HumanVsGenAI: AI Code Detection Performance in Educational Programming Tasks using DetectCodeGPT

Aagat Pokhrel, Chandra Raskoti, Chong Shen, and Iftekharul Islam

Abstract— **Motivation: The widespread adoption of AI coding tools in educational settings has created challenges for assessing student work and maintaining academic integrity.**

Objectives: This study aims to evaluate the effectiveness of DetectCodeGPT in distinguishing between AI-generated and human-written code across different programmer experience levels.

Methods: We evaluated DetectCodeGPT using two datasets: solutions from experienced GitHub contributors (Dataset A) and CS1/CS2-level student submissions from educational assignments (Dataset B). For each problem, we collected human-written solutions and generated corresponding AI solutions using state-of-the-art language models. Detection performance was measured using AUROC scores.

Results: DetectCodeGPT achieved an AUROC of 0.72 on GitHub code but only 0.47 on CS1/CS2-level student code, suggesting limited effectiveness in educational settings. Performance improved with additional perturbations up to 50, beyond which no significant improvements were observed.

Conclusion: While promising for code from more experienced programmers, current whitespace-based detection methods struggle with novice student submissions. This highlights the need for more sophisticated semantic analysis in educational AI code detection and raises questions about potential AI contamination in student code datasets.

I. Introduction

Instructors today face a growing challenge: AI coding tools are rapidly changing how students approach programming assignments, making it difficult to distinguish original work from AI-assisted solutions. As these tools become increasingly capable of generating syntactically correct and functionally appropriate code, educators struggle to fairly evaluate student understanding and skills.

While various AI detection approaches have been developed [1]–[4], their effectiveness within educational programming contexts remains largely unexplored. This gap is significant as programming courses require reliable authenticity assessment methods in an environment where AI assistance is becoming widely accessible.

Our study addresses this gap by evaluating DetectCodeGPT, a perturbation-based approach to AI code detection, across different programmer experience levels. We investigate two primary research questions:

- **RQ1:** How effective is DetectCodeGPT in detecting AI-generated solutions across educational programming tasks?
- **RQ2:** How does programmer experience influence detection accuracy?

By systematically comparing detection performance between open-source GitHub solutions and CS1/CS2-level student submissions, we provide insights into the practical applicability of current detection approaches in educational settings and identify key limitations that must be addressed in future work.

Aagat Pokhrel, Chandra Raskoti, Chong Shen, and Iftekharul Islam are with Min H. Kao Department of Electrical Engineering and Computer Science at University of Tennessee, Knoxville, TN, USA apokhre2@vols.utk.edu, craskoti@vols.utk.edu, cshen8@vols.utk.edu, mislam73@vols.utk.edu

II. Related Work

A. Detecting AI-Generated Content

Early AI detection methods focused on natural language text, using statistical features to identify machine-generated content. Tools like GPTZero [1] and OpenAI's text classifier [5] relied on statistical predictability, while GLTR [6] visualized token probabilities. DetectGPT [7] introduced a perturbation-based approach, examining how sentence likelihood changes after token replacements. However, these text-focused methods poorly transfer to programming contexts, where human writing exhibits less stylistic variation.

B. Detection Approaches for Code

Recent research has introduced detection strategies tailored specifically for source code. Supervised learning methods like GPTSniffer [8] fine-tune models such as CodeBERT [9] to classify code authorship. While effective within specific domains, these approaches often struggle with generalization across different languages and with model outputs [3]. Unsupervised techniques from Jegourel et al. [2] employ clustering based on structural distance metrics, revealing that novice and AI code submissions often form distinct stylistic clusters.

Another category focuses on detection based on likelihood or entropy, using pretrained code models to assign probability scores to sequences. While LLM-generated code often appears more "natural" under such models, this distinction weakens when comparing against human-written code [7]. Xu and Sheng introduce AIGCode [3] which computes perplexity and burstiness on a line-by-line basis to detect stylistic regularity. Similarly, Yang el al. [10] proposes surrogate modeling techniques for code detection on black-box LLMs, estimating rightmost token probabilities using white-box proxies like PolyCoder [11].

Among perturbation-based approaches, DetectCodeGPT [4] shows promise. It extends DetectGPT [7] to code by perturbing whitespace instead of semantic tokens, avoiding syntax violations while preserving meaning. By measuring the Normalized perturbed Log-rank (NPR) score,

it captures the rigid formatting of LLM-generated coda and does not require training on datasets. This zero-shot capability makes it potentially suitable for educational contexts, where training data may be limited or unavailable.

C. Educational Context Challenges

Education presents unique challenges for AI code detection. Students use AI tools in varied ways—from complete reliance to hybrid approaches that blend AI assistance with manual coding [12]. Detection systems perform inconsistently in educational settings: Pan et al. [13] found all tested detectors produced concerning rates of both false positives and false negatives, while Dunder et al. [14] demonstrated that AI tools succeed on simple assignments but often fail on complex problems.

Despite growing research on AI's role in programming education, systematic evaluation of detection tools across varying programmer expertise levels remains limited. Our work addresses this gap by evaluating DetectCodeGPT using datasets that span from novice student submissions to experienced programmer solutions.

III. Methodology

A. Dataset Collection and Preparation

For a comprehensive evaluation of DetectCodeGPT, we curated two distinct datasets representing different programmer experience levels and coding contexts.

1) Dataset A: Dataset A consists of human-written Python solutions collected from GitHub repositories with MIT licenses, ensuring ethical usage for research purposes. We selected 35 LeetCode problems covering different difficulty levels (easy, medium, and hard) and algorithmic concepts including dynamic programming, graph algorithms, and data structures.

When selecting problems, we focused on those requiring specific algorithm implementations that reflect typical educational assignments, with unambiguous specifications and varied complexity levels. For each problem, we collected three human-written solutions from GitHub repositories, representing code from intermediate to experienced programmers who actively contribute to open-source platforms. These solutions typically demonstrate advanced programming techniques, optimization strategies, and diverse coding styles gained through experience.

2) Dataset B: Dataset B leverages the publicly available "Dataset of Student Solutions to Algorithm and Data Structure Programming Assignments" from the University of Hamburg [15]. From this collection, we selected 21 algorithmic programming problems focusing on fundamental data structures and algorithms that align with typical curriculum topics. Similar to Dataset A, we included three student-written Python solutions for each problem. This dataset contains code from undergraduate computer science students, likely at beginner to intermediate skill levels (CS1/CS2-level), reflecting the coding patterns of learners in structured educational environments.

The dual-dataset approach offers two-fold methodological advantages:

- Tests detection robustness across different programmer experience levels
- Enhances real-world applicability by covering both competitive programming scenarios and educational assignments

B. AI Solution Generation

For both datasets, we generated three AI solutions per problem using state-of-the-art large language models: ChatGPT-4o [16], Claude 3.7 Sonnet [17], and DeepSeek-v3 [18]. Each model received identical input consisting of the complete problem description, input/output examples, and any constraints specified in the original problem, along with a request to solve the problem in Python. We used default temperature settings to ensure the solutions represented the models' typical code generation capabilities without additional instructions or constraints.

C. Dataset Analysis

To develop effective detection strategies, we analyzed our datasets for distinguishing characteristics between human and machine-generated code through two key approaches: lexical diversity and naturalness.

1) Lexical Diversity: Lexical diversity in programming refers to the variety in variable names, functions, and other code elements [4]. We examined this through two established metrics: Zipf's law [19] and Heaps' law [20].

a) Zipf's Law: Zipf's Law describes the inverse relationship between a token's frequency and its rank. In code, common tokens (like keywords) should appear much more frequently than rare ones, following a power-law distribution. For human-written code, this reflects habitual use of common constructs, while in AI-generated code, distributions may appear sharper due to training biases.

As shown in Figure 1, in both of our datasets, the distinction is not that different between human and machine code. However, we do see slight differences in that machine code has a sharper decline, which may suggest that the code might be uniform. Still this is not a good result to use for our algorithm, as there is only a slight unnoticeable difference.

b) Heaps' Law: Heaps' Law characterizes how vocabulary size grows with corpus size. In human code, the vocabulary grows more gradually, as developers tend to reuse variables and structures. In contrast, AI-generated code may show faster vocabulary growth (higher Heaps' exponent), especially if it creates diverse or inconsistent identifiers, suggesting less abstraction or modular reuse.

Figure 2 demonstrates that the Heaps' Law practically did not have any differences graphically. This means for our dataset, the vocabulary size of human and AI code increased similarly with the size of the corpus.

2) Naturalness: Code naturalness refers to how predictable code patterns are to language models. This concept leverages the observation that both human and AI code follow predictable patterns, but potentially in different ways [4].

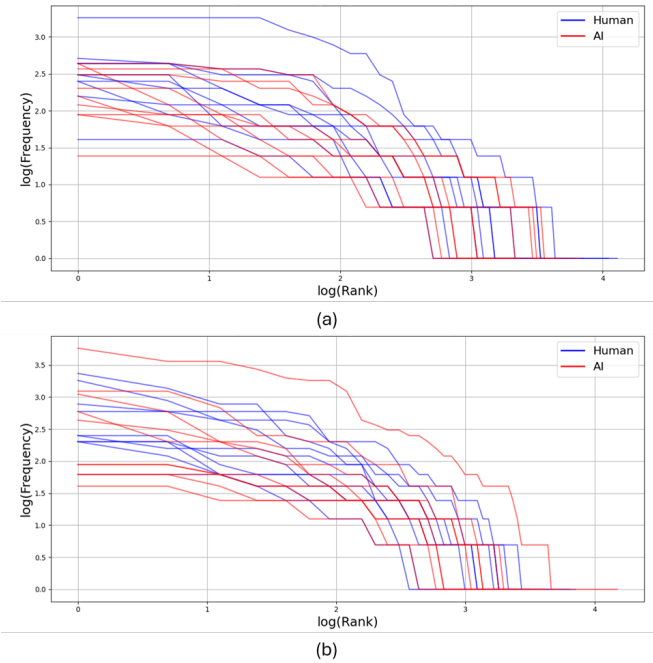

Fig. 1: Zipf's Law log-log plots comparing frequency distributions between human-written (blue) and AI-generated (red) code. (a) Dataset A and (b) Dataset B. While both datasets show similar token frequency distributions, AI-generated code exhibits a slightly sharper decline in some samples, potentially indicating more uniform vocabulary usage. However, the distinction is subtle and not pronounced enough to serve as a reliable differentiator for AI detection purposes.

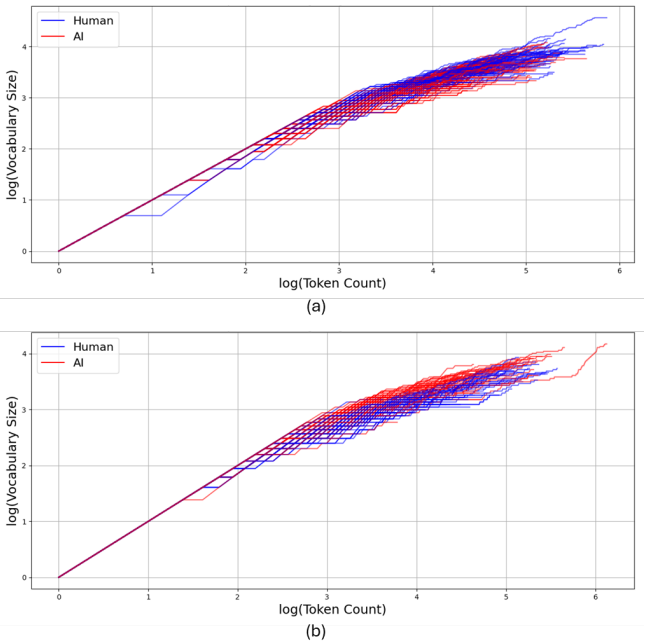

Fig. 2: Heaps' Law log-log plots showing vocabulary growth as corpus size increases for human-written (blue) and AI-generated (red) code. (a) Dataset A and (b) Dataset B. Both datasets demonstrate virtually identical vocabulary growth patterns between human and AI code, with closely overlapping trajectories indicating similar rates of introducing new tokens as code length increases. This lack of differentiation suggests that vocabulary growth patterns are not a reliable indicator for distinguishing between human and AI-authored code.

To quantify this, researchers use language models that assign probabilities to tokens in code, measuring how "natural" or "expected" each token is in its context. Two key metrics for evaluating naturalness are:

a) Log Likelihood [4]: This represents a model's confidence in predicting a token—higher values indicate more "natural" or expected tokens.

b) Log Rank [4]: This measures a token's position in a model's ranked predictions—lower ranks indicate better anticipation by the model. We analyze log rank differences between human and AI code in Table I.

As Table I shows, the log rank analysis for naturalness shows that Comments and Whitespace differ the most and AI comments have significantly higher log ranks compared to human ones, especially in Dataset B. This suggests that AI-generated comments may be less natural or consistent, possibly more generic or less context-aware. Keywords and Operators have very similar log ranks between AI and humans, suggesting both use standard programming constructs in predictable ways. Identifiers and Literals show small but notable differences. Human-written code tends to have slightly more predictable identifiers, likely due to more consistent naming conventions and reuse.

D. DetectCodeGPT Implementation [4]

Based on our dataset analysis showing whitespace and comments as the most distinctive elements, we implemented the DetectCodeGPT approach focusing on whitespace perturbation as in the original work. We implement the same perturbation strategy suggested from the white paper. The measure of impact would rely on tracking changes in naturalness scores.

1) Perturbation: DetectCodeGPT formulates AI code detection as a classification task by applying perturbations to code snippets and measuring how these affect naturalness metrics. As a strategy, we introduce a perturbation process $q(\cdot|x)$, which transforms the original snippet x into a semantically equivalent but syntactically altered version $\tilde{x}$.

Our perturbation approach operates on two levels:

a) Space Insertion: We randomly insert spaces at a subset of potential positions in the code, with the number of spaces at each position following a Poisson distribution:

$$n_{spaces}(c) \sim P(\lambda_{spaces}) \tag{1}$$

b) Newline Insertion: Similarly, we insert newlines with probabilities following a Poisson distribution:

$$n_{newlines}(l) \sim P(\lambda_{newlines}) \tag{2}$$

For each code sample, we generate multiple perturbed variants ($k = 10$ by default) that maintain semantic equivalence while altering formatting.

TABLE I: Log-Rank comparison between AI and human code by token category for Dataset A and Dataset B. Notable differences appear in comments and whitespace formatting, where AI code shows higher log-ranks in both datasets. This suggests these syntactic elements are potentially valuable signals for detection algorithms.

Category	Log-rank (AI)		Log-rank (Human)	
	Mean	Std	Mean	Std
comment	8.29	0.36	8.15	0.80
identifier	6.11	0.75	5.81	0.66
keyword	9.35	0.62	9.27	0.63
literal	4.84	0.56	4.69	0.87
operator	6.19	0.32	6.31	0.31
whitespace	5.43	0.27	5.18	0.35

(a) Dataset A

Category	Log-rank (AI)		Log-rank (Human)	
	Mean	Std	Mean	Std
Comment	8.35	0.38	7.48	0.36
Identifier	5.96	0.65	6.06	0.74
Keyword	9.39	0.61	9.43	0.46
Literal	5.26	0.85	4.83	0.39
Operator	6.26	0.27	6.10	0.31
Whitespace	5.50	0.30	5.08	0.38

(b) Dataset B

2) Normalized Perturbed Log-Rank Measure: For detection, we use the Normalized Perturbed Log-Rank (NPR) score, which has been shown to outperform likelihood-based measures for code detection. The NPR score measures how a code snippet's "naturalness" changes when its whitespace is altered—AI-generated code typically shows higher sensitivity to these changes than human code. The NPR score is calculated as:

$$\text{NPR}(x, p_\theta, q) \triangleq \frac{\mathbb{E}_{\tilde{x}\sim q(\cdot|x)} \log r_\theta(\tilde{x})}{\log r_\theta(x)}, \tag{3}$$

E. Evaluation Metric

To evaluate detection performance, we use the Area Under the Receiver Operating Characteristic curve (AUROC), which measures a classifier's ability to distinguish between classes (human vs. AI code). AUROC values range from 0.5 (random guessing) to 1.0 (perfect classification), with higher values indicating better detection capability. Using NPR scores as input, AUROC calculation is defined as:

$$\text{AUROC} = \int_0^1 \text{TPR}(t)dt, \tag{4}$$

where $\text{TPR}(t)$ is the true positive rate at threshold t. AUROC is particularly appropriate for our task as it captures the subtle differences between human and machine code without requiring a fixed decision boundary.

IV. Results

We evaluated DetectCodeGPT on two distinct datasets representing different programming contexts and examined how implementation parameters affect performance.

A. Experimental Setup

Following the original DetectCodeGPT methodology, we configured our experimental parameters as follows: The number of perturbations was limited to 10 permutations applied to each code sample. For whitespace manipulations, we set the probability parameters α and β to 0.5, determining the proportion of positions selected for perturbation. The Poisson distribution parameters $\lambda_{newline}$ and λ_{space} were both set to 2, controlling the number of spaces or newlines inserted at each position.

For model evaluation, we employed multiple language models: CodeGen-350M, CodeGen-2.3B, and Llama-1.3B.

B. Detection Performance

Figure 3 shows that on Dataset A, the model demonstrated effective performance in distinguishing between human-written and machine-generated code. An Area Under the Receiver Operating Characteristic (AUROC) score of 0.72 was achieved. Among the various models experimented with, the CodeGen 3.7B parameter model achieved the highest performance. A distinct distribution of naturalness scores was observed between human and machine-generated code samples. Machine-generated code exhibited higher naturalness scores compared to human-written code as expected.

In contrast, as depicted in Figure 4, the results obtained on Dataset B were not as good as achieved in Dataset A. The distribution of naturalness scores for human-written and machine-generated code was found to be similar. Thus, the model was less effective in detecting a clear distinction between the two code types. An AUROC score of only 0.47 was achieved, which is below the performance of a random prediction model.

A potential hypothesis for this outcome is related to the nature of Dataset B. Since the dataset is taken from student-submitted solutions, one speculation is that a portion of these student-submitted codes may have been initially generated by AI tools. If this was the case, the model would be tasked with distinguishing between AI-generated code and student-submitted code that was also AI-generated, leading to similar naturalness scores and a reduced ability to differentiate, even with the application of perturbations.

C. Effect of Number of Perturbations

We also investigated how varying the number of perturbations affects detection performance, as shown in Figure 5.

Increasing perturbations from 10 to 20 improved AUROC by approximately 6% and further increase from 20 to 50 perturbations improved an additional 1.4%. No significant improvements were achieved beyond 50 perturbations. The method achieved an optimal level of performance even with a relatively small number of perturbations. This indicates that the method is efficient with respect to the number of perturbations and thus with computational cost.

V. Limitations and Future Work

While DetectCodeGPT demonstrates strong sensitivity to whitespace perturbations and effectively exposes rigid for-

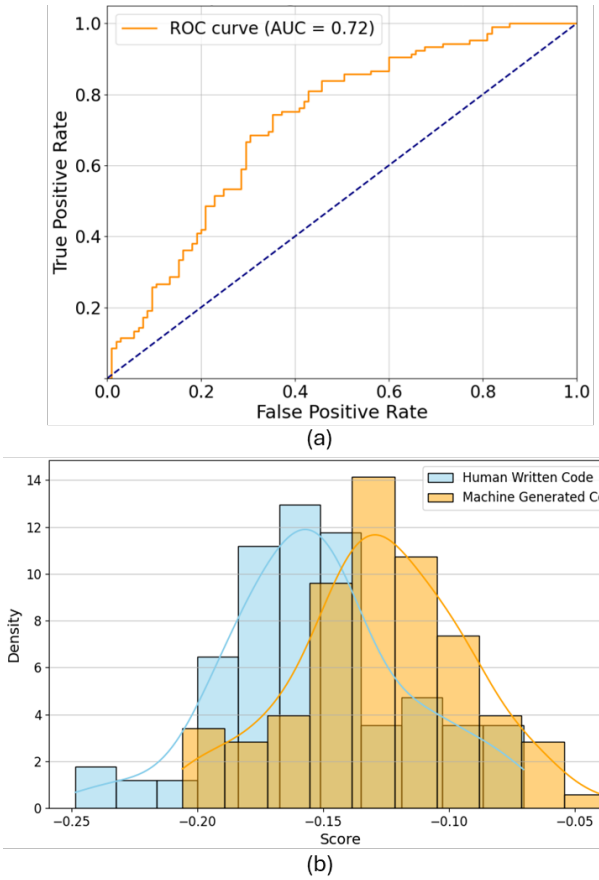

Fig. 3: Dataset A: (a) AUROC performance curve and (b) Naturalness score distribution for human vs. machine-generated code. The AUROC of 0.72 indicates good classification performance, with clear separation between the naturalness score distributions, reflecting the model's ability to distinguish between human and AI-generated code in professional programming contexts.

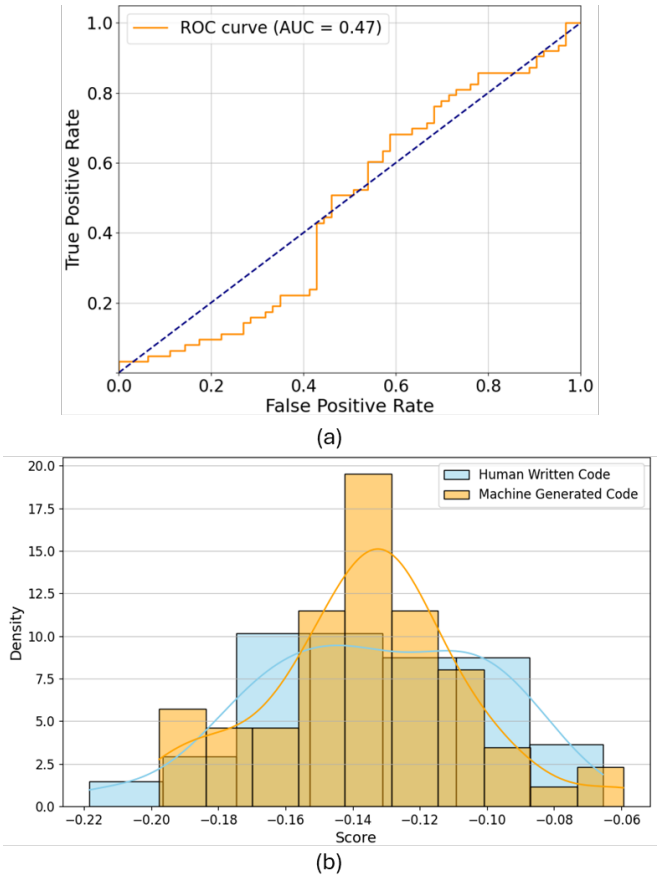

Fig. 4: Dataset B: (a) AUROC performance curve and (b) Naturalness score distribution for human vs. machine-generated code. The AUROC of 0.47, slightly below random-chance performance, indicates that the model struggles to differentiate between student-written and AI-generated code. The overlapping distributions suggest either highly similar code characteristics or potential AI assistance in the supposedly human-written samples.

matting conventions of machine-generated code, our evaluation reveals important limitations and validity concerns that warrant consideration.

A. Internal Validity

Internal validity concerns the credibility of the causal relationships we identify. The primary threat is potential contamination of our "human" dataset, particularly in Dataset B. Student submissions may already incorporate AI assistance, blurring the distinction between human and AI-authored code. This could explain the poor performance (AUROC 0.47) on student code.

DetectCodeGPT's performance disparity between Dataset A and B cannot be conclusively attributed solely to the detection algorithm without considering possible dataset contamination. Previous research has shown that students increasingly incorporate AI tools in their programming assignments, even when not explicitly permitted.

Future work: We will develop heuristic filtering and manual vetting procedures to verify authorship before including samples in our benchmarks.

B. External Validity

External validity addresses the generalizability of our findings. Our experiments were limited to LLMs up to 7B parameters (e.g., CodeLlama-7B). Preliminary tests on larger models (13B, 30B, 70B+) indicate a notable drop in AUROC, as these models adopt more refined whitespace conventions and richer token distributions. Additionally, we focused exclusively on Python code, potentially limiting applicability to other programming languages with different formatting conventions and tools.

Future work: We plan to extend our evaluation to next-generation LLMs and additional programming languages to ensure robustness against evolving formatting patterns.

C. Construct Validity

Construct validity addresses whether our measurements capture the intended concepts. DetectCodeGPT's decision rule relies almost exclusively on stylistic tokens (spaces, newlines, indentation), ignoring semantic or structural code properties such as control flow or API usage. This raises concerns that our detection approach may not fully capture

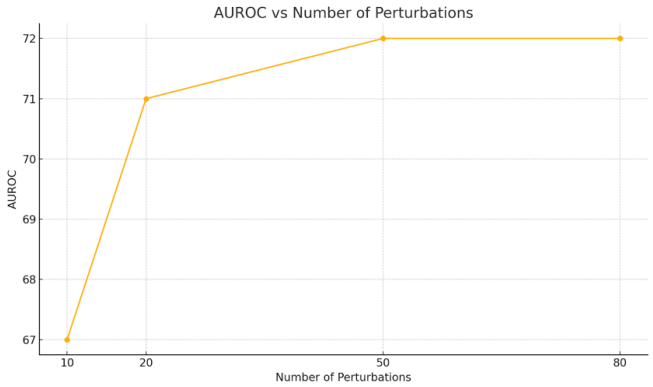

Fig. 5: Effect of different levels of perturbations on AUROC performance. Significant improvement is observed when increasing from 10 to 20 perturbations (6% gain), with smaller improvement from 20 to 50 perturbations (1.4% gain), and no meaningful improvement after 50 perturbations. This demonstrates that the method reaches optimal performance with a moderate number of perturbations, offering computational efficiency for practical applications.

the distinction between human and AI code generation processes.

Consequently, well-formatted human code adhering to industry style guides (e.g., PEP 8, Google Java Style) can exhibit the same perturbation-sensitivity signal as LLM outputs, leading to elevated false-positive rates.

Future work: We will augment the detection score with language-aware features—such as AST structure, API-call patterns, or data-flow graphs—to capture deeper, format-invariant differences between human and machine code.

VI. Conclusion

This study evaluates DetectCodeGPT across different programmer experience levels reveals significant disparities in detection performance. For RQ1, we found the detector achieves reasonable effectiveness on professional code (AUROC 0.72) but performs no better than random chance on student code (AUROC 0.47). For RQ2, results indicate that programmer experience strongly influences detection accuracy, with GitHub code from more experienced programmers being more distinguishable from AI-generated code than CS1/CS2-level student submissions.

Our findings suggest current whitespace-based detection approaches may be inadequate for educational settings. Future work should focus on developing more sophisticated detection methods incorporating semantic analysis, expanding evaluation to additional programming languages, and investigating potential AI contamination in student code datasets. As AI tools become increasingly integrated into programming education, detection systems must adapt to both evolving AI capabilities and diverse student programming styles.

References

[1] E. Tian and A. Cui, "Gptzero," 2023. [Online]. Available: https://gptzero.me

[2] C. Jegourel, J. Y. Ong, O. Kurniawan, L. Meng Shin, and K. Chitluru, "Sieving coding assignments over submissions generated by ai and novice programmers," in *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, 2024, pp. 1–11.

[3] Z. Xu and V. S. Sheng, "Detecting ai-generated code assignments using perplexity of large language models," in *Proceedings of the aaai conference on artificial intelligence*, vol. 38, no. 21, 2024, pp. 23 155–23 162.

[4] Y. Shi, H. Zhang, C. Wan, and X. Gu, "Between lines of code: Unraveling the distinct patterns of machine and human programmers," *arXiv preprint arXiv:2401.06461*, 2024.

[5] OpenAI, "New ai classifier for indicating ai-written text," 2023. [Online]. Available: https://openai.com/index/new-ai-classifier-for-indicating-ai-written-text/

[6] S. Gehrmann, H. Strobelt, and A. M. Rush, "Gltr: Statistical detection and visualization of generated text," *arXiv preprint arXiv:1906.04043*, 2019.

[7] E. Mitchell, Y. Lee, A. Khazatsky, C. D. Manning, and C. Finn, "Detectgpt: Zero-shot machine-generated text detection using probability curvature," in *International Conference on Machine Learning*. PMLR, 2023, pp. 24 950–24 962.

[8] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, "Gptsniffer: A codebert-based classifier to detect source code written by chatgpt," *Journal of Systems and Software*, vol. 214, p. 112059, 2024.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[10] X. Yang, K. Zhang, H. Chen, L. Petzold, W. Y. Wang, and W. Cheng, "Zero-shot detection of machine-generated codes," *arXiv preprint arXiv:2310.05103*, 2023.

[11] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, 2022, pp. 1–10.

[12] M. Kazemitabaar, X. Hou, A. Henley, B. J. Ericson, D. Weintrop, and T. Grossman, "How novices use llm-based code generators to solve cs1 coding tasks in a self-paced learning environment," in *Proceedings of the 23rd Koli calling international conference on computing education research*, 2023, pp. 1–12.

[13] W. H. Pan, M. J. Chok, J. L. S. Wong, Y. X. Shin, Y. S. Poon, Z. Yang, C. Y. Chong, D. Lo, and M. K. Lim, "Assessing ai detectors in identifying ai-generated code: Implications for education," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, 2024, pp. 1–11.

[14] N. Dunder, S. Lundborg, J. Wong, and O. Viberg, "Kattis vs chatgpt: Assessment and evaluation of programming tasks in the age of artificial intelligence," in *Proceedings of the 14th Learning Analytics and Knowledge Conference*, 2024, pp. 821–827.

[15] F. Petersen-Frey, M. Soll, L. Kobras, M. Johannsen, P. Kling, and C. Biemann, "Dataset of student solutions to algorithm and data structure programming assignments," in *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, 2022, pp. 956–962.

[16] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford, *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[17] Anthropic, "Claude 3 family of ai assistants," https://www.anthropic.com/claude, 2024.

[18] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[19] G. K. Zipf, *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio books, 2016.

[20] H. S. Heaps, *Information retrieval: Computational and theoretical aspects*. Academic Press, Inc., 1978.