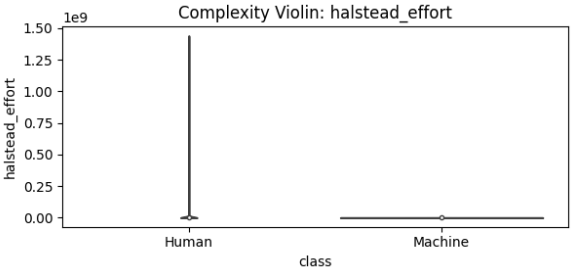

Figure 5: Halstead effort variation.

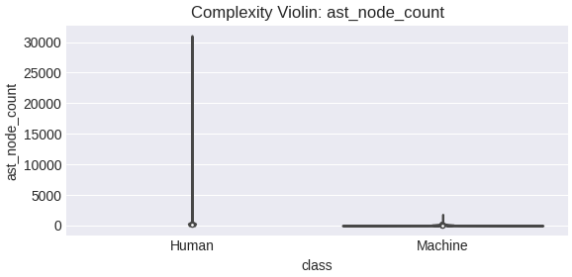

Figure 6: AST node count variation.

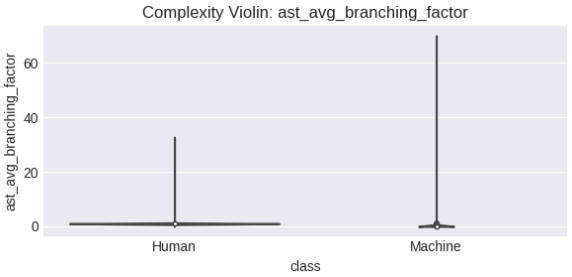

Figure 7: AST branching factor variation.

Feature	Human Mean	Machine Mean	Human Std	Machine Std	Mean Diff	Diff (%)
loc_total	31.872	39.905	84.215	34.870	8.033	25.205
loc_comments	0.453	3.175	4.194	7.544	2.722	600.237
loc_blank	6.955	6.960	45.324	9.735	0.006	0.081
comment_ratio	0.005	0.072	0.029	0.119	0.067	1414.397
avg_line_length	20.452	26.496	98.852	18.762	6.044	29.553
max_line_length	56.320	99.598	1037.364	93.849	43.278	76.842
indentation_std_dev	3.598	3.573	1.592	2.044	-0.025	-0.694
unique_identifier_ratio	0.105	0.049	0.050	0.067	-0.056	-53.090
avg_identifier_length	2.661	1.696	1.290	2.334	-0.965	-36.273
snake_case_ratio	0.153	0.250	0.345	0.425	0.097	63.598
keyword_density	0.079	0.077	0.038	0.047	-0.002	-2.111
hapax_legomena_count	3.791	1.852	3.163	2.794	-1.940	-51.161

Table 4: Statistical comparison of handcrafted features between human-written and machine-generated code.

C Hyperparameters

Model	Hyperparameters
Logistic Regression	`penalty=l1, C=2.63, solver=liblinear, max_iter=2000`
Linear SVC	`penalty=l2, C=1.0, loss=squared_hinge, max_iter=25, dual=False`
Decision Tree	`criterion=gini, max_depth=None, min_samples_split=2, min_samples_leaf=1, ccp_alpha=0.01`
Random Forest	`n_estimators=300, max_depth=30, max_features=sqrt, min_samples_leaf=2`
XGBoost	`n_estimators=200, learning_rate=0.01, max_depth=10, subsample=0.6, eval_metric=logloss, colsample_bytree=0.8`
CatBoost	`iterations=800, learning_rate=0.05, depth=4, l2_leaf_reg=3.0, loss_function=Logloss`

Table 5: Final hyperparameters used for classical models.

D Post Submission Results

Model	Seen Domain and Language	Unseen Domain and Language
Logistic Regression	86.55	43.23
SVM	86.54	44.24
Decision Tree	88.2	**50.74***
Random Forest	90.3	50.46
XGBoost	93.7	49.95
CatBoost	93.6	48.99
ANN	90.21	–