FMI_SU_Yotkova_Kastreva at SemEval-2026 Task 13: Lightweight Detection of LLM-Generated Code via Stylometric Signals

Elitsa Yotkova[1*], **Violeta Kastreva**[1*], **Dimitar Dimitrov**[1], **Ivan Koychev**[1], **Preslav Nakov**[2]
[1]Faculty of Mathematics and Informatics, Sofia University "St. Kliment Ohridski", Bulgaria
[2] Mohamed bin Zayed University of Artificial Intelligence, UAE
eyotkova@g.fmi.uni-sofia.bg, vkastreva@uni-sofia.bg,
{ilijanovd, koychev}@fmi.uni-sofia.bg, preslav.nakov@mbzuai.ac.ae

Abstract

SemEval-2026 Task 13 investigates machine-generated code detection across multiple programming languages and application scenarios, asking participating systems to generalize to unseen languages and domains. This paper describes our participation in Subtask A (binary classification) and explores both pretrained code encoders and lightweight feature-based methods. We design ratio-based features that are less sensitive to snippet length. To support the extraction of descriptiveness-related signals, we use parsing engines and a programming-language classifier. Additionally, we train a separate code-vs-text line classifier to identify raw natural language segments embedded within samples. We combine a shallow decision tree with heuristic rules derived from data analysis to produce the final predictions. Our approach is computationally efficient, requires only CPU resources for training, and achieves near-instant inference time, offering a lightweight alternative to large pretrained models. Our code is available at https://github.com/violeta-kastreva/SemEvalTask13-SubtaskA.

1 Introduction

The widespread adoption of large language models (LLMs) for code generation has reshaped modern software development. Current models can generate syntactically correct and semantically meaningful programs across many programming languages and domains. This has led to increased presence of machine-generated code in cases such as automated unit test generation (Jain et al., 2025), code infilling and completion (Bavarian et al., 2022), and production development workflows (Dunay et al., 2024; Froemmgen et al., 2024). While LLM-based tools offer clear productivity benefits, their ability to author and refine code raises concerns in domains where human authorship is critical.

In academia, detecting code authorship in programming assignments is increasingly important, as reliance on LLMs undermines educational integrity (Sullivan et al., 2023). Similarly, fair technical hiring requires verifying that submitted artifacts are genuinely human-written. These use cases motivate the need for robust detection of machine-generated code.

In an industrial setting, machine-generated code can pose technical risks when deployed without sufficient oversight. Prior work has shown that LLM-generated code may contain insecure logic, hidden backdoors, or injection vulnerabilities, threatening software reliability and data security (Bukhari, 2024; Pearce et al., 2025).

Distinguishing machine-generated code from human-written one is therefore a critical challenge. Unlike natural language, source code is tightly constrained by syntax and semantics, which can obscure stylistic differences between human and machine authors, which are commonly used for machine-generated text detection. In practice, code detection systems must generalize across programming languages, application domains, and generator families, and many existing approaches degrade substantially in such conditions (Orel et al., 2025b).

Recent work has largely focused on pretrained code encoders and large neural models for authorship detection. These models can be effective in in-domain settings, but their performance and reliability can vary under distribution shift, particularly for unseen languages or domains. Moreover, their predictions are often difficult to interpret, making it harder to identify the properties of code that distinguish human-written from machine-generated samples.

This paper presents our approach to SemEval-2026 Task 13 Subtask A (Orel et al., 2026), which targets binary classification of machine-generated versus human-written code under diverse and challenging generalization settings.

[*]Equal contribution.

Proceedings of the The 20th International Workshop on Semantic Evaluation (2026), pages 2179–2186
July 3-4, 2026 ©2026 Association for Computational Linguistics

Instead of relying solely on high-capacity neural representations, we explore lightweight, interpretable signals that capture descriptiveness and stylistic leakage commonly observed in LLM-generated code. Our approach is motivated by the observation that machine-generated code often includes verbose comments, explanatory text, or natural-language artifacts that differ in both frequency and structure from human-written code.

We propose a feature-based detection pipeline centered on ratio-based descriptors that are less sensitive to code length and formatting variability. To ensure robust feature extraction across incomplete or noisy code samples, we employ parsing engines and introduce a dedicated programming language identification classifier. In addition, we train an auxiliary code-versus-text line classifier to detect natural language segments embedded within samples.

Our final system combines shallow machine learning classifiers with heuristic rules, prioritizing robustness and interpretability over model complexity. Through extensive experimentation, we compare pretrained encoder-based representations with feature-driven methods and analyze the strengths and limitations of each approach under cross-language and cross-domain evaluation. On the official leaderboard, our approach is ranked within the top 15% of submissions. Our contributions are as follows:

- We analyze the limitations of pretrained code encoders for code detection under distribution shift.
- We introduce interpretable ratio-based features that capture descriptiveness and stylistic signals in LLM-generated code, and use them to train classical machine learning classifiers.
- We construct a dataset for code-vs-text line classification,[1] [2] enabling line-level discrimination between natural language text and source code.
- We present a lightweight detection pipeline that performs competitively in SemEval-2026 Task 13 while remaining simple and interpretable.

[1] violetakastreva/line-level-code-vs-text-classification-dataset

[2] violetakastreva/line-level-code-vs-text-classification

2 Related Work

Prior work on detecting machine-generated code has emphasized the role of code stylometric features, showing that LLM-generated code differs from human-written code in surface and structural properties such as line statistics, comment usage, and complexity measures (Rahman et al., 2024). Studies on Claude 3-generated code report that features related to the number of lines, blank lines, comment ratio, and cyclomatic complexity are particularly informative, with generated code often exhibiting simpler structure and distinct comment-to-code patterns compared to human-written code (Rahman et al., 2024).

Other approaches move beyond stylometry by leveraging model-internal signals. Some methods compute token-level likelihoods from large language models and frame detection as a classification problem over probability maps using vision architectures (Xu and Sheng, 2025). In addition, pretrained code encoders are commonly used as feature extractors for supervised classification of code authorship, providing strong semantic representations at the cost of reduced interpretability (Rudin, 2019; Nguyen et al., 2024).

3 Dataset

We use the dataset provided for SemEval-2026 Task 13 (Orel et al., 2025a,b) Subtask A, which targets binary classification between machine-generated and human-written code. Each sample is labeled as either fully human-written or fully machine-generated. The training data consists of algorithmic code snippets written in C++, Python, and Java, while evaluation additionally includes unseen programming languages (Go, PHP, C#, C, and JavaScript) and unseen application domains (research and production code).

The dataset contains 500K training samples (238K human-written and 262K machine-generated), of which approximately 457K are written in Python, a 100K validation set, and a 500K test set. The participating systems are evaluated using macro-F1 scores across multiple settings that assess generalization across languages and domains.

4 Methodology

Our system is built around feature-based representations supported by several auxiliary classifiers to enable robust detection of machine-generated code across multiple programming languages and heterogeneous input formats as shown on Figure 1.

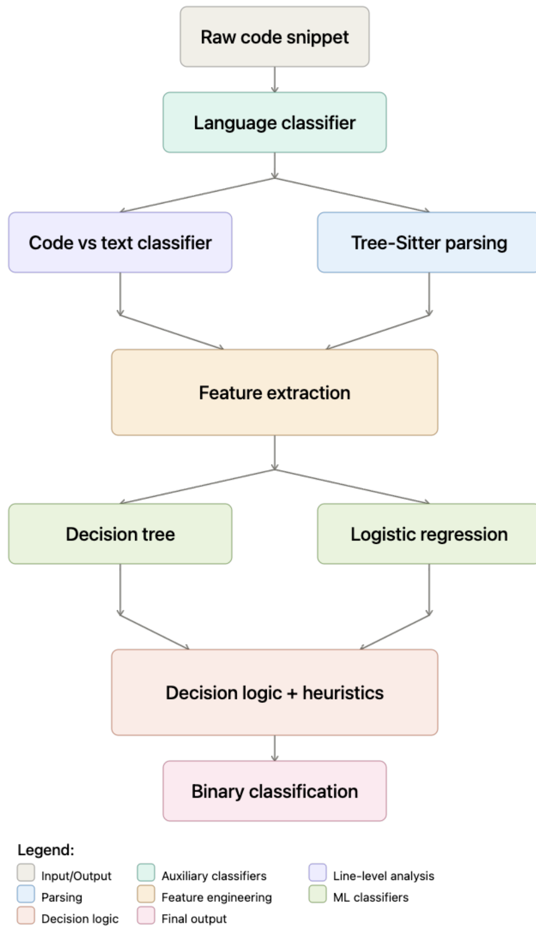

Figure 1: Overview of the proposed machine-generated code detection pipeline.

4.1 Feature Extraction and Engineering

We design ratio-based, interpretable, length-normalized features that capture stylistic descriptiveness and structural cues often present in LLM-generated code. Table 1 lists our stylometric ratio features, while Table 2 summarizes our syntactic and structural features, which we derive from parsing and complexity estimates.

We extract our linguistic features using spaCy[3] for tokenization and verb identification. For reliable comment extraction, we use the incremental parser Tree-Sitter,[4] which enables parsing across all seven languages in the shared task and remains robust to incomplete or syntactically invalid code snippets. Since Tree-Sitter requires the programming language to be known in advance, we introduce a dedicated language identification component, as described in the next subsection.

[3] https://github.com/explosion/spaCy
[4] https://tree-sitter.github.io/tree-sitter/

Feature	Definition
Comment Ratio	comment lines to total lines in the code snippet
Verb Comment Ratio	verb tokens to all word tokens in comment text
Text-like Ratio	lines classified as natural language text to total number of code lines
Identifier Verb Ratio	verb tokens to total number of identifier tokens; the identifiers are decomposed into their constituent word units before token counting
Comment Code-like Ratio	comment lines exhibiting code-like patterns to all comment lines

Table 1: Code stylometric ratio features computed at the snippet level.

Feature	Definition
Completeness Score	an indicator of whether the code snippet forms a syntactically complete unit
Error Nodes Near EOF	concrete syntax tree error nodes occurring near the end of the file to all error nodes
Cyclomatic Complexity (Mean)	average cyclomatic complexity computed across all functions or methods in the snippet
Statements-to-LOC	ratio of the number of statements to the total number of lines of code

Table 2: Snippet-level syntactic and structural features derived from parsing and complexity estimates.

4.2 Programming Language Identification

To support language-specific parsing, we train a programming language classifier using the Rosetta Code dataset (Nanz and Furia, 2014). It is based on character (3,8) n-gram TF.IDF features and a Multinomial Naïve Bayes model with a smoothing hyperparameter of 0.1, enabling fast language prediction across all languages included in the task.

4.3 Code vs. Text Classification

During our analysis of the data, we observed that some samples contain substantial amounts of raw natural language, mixed with or replacing code. To explicitly capture this behavior, we train a binary code-vs-text line-level classifier that operates at the granularity of individual lines. For this task, we construct a custom dataset combining code snippets extracted from Stack Overflow posts (michaelfumery, 2021) and natural language samples from Twitch chat data (mowglii, 2020). Stack Overflow provides diverse, real-world code fragments, while Twitch chat provides informal, conversational text,

allowing the classifier to learn robust distinctions between code-like and text-like content.

We further use a TF.IDF vectorizer with character (3,5)-gram features, combined with a linear classifier, to distinguish code from text. This classifier is computationally lightweight, making it well-suited as an auxiliary component in the pipeline.

4.4 Classification and Decision Logic

We train shallow classifiers, including logistic regression and decision trees. The feature set comprises comment density and the verb-comment ratio within comments, capturing descriptive patterns frequently observed in LLM-generated code.

The final predictions are produced by combining classifier outputs with lightweight heuristics derived from exploratory data analysis. After manual inspection of the training data, we observed a recurring stylistic artifact in the LLM-generated code: the programming language is often emitted as a standalone line preceding the snippet (e.g., `python` or `java`). This pattern appeared in approximately 5k out of the 500k examples. As a sanity check, we first submitted a trivial baseline that predicted only the human-written class (all zeros). We then incorporated a simple heuristic that flags instances with such standalone language markers, yielding an absolute macro-F1 improvement of 0.05 over the baseline. We attribute this to LLMs mimicking Markdown formatting.

Model	Leak Rate (%)
Qwen2.5-Coder-1.5B-Instruct	87.28
Phi-3-medium-4k-instruct	35.78
Phi-3.5-mini-instruct	59.81
Yi-Coder-1.5B-Chat	74.39
Qwen2.5-Coder-7B-Instruct	66.18

Table 3: Leak rate denotes the percentage of all snippets from that model Markdown formatting.

We also design a heuristic based on the text-like content ratio. On the training set, we compute the mean text-like ratio over samples with non-zero values. The mean text-like ratio is 0.07 for human-written labeled examples and 0.17 for LLM-generated code (considering only non-zero text-like ratio values). Based on these statistics, we define an optimistic threshold strictly above both means and set it to 0.3. A Kolmogorov-Smirnov test confirms that the text-like ratio distributions differ significantly between classes ($D = 0.40$, $p < 0.001$). While Youden's J optimization yields a threshold of 0.008, we use 0.3 to prioritize specificity. This threshold achieves 99.9% specificity on human-written code and flags only the top 10.3% of LLM-generated samples with the highest text-like ratios.

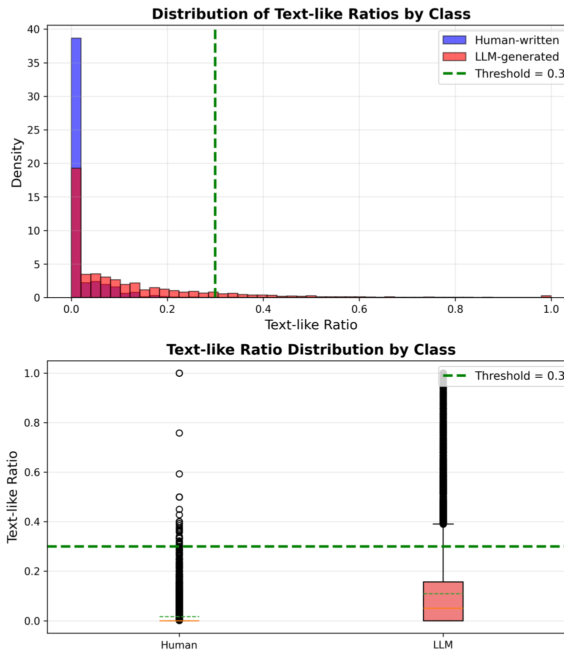

Figure 2: Text-like ratio distributions by class. (a) Histograms and (b) box plots show significant differences (K-S test: D=0.40, p<0.001). Threshold = 0.3 (green line).

During inference, test samples whose text-like ratio exceeds 0.3 are directly classified as LLM-generated. As illustrated in Figure 1, these heuristics are applied alongside classifier predictions to produce the final binary classification.

To account for systematic differences across snippet lengths, we partition the code samples into small, medium, and large buckets in terms of line count and apply separate feature scaling to each bucket. The logistic regression decision threshold is tuned on the validation set to optimize the precision–recall trade-off, whereas the decision tree classifier is constrained to depth 2 and trained using the Gini impurity criterion in order to preserve the interpretability.

5 Experiments

We evaluate the utility of using pretrained code embeddings, handcrafted features, and auxiliary signals for machine-generated code detection on the official SemEval-2026 Task 13 train/validation split and following the official evaluation protocol. The full pipeline, including embedding ex-

traction, runs in approximately 2 hours on a CPU; however, computing the embeddings with large pretrained models can incur substantially higher computational cost, even on a GPU.

5.1 Pretrained Code Embeddings

We evaluated the pretrained code encoders as standalone representations for machine-generated code detection, extracting embeddings from GraphCodeBERT (Guo et al., 2021), CodeT5+ (?), and CodeRankLLM (Suresh et al., 2025) with standard classifiers. Table 4 reports results on the initial 1,000-example development test subset. We also combined encoder embeddings with our feature-based pipeline, but this yielded no improvement. Analysis of Decision Tree feature importances and Logistic Regression coefficients showed that the embedding features contributed negligibly compared with our handcrafted features. Although these pretrained models performed well on the training data, they generalized poorly. We attribute this largely to language shift: 91.4% of the training set consists of Python snippets, whereas the official test set spans seven programming languages. As a result, models may overfit to language-specific cues that transfer weakly across languages. For instance, indentation patterns may be a strong signal for distinguishing LLM-generated from human-written code in Python, where formatting is syntactically meaningful, but such cues are far less informative in languages such as C#, JavaScript, or C++.

Model	Validation	Test
GraphCodeBERT	98.21	25.11
CodeT5+ encoder	97.46	49.39
nomicai/CodeRankLLM	95.27	33.65

Table 4: Pretrained code embeddings: best downstream classifier results (validation vs test macro-F1).

5.2 Feature Selection

We select the final feature subset based on model-based interpretability, relying on SHAP feature importance for tree-based classifiers and coefficient inspection for logistic regression. The initial feature pool comprised approximately 30 stylometric and syntactic features, including measures such as nesting-mean, return-statement-ratio, identifier-entropy, and related statistics. We also examined pairwise feature correlations to identify redundant signals and reduce multicollinearity. Although this broader set captured diverse properties of the source code, only a small subset was consistently informative for the final classification task. Based on this analysis, we retain the two best-performing features for the final system: the comment-ratio and the verb-comment-ratio. We additionally consider several auxiliary features that provided complementary signals in intermediate models, which we describe in detail below.

Errors near EOF ratio We introduce a syntax-based feature that captures the distribution of parsing errors near the end of a code snippet. We collect all nodes labeled as parsing errors and compute the proportion of those whose spans overlap with the final 20% of the file. This ratio is designed to reflect incomplete or abruptly terminated code, a pattern frequently observed in LLM-generated snippets. Incorporating this feature yields stable performance across splits, with results nearly identical to our best-performing configuration.

Completeness score To capture incomplete or abruptly terminated code without relying on Tree-Sitter parsers, we prompt Qwen2.5-7B-Instruct-1M (Yang et al., 2025) to assess the completeness of the final line in a code snippet. The model assigns a score of 0 to complete lines, 0.5 to uncertain cases, and 1 to incomplete lines. While this approach successfully captures some forms of truncated code, it is computationally expensive and does not really yield improvements on the test set.

Computing cyclomatic complexity We compute the cyclomatic complexity by heuristically extracting function or method bodies from signatures and block structure using language-agnostic patterns. For each function, the complexity is defined as one plus the number of decision points (e.g., conditionals, loops, logical operators, and exception handling), computed only over lines that are classified as code, with comments and string literals removed. We aggregate the function-level values within each snippet and use the means as global features. These statistics capture structural differences but provide only a moderate signal, particularly for production code where complexity varies widely across styles and domains.

Summary We observe similar behavior for other structural features, including Identifier Verb Ratio and Statements-to-LOC. Although they capture certain stylistic and syntactic properties, their contribution to the final classification decision is limited. In practice, they either provide only a weak signal or overfit to superficial patterns in the training data, which reduces their robustness across domains.

Feature	Validation	Test
Error near EOF ratio	64.75	63.90
Completeness score	64.14	63.26
Cyclomatic complexity mean	53.59	63.16

Table 5: Performance of additional features combined with the main features (Comment Ratio and Verb Comment Ratio) using Logistic Regression.

5.3 Logistic Regression Threshold Tuning

As described earlier, we partition the code samples into three buckets by the number of code lines and apply separate feature scaling to each group. For each bucket, we optimize a decision threshold for a logistic regression classifier on the validation set and assign labels at inference time using the bucket-specific threshold. This strategy proves highly effective in improving validation performance. However, the optimal thresholds learned for individual buckets on the validation set do not generalize consistently to the test set, indicating sensitivity to distributional differences across code-length regimes and limiting generalization.

6 Results

Detailed per-model results are provided in Appendix A (Table 7); below, we summarize the main findings and discuss the key performance trends.

6.1 Code-vs-Text Classifier

The code-vs–text line classifier achieves strong performance on the validation set, with a macro-averaged F1 score of 96.07 and an accuracy of 97.11. On a manually annotated subset of 30 examples drawn from the test set, performance decreases to an accuracy of 89.47 and a macro-averaged F1 score of 86.43.This reduction is largely due to ambiguous cases in the competition's test data, where certain lines contain repetitive or non-linguistic character sequences (e.g., "*i... i... i...* ", "*it's 4am i've been there ? what do i have ?*") originating from code snippets. Although such lines are labeled as text in the dataset, classifying them as code is reasonable and expected. Due to its speed and strong performance, we favor this approach over embedding-based alternatives, which would introduce additional computational overhead without clear benefits for this subtask.

6.2 Language Identification Classifier

The programming language identification classifier achieves an accuracy of 95.10 on a test set of 1k examples derived from the public test, with labels obtained by prompting the OpenAI ChatGPT-5.2 API. This auxiliary annotation was introduced solely for testing purposes and as a sanity check; model validation was conducted on the Rosetta Code dataset. While most predictions are correct, some misclassifications are critical in principle, as different programming languages may induce distinct error nodes or comment signatures in the Tree-Sitter grammars used for parsing. The most frequent confusion occurs between C and C++, and between C# and Java. However, these errors have a limited impact on downstream performance, as the affected languages share highly similar grammatical structures in Tree-Sitter, yielding comparable parse trees for feature extraction.

6.3 Code Classification

For the binary classification of machine-generated versus human-written code, the decision tree classifier achieves a macro-averaged F1 score of 65.62. In contrast, logistic regression with adjusted decision thresholds attains a macro-averaged F1 score of 63.16. Incorporating the heuristic-based decision function yields an absolute improvement of approximately 2% in macro-averaged F1. Overall, our system achieves a macro-F1 of 67.35, demonstrating benefits of combining lightweight classifiers with heuristics.

Classifier	Macro-F1
Logistic Regression, default treshold	63.16
Logistic Regression, threshold = 0.65	64.59
Decision Tree, depth = 2	65.62
Decision Tree, depth = 2 + heuristic	**67.35**

Table 6: Classifier overview with Comment Ratio and Verb Comment Ratio as features.

7 Conclusion and Future Work

We presented a lightweight, interpretable system for SemEval-2026 Task 13 Subtask A based on ratio-driven stylometric features, supported by language identification and line-level code–text classification. A shallow classifier with minimal heuristics achieves a macro-F1 of 67.35 on the official test set with CPU-only, low-latency inference.

In future work, we plan to improve multi-core CPU utilization by parallelizing feature extraction and reducing Python overhead. We further plan to combine these features with LLM-based signals (e.g., selective fallback or distillation) to improve robustness under distribution shift.

Acknowledgements

This research is partially funded by the EU NextGenerationEU, through the National Recovery and Resilience Plan of the Republic of Bulgaria, project SUMMIT, No BG-RRP-2.004-0008.

References

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle.

Sufiyan Ahmed Bukhari. 2024. Issues in detection of AI-generated source code. Master's thesis, University of Calgary, Calgary, Alberta, Canada.

Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C. Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Maddila, Vijayaraghavan Murali, Ali Tayyebi, and Nachiappan Nagappan. 2024. Multiline AI-assisted code authoring. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 150–160, Porto de Galinhas, Brazil. ACM.

Alexander Froemmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, Maxim Tabachnyk, Daniel Tarlow, Kevin Villela, Daniel Zheng, Satish Chandra, and Petros Maniatis. 2024. Resolving code review comments with machine learning. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '2024, pages 204–215, Lisbon, Portugal. ACM.

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 B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. 2025. Testgeneval: A real world unit test generation and test completion benchmark. In *International Conference on Learning Representations*.

michaelfumery. 2021. Stackoverflow questions filtered (2011–2021). Kaggle dataset.

mowglii. 2020. Twitch chat test data. Kaggle dataset.

Sebastian Nanz and Carlo A. Furia. 2014. A comparative study of programming languages in rosetta code. *Preprint*, arXiv:1409.0252.

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*, page 112059.

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. 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 31263–31289, Suzhou, China. Association for Computational Linguistics.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *Communications of the ACM*, 68(2):96–105.

Musfiqur Rahman, SayedHassan Khatoonabadi, Ahmad Abdellatif, and Emad Shihab. 2024. Automatic detection of LLM-generated code: A comparative case study of contemporary models across function and class granularities.

Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1:206–215.

Miriam Sullivan, Andrew Kelly, and Paul McLaughlan. 2023. ChatGPT in higher education: Considerations for academic integrity and student learning. *Journal of Applied Learning & Teaching*, 6(1):31–40.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2025. CoRNStack: High-quality contrastive data for better code retrieval and reranking. In *International Conference on Learning Representations*. Poster.

Zhenyu Xu and Victor S. Sheng. 2025. CodeVision: Detecting LLM-generated code using 2D token probability maps and vision models.

An Yang, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoyan Huang, Jiandong Jiang, Jianhong Tu, Jianwei Zhang, Jingren Zhou, Junyang Lin, Kai Dang, Kexin Yang, Le Yu, Mei Li, Minmin Sun, Qin Zhu, Rui Men, Tao He, and 9 others. 2025. Qwen2.5-1m technical report.

A Main-Task Model Results

Table 7 reports detailed main-task results on the official test set, comparing macro-F1 across classifiers and training setups. Notably, the strongest performance comes from linear models (and a shallow decision tree), while more flexible nonlinear baselines (random forests, MLP) perform worse—suggesting the stylometric ratio features induce a linear decision boundary and that additional nonlinearity tends to reduce robustness under distribution shift.

Model	**Setup**	**Macro-F1**
Logistic Regression	solver=liblinear, bucket scaling + tuned decision threshold 0.65	64.59
Decision Tree	depth=2	**65.62**
Linear SVC	default parameters	63.27
Random Forest	11 trees, depth=2	62.25
Logistic Regression	solver=liblinear	63.16
MLP	(10,5), tanh, adam, alpha=0.0001	60.25

Table 7: Results for the main task. For logistic regression, *tuned decision threshold* refers to selecting the probability cutoff on the validation set to maximize macro-F1 (rather than using the default value of 0.5).