Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models

Zhenyu Xu, Victor S. Sheng
Department of Computer Science, Texas Tech University
zhenxu@ttu.edu, victor.sheng@ttu.edu

Abstract

Large language models like ChatGPT can generate human-like code, posing challenges for programming education as students may be tempted to misuse them on assignments. However, there are currently no robust detectors designed specifically to identify AI-generated code. This is an issue that needs to be addressed to maintain academic integrity while allowing proper utilization of language models. Previous work has explored different approaches to detect AI-generated text, including watermarks, feature analysis, and fine-tuning language models. In this paper, we address the challenge of determining whether a student's code assignment was generated by a language model. First, our proposed method identifies AI-generated code by leveraging targeted masking perturbation paired with comprehensive scoring. Rather than applying a random mask, areas of the code with higher perplexity are more intensely masked. Second, we utilize a fine-tuned CodeBERT to fill in the masked portions, producing subtle modified samples. Then, we integrate the overall perplexity, variation of code line perplexity, and burstiness into a unified score. In this scoring scheme, a higher rank for the original code suggests it's more likely to be AI-generated. This approach stems from the observation that AI-generated codes typically have lower perplexity. Therefore, perturbations often exert minimal influence on them. Conversely, sections of human-composed codes that the model struggles to understand can see their perplexity reduced by such perturbations. Our method outperforms current open-source and commercial text detectors. Specifically, it improves detection of code submissions generated by OpenAI's text-davinci-003, raising average AUC from 0.56 (GPTZero baseline) to 0.87 for our detector.

Introduction

The emergence of large language models has profoundly impacted software development, providing invaluable tools for programmers. These models possess the capability to autonomously generate code, which, while beneficial in many respects, poses significant challenges to programming education. There's a growing concern that students may exploit these models by merely providing a problem description and test cases. As a result, they could potentially complete code assignments without ever typing a single line of code themselves. Given the innate proficiency of these language models, they can effortlessly produce multiple, unique solutions to students' basic assignments within mere seconds. Moreover, the generated code often includes annotations and is constructed with the expertise reminiscent of seasoned programmers. Such accessibility allows students to merely copy and paste the code, followed by a direct submission. This ease of misuse gravely threatens the integrity and objectives of programming education.

Despite the rising challenges, most current detectors focus on pinpointing AI-generated text, leaving a gap in tools specifically designed for AI-generated code. In this paper, we bridge this gap by introducing a dedicated AI code detector. We then rigorously test its efficiency, comparing it against renowned open-source and commercial alternatives within a coding context.

In the prior work DetectGPT (Mitchell et al. 2023), it shows a very inspiring hypothesis and perturbation method on the textual context. DetectGPT posits that subtle modifications to AI-generated text generally result in lower log probability under the model than the original text. Conversely, slight alterations to human-written text can yield either higher or lower log probabilities than the original. This suggests that each token produced by the model often resides at the pinnacle of the log probability function curve.

The concept of code naturalness implies that a similar behavior should manifest within code contexts. Building upon these foundational insights, we refine and validate this perspective for code environments. Our hypothesis emphasizes that minor perturbations can distinctly differentiate between AI-generated and human-authored code. The core of our experiments revolves around this principle: slight modifications to AI-generated code are more likely to increase its perplexity under the model compared to the original code. On the other hand, alterations to human-written code can either raise or lower its perplexity relative to the original. Furthermore, we propose another hypothesis: Code generated by LLMs exhibits more consistent line-by-line perplexity compared to human-authored code. Our findings affirm this perspective.

We harness these two observations from the code context to construct our AIGCode detector. Our strategy seeks to discern whether a given piece of code is AI-generated by employing a two-fold approach: perturbation and scoring.

- Perturbation Mechanism: This involves using a mask modeling task to subtly modify segments of code that exhibit higher perplexity (PPL).
- Scoring Mechanism: The scoring is influenced by three metrics: perplexity of the code, standard deviation of perplexity across individual lines, and the code's burstiness.

The foundational premise of our methodology is that AI-generated code inherently possesses a low score, gauged by the combined metrics of overall perplexity, standard deviation of perplexity across code lines, and burstiness. Consequently, post-perturbation, it becomes challenging to generate samples that score lower than the original AI-generated code. In contrast, human-authored code might be perceived as more ambiguous by the model, so minor tweaks might yield an "optimized" version that bears a score lower than the original code.

We articulate our contributions in this paper as follows:

- We validate the hypothesis, previously observed in model-generated text, within the code context: perturbations tend to elevate the model's perplexity for AI-generated code but can diminish it for human-written code.
- Building on the foundation of DetectGPT, we have addressed the gap in AI-generated code detection by enhancing current zero-shot detection method and developing a specialized detector for AI-generated code. Empirical results show that our improved detector surpasses both leading open-source and commercial alternatives in the code domain.
- We delve into the robustness of AI-generated code and explore various adversarial tactics, including refined prompting, temperature adjustments, rewrites, and code blending.

Background

In this section, we delve into the domain of Pre-trained Large Language Models specifically tailored for coding languages. We introduce metrics to evaluate the proximity of code to model-generated content.

Pre-trained Large Language Models on Codes

Pre-trained language models are transforming natural language processing (NLP) with their strong performance on tasks like translation and text summarizing. This has led to interest in applying these models to programming languages. By training on source code and documentation, then fine-tuning for specific programming tasks, these models can assist with code completion, bug finding, and code generation. They leverage their pre-existing knowledge to understand and generate code. Notable research projects demonstrate the potential of pre-trained models for programming. Models like CodeBERT (Feng et al. 2020), CodeT5 (Wang et al. 2021), PolyCoder (Xu et al. 2022), and PaLM-Coder (Chowdhery et al. 2022) show early success in adapting large language models for code. By pre-training on code and documentation, they learn about programming. Fine-tuning then enables applications like automated code completion and error correction. Though still an emerging field, these models represent progress toward using the knowledge within pre-trained language models to comprehend, generate, and reason about code.

In our approach, we utilize Large Language Models on Codes (LLMCs) during the perturbation and scoring phase, specifically for tasks such as tokenizing, perplexity calculating, and mask-filling.

Perplexity and Burstiness

Perplexity and burstiness are two key metrics for assessing the quality of code generated by large language models for code (LLMCs).

Perplexity Perplexity is a measure of how well a probability model predicts a sample. In the context of code, it's used to evaluate the predictability of the next token or line of code based on a given context. A lower perplexity indicates that the model's predictions are generally more accurate, while a higher perplexity suggests the model finds the content more unpredictable.

Burstiness Burstiness in the context of code refers to the clustering or frequent appearance of certain patterns, identifiers, or constructs in a specific section of the code. For example, in a code module dealing with database operations, there might be a burst of commands and identifiers related to database queries. Analyzing burstiness in code can help in understanding patterns, detecting anomalies, or assessing code quality.

Approach

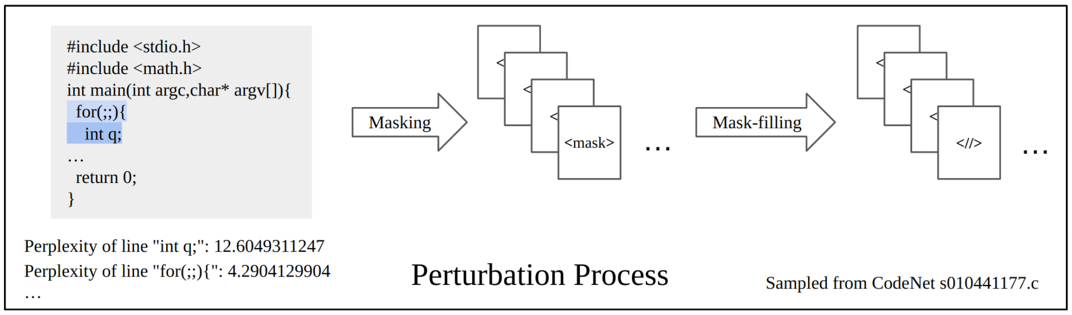
Problem Description

We are addressing the problem of detecting AI-generated code submissions and preventing the misuse of large language models in education. Our method mainly comprises three processes: perturbation, scoring, and prediction. Unlike the random perturbation and score ranking of DetectGPT, we optimize the masking process, and fine-tune mask-filling models on different code languages. Moreover, we also propose a new scoring algorithm for the ranking of original code and perturbed codes. Algorithm 1 provides a detailed description of our evaluation strategy and detection process.

Perturbation

As shown in Figure 1, the perturbation process includes masking and mask-filling, which produces several samples with minor modifications. The degree of modification depends on our mask percentage.

Masking In contrast to the indiscriminate masking technique applied by DetectGPT to textual data, our strategy adopts a more nuanced approach. Initially, we compute the Perplexity (PPL) for each line of code. Based on these calculations, a weight for masking is assigned, contingent on the PPL value of each line. Subsequently, random masking is applied. Hence, sections of code with elevated PPL values are subjected to more extensive masking, leading to more



```
#include <stdio.h>
#include <math.h>
int main(int argc,char* argv[]){
 for(;;){
  int q;
…
 return 0;
}
```

```
Perplexity of line "int q;": 12.6049311247
Perplexity of line "for(;;){": 4.2904129904
…
```

Figure 1: Illustration of the perturbation process. Weights are assigned to code segments based on line-level perplexity, higher weight means more allocated masks, followed by mask-filling task for slight modifications.

profound modifications. This approach facilitates the generation of samples that align more harmoniously with the model's comprehension, thereby optimizing the number of required samples. It's pertinent to note that we abstain from directly masking tokens that exhibit low log probability. This decision stems from the observation that models frequently assign low probabilities to specialized tokens, such as '#' or indentations integral to code formatting, which, in essence, are not informative for our purpose. Additionally, our approach offers a distinct advantage over DetectGPT's random masking strategy: it enhances the likelihood of generating samples with reduced PPL values, implying that fewer perturbed samples are requisite.

Mask-Filling We have fine-tuned the CodeBERT model using code corpora from six distinct programming languages. Then CodeBERT is employed to fill the masks introduced in the input code. Instead of relying solely on the top-scoring candidate we utilize Nucleus Sampling. This method ensures a broader spectrum of token candidates, thereby generating more varied samples.

Algorithm 1: AI-Generated Code Detection

```
1:  procedure DETECT(C)
2:      for each line $L_i$ in $C$ do // For every line in the code
3:          $\text{PPL}(L_i) \leftarrow e^{-\frac{1}{N}\sum_{j=1}^{N}\log p(w_j|w_1,w_2,\ldots,w_{j-1})}$ // Compute PPL for the line
4:      end for
5:      $W(L_i) \leftarrow$ Function of $\text{PPL}(L_i)$ // Assign mask weights based on PPL
6:      $M \leftarrow \text{Mask}(C, W)$ // Mask the code based on weights
7:      $C' \leftarrow \text{FillMask}(M)$ // Fill the masked parts
8:      $S \leftarrow \alpha \times \text{PPL} + \beta \times \text{Std}(\text{PPL}) + \gamma \times B$ // Compute the score
9:      $R \leftarrow \text{Rank}(S, C')$ // Rank the score in comparison with perturbed codes
10:     $P \leftarrow \frac{\text{count}(S > S(C))}{\text{count}(C')}$ // Calculate prediction probability
11:     if $P > 0.97$ then // Threshold check
12:         return 1 // AI generated
13:     else
14:         return 0 // Human generated
15:     end if
16: end procedure
```

Scoring and Prediction

We employ a composite strategy to evaluate code submissions, factoring in three distinct metrics: code perplexity (PPL), standard deviation of PPL across code lines, and code burstiness. Each metric is assigned a specific weight to compute an overall score. We juxtapose the scores of the original codes with their perturbed counterparts. A higher ranking for the original code often indicates a higher likelihood of it being AI-generated. The rationale behind this is that AI-generated codes inherently occupy positions at the bottom of the perplexity curve. During the perturbation process, it is challenging to produce a sample more preferable to the model than the AI-generated one, given its grammatical accuracy, coherence, and lower burstiness. Hence, perturbed versions rarely exhibit a lower score than the AI-generated samples. Conversely, human-written code might have diverse elements that the model finds perplexing. Even minor modifications can lead to 'optimized' samples with a score lower than their original versions. This process is shown in Figure 2.

Experiments Design

Dataset Preparation

In this section, we will introduce our data collection process and dataset we created.

CodeNet Dataset The CodeNet dataset (Puri et al. 2021) by IBM contains 14 million code samples from 4000 coding problems and covers over 50 programming languages, including C++, Python, and Java. Each sample provides information on problem description, size, memory use, and CPU run time. It also includes human-written solutions to these problems, showcasing different coding strategies. Building on CodeNet, we refined the data and collect AI-generated codes for each programming problems.

AIGCode Dataset The AIGCode dataset is derived from IBM's CodeNet dataset and is specifically designed to re-

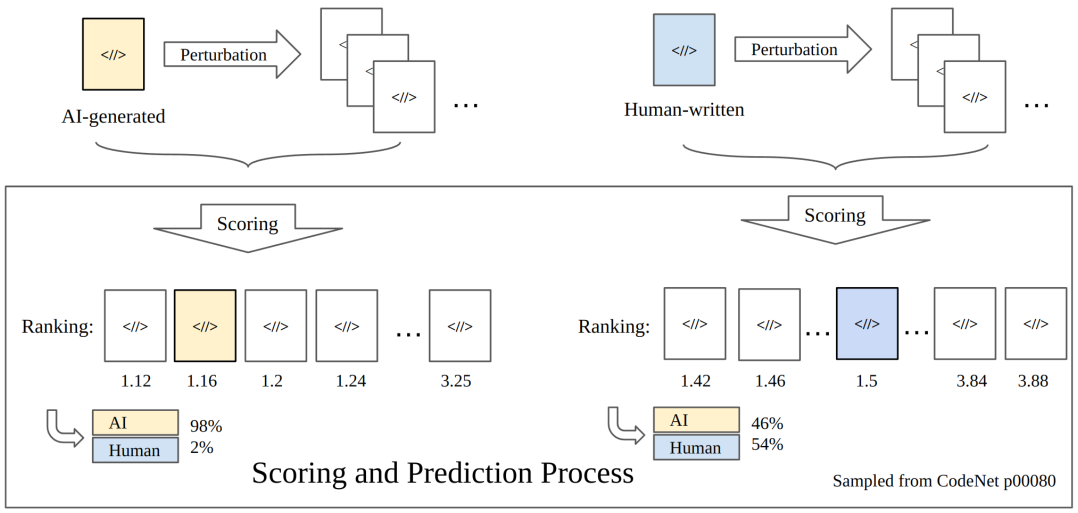


Figure 2: Illustration of the scoring and prediction procedure. The scoring considers overall perplexity, standard deviation of perplexity across code lines, and burstiness. A lower score suggests a higher likelihood of the code being generated by a model.

flect the nature of student submissions in educational programming tasks. We concentrated our efforts on six programming languages that are frequently used in academic environments: C, C++, C#, Java, JavaScript, and Python. To ensure the quality and uniformity of our dataset, we applied a data cleanup process. We filtered codes based on specific criteria: they needed to have a line length between 10 and 100, an alphanumeric character fraction exceeding 0.25, and we eliminated all comments and duplicate files. After this rigorous cleanup, we utilized 80% of the resultant CodeNet data to fine-tune CodeBERT. Simultaneously, 10% was reserved for validation purposes, and the remaining 10%, encompassing roughly 400 programming problems, was designated a segment of our test set.

Building on these 400 programming challenges, we employed the OpenAI's text-davinci-003 (OpenAI 2022) model for two main tasks: text-to-code generation and code translation. The former involves generating code from problem descriptions, while the latter translates code from one programming language to another. All AI-generated codes that successfully passed at least one test case were integrated into the test set. This resulting dataset comprises 5,214 AI-generated codes. To maintain balance, an equal number of human-written codes were selected at random from CodeNet, ensuring they matched in terms of language and programming problem.

Detector and Baseline

In this section, we introduce a range of AI-generated content detectors, from open-source to commercial solutions, that serve as baselines. Each detector possesses its own distinctive strategies and mechanisms. These detectors are compared with our AIGCode Detector on the AI-generated code dataset to benchmark performance.

GPT2-Detector Leveraging the RoBERTa (Liu et al. 2019) architecture, the GPT2-Detector (Solaiman et al. 2019) is fine-tuned specifically to identify outputs from the 1.5B-parameter GPT-2 model. This detector is trained on outputs generated using a combination of temperature and nucleus sampling. This training methodology ensures it generalizes well across outputs produced by various sampling methods.

DetectGPT DetectGPT (Mitchell et al. 2023) introduces a novel approach to discern machine-generated text through the analysis of probability curvature. Remarkably, it bypasses the need for dedicated classifiers or assembling datasets of real or generated texts. Using out-of-the-box mask-filling models, like T5 (Raffel et al. 2020) and mT5 (Xue et al. 2020), it gauges the curvature of a model's output probability distribution. In practice, DetectGPT exhibits superior discriminative capabilities over other zero-shot detection methods.

RoBERTa-QA Derived from the RoBERTa language model, RoBERTa-QA (Guo et al. 2023) is designed for text classification, particularly in question-answering contexts. By accepting paired inputs, namely a question and its corresponding answer, the model can discern AI-generated content with higher accuracy. A distinctive token is employed to merge the question and its answer, optimizing the classification process.

GPTZero Renowned as a premier AI detector, GPTZero (Tian and Cui 2023) specializes in determining if content, be it a sentence, paragraph, or entire document, originates from a large language model, for instance, ChatGPT. The detection strategy relies on metrics like perplexity and burstiness. Having been trained on a vast corpus encompassing both human and AI-generated English texts, GPTZero boasts of serving millions of users worldwide, spanning various sectors such as education, publishing, and legal.

The Writer AI Detector This freely accessible tool aims to discern AI-generated content by identifying specific textual patterns that AI models frequently produce (Writer.com 2023). The detection criterion encompasses various aspects, from recurrent phrasal patterns and sentence structures to the overarching tone of the content.

Research Questions

RQ1: How Does AIGCode Detector Compare With Current Open-Source and Commercial Text Detectors? To ensure that the AIGCode Detector stands up to the demands of current technology, we benchmark its performance against a selection of five renowned open-source and commercial text detectors.

RQ2: How Does the Perplexity Calculation Influence the Performance of AIGCode Detector? The perplexity calculation model serves as the foundation for our AIGCode Detector and can significantly impact its efficacy.

RQ3: How Robust Is the AIGCode Detector Against Attacks? In real-world applications, codes generated by Large Language Models (LLMs) often undergo modifications before submission. Users might apply regular rewrites to AI-generated codes or adjust certain hyper-parameters of LLMs to bypass detection. We designed a variety of mutation methods to try to bypass the detector.

Metrics

To comprehensively assess the effectiveness, we used following metrics:

AUC. The AUC (Area Under the Receiver Operating Characteristic Curve) score of a detector measures its ability to differentiate between AI-generated text (positive class) and human-written text (negative class). A score closer to 1.0 signifies effective differentiation, while a score around 0.5 indicates the detector's performance is equivalent to random chance.

FPR. FPR (False Positive Rate) measures the proportion of human-written code that is incorrectly identified as AI-generated by the detector.

FNR. FNR (False Negative Rate) quantifies the proportion of AI-generated code that the detector mistakenly classifies as human-written.

Bypass Rate. For the robustness test, we assess the ability of our detector to identify mutated AI-generated codes by $BypassRate = \frac{N_{bypassed}}{N_{total}}$. Here, the numerator denotes the count of AI-generated codes that, post-mutating, eluded the detector's scrutiny, while the denominator signifies the overall count of AI-generated codes subjected to mutating with the aim of circumventing the detector.

Hyper-Parameters

For fine-tuning CodeBERT, we employed 5.8G of code sourced from CodeNet, spanning 6 distinct programming languages. The training process lasted for 500,000 steps, using a batch size of 32, and was executed on two GTX 4090 GPUs. From this process, we obtained 6 distinct CodeBERT-tuned models, each tailored for a specific programming language. We conduct mask-filling task using CodeBERT with a masking percentage of 5%. For calculating perplexity, we mainly leveraged the OpenAI's text-davinci-003 model. In contrast to the DetectGPT method which required 500 samples, our approach significantly reduced the sample requirement to just 50. This efficiency is attributed to our strategic mask selection mechanism. For AI-generated codes using text-davinci-003, the temperature was set to 0.6, and top_p to 0.1. Higher temperature and top_p values increased the error rate, while lower values tended to produce more similar answers.

Main Results

In this section, we will present experiment results along with research questions in an attempt to answer each question.

RQ1: Performance of AIGCode Detector Against Existing Detectors

To evaluate the efficacy of the AIGCode Detector, we benchmark its performance against five prominent open-source and commercial text detectors. Our comparisons are conducted on the AIGCode dataset. Within the AIGCode Detector framework, we employ CodeBERT as the mask-filling model and text-davinci-003 as the primary scoring model.

For those detectors giving probability, we choose its best performance threshold. For detectors having requirement of input length, we truncate and use the prior tokens as input. Results is shown in Table 1. AIGCode Detector has relatively high AUC on all programming languages, as well as low FPR and FNR.

RQ2: Influence of Models Used to Calculate Perplexity

To investigate the role of perplexity calculation models in the effectiveness of the AIGCode Detector, we utilized a variety of Large Language Models including GPT2-xl (Radford et al. 2019), GPT-J (Wang and Komatsuzaki 2021), GPT-NeoX (Black et al. 2022), and text-davinci-003 (OpenAI 2022), to compute perplexity scores. Additionally, we examined how alterations in the masking percentage of CodeBERT can impact the detector's performance when aligned with different scoring models. In Figure 3, we chose the C language subset as our test data. Figure 3 gives insights into how different perplexity calculation models perform in terms of AUC as the masking in the test code varies. It particularly highlights the heightened sensitivity of large parameter models like 'text-davinci-003,' which are trained on specialized code corpora.

In the Table 2, we employed these models to compute perplexity and set 15% mask percentage. It shows models with larger parameters have a better grasp of code semantics and, as a result, perform better in detection.

RQ3: Robustness Against Attacks

We evaluated the robustness of the AIGCode Detector against several adversarial attack methods designed specifically to target such detectors. We selected 560 AI-generated C codes from the AIGCode dataset, all of which were accurately detected by both GPTZero and the AIGCode Detector. Subsequently, we employed four mutation strategies to

	C			C++			C#			Java			JavaScript			Python		
Detectors	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR
GPT2-Detector	0.64	0.83	0.03	0.68	0.84	0.05	0.46	0.92	0.13	0.44	0.91	0.11	0.39	0.90	0.26	0.38	0.89	0.25
DetectGPT	0.56	0.00	1.00	0.42	0.00	1.00	0.49	0.00	1.00	0.43	0.00	1.00	0.51	0.00	1.00	0.49	0.00	1.00
RoBERTa-QA	0.68	0.00	1.00	0.53	0.00	1.00	0.48	0.00	1.00	0.64	0.00	1.00	0.52	0.00	1.00	0.60	0.00	1.00
Writer	0.78	0.13	0.91	0.62	0.15	0.98	0.52	0.13	0.91	0.54	0.01	0.96	0.56	0.10	0.89	0.51	0.08	0.97
GPTZero	0.90	0.06	0.20	0.83	0.20	0.68	0.29	0.18	0.96	0.28	0.18	0.88	0.41	0.08	0.90	0.59	0.00	1.00
CGCode Detector	**0.95**	0.16	0.08	**0.88**	0.13	0.02	**0.86**	0.12	0.07	**0.82**	0.15	0.05	**0.81**	0.21	0.15	**0.92**	0.14	0.03

Table 1: Performance of Different Detectors Across Six Programming Languages.

	C			C++			C#			Java			JavaScript			Python		
Detectors	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR	AUC	FPR	FNR
GPT2-xl	0.79	0.28	0.12	0.75	0.30	0.15	0.73	0.29	0.14	0.72	0.27	0.13	0.71	0.28	0.16	0.69	0.32	0.17
GPT-J	0.82	0.25	0.18	0.76	0.27	0.19	0.76	0.24	0.20	0.77	0.26	0.21	0.79	0.25	0.22	0.78	0.24	0.23
GPT-NeoX	0.87	0.26	0.24	0.83	0.27	0.25	0.80	0.26	0.26	0.74	0.25	0.25	0.75	0.27	0.27	0.83	0.28	0.28

Table 2: Performance for Different Perplexity Calculation Models of AIGCode Detector.

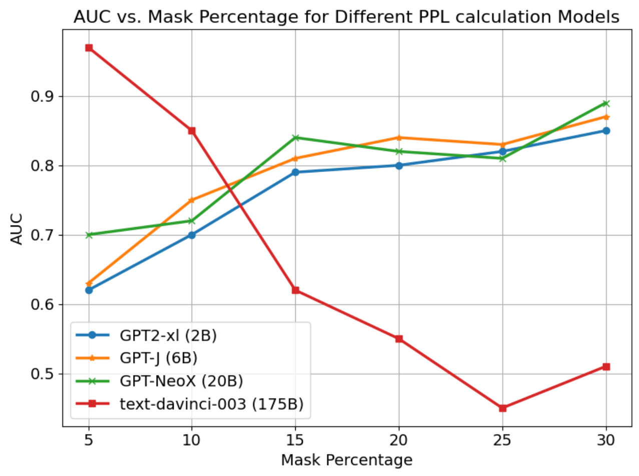


Figure 3: Illustration of the AUC vs. Mask Percentage for Various Perplexity Calculation Models. Model text-davinci-003 has a greater sensitivity on codes compared to other models. Consequently, text-davinci-003 will consistently results in an increase in the perplexity of perturbed samples while its mask percentage higher than 15%, which make almost all perturbed samples' PPL is larger than original code.

generate four derivative subsets, with the primary objective of challenging and potentially evading detection by the said detectors.

Regular Rewrite: To enhance the human-like quality of AI-generated code, we employed two primary strategies. Firstly, using a curated lexicon of function and variable names, we systematically replaced 80% AI-chosen names with alternatives from our list. Secondly, guided by specific stylistic conventions, we made targeted character replacements.

Sampling Techniques: These techniques dictate the output randomness and determinism of AI-generated content, impacting its resemblance to human-generated content. We set Temperature Sampling (Top-k) to 0.9, Nucleus Sampling (Top-p) to 0.5 to create more creative and random answers.

- *Temperature Sampling/Top-k:* By varying the sampling temperature, we can influence the randomness of the output. A higher temperature (closer to 2) produces more randomness, whereas a lower value (closer to 0) results in more deterministic outputs.
- *Nucleus Sampling/Top-p:* This approach selects tokens based on the cumulative probability. Using a smaller percentage, like 0.1, the output is constrained to the top 10% probable tokens, leading to more deterministic and coherent text.

Smarter Prompts: Effective prompting plays a pivotal role in steering LLMs towards desired outputs. Utilizing more precise and cleverly formulated prompts can induce LLMs to generate outputs that are close to human-produced content. We add several prefix-prompts for code generation task, such as: ”Please generate a code answer written in a more casual, human-like style with high randomness as if a beginner programmer or a hobbyist wrote it. I want the code to appear less formal and more like something someone might quickly jot down.”

Code Blending: Integrating AI-generated segments with human-authored parts can craft a seamless blend that challenges detection. Instead of entirely AI-generated or human-written content, merging segments from both can produce a code that retains human-like irregularities while benefiting from AI's efficiency. We truncated half of the human-written code and prompted the model with the program specification to complete the remaining portion.

Table 3 indicates ”Sampling Techniques” and ”Code Blending” strategy significantly challenge detectors, resulting in high bypass rates. The variability introduced by in-

Detector	Mutation Set	Bypass Rate
GPTZero	Regular Rewrite	0.22
	Sampling Techniques	0.32
	Smarter Prompts	0.20
	Code Blending	0.87
AIGCode Detector	Regular Rewrite	0.13
	Sampling Techniques	0.37
	Smarter Prompts	0.17
	Code Blending	0.75

Table 3: Bypass Rates for Different Detectors and Mutation Sets.

creased temperature in sampling and the hybrid nature of code blending effectively mask AI-generated code patterns, making detection difficult.

Related Work

Code Generation Using Large Language Models

AI chat-bots like ChatGPT and Claude, while designed for natural language interactions, have shown proficiency in code-related tasks due to their training on vast datasets of source code and documentation. This enables them to handle tasks such as code completion, bug fixing, and code synthesis. In a more specialized vein, OpenAI's Codex, an offshoot of GPT-3 (Brown et al. 2020), is tailored for programming tasks and shines in code generation and comprehension. A prime example of its application is GitHub Copilot (GitHub 2021), an AI-assisted coding tool co-developed by GitHub, OpenAI, and Microsoft. It intelligently offers real-time code suggestions, drawing from the user's coding context or a descriptive comment, and is adept at understanding both the active file and its associated files. Similarly, tools like CodeGen (Nijkamp et al. 2022), Amazon CodeWhisperer (Amazon Web Services 2022), and CodeGeeX (Zheng et al. 2023) harness the power of AI to generate code based on natural language inputs.

AI-Generated Text Detectors

Classical machine learning techniques, such as bag-of-words combined with logistic regression, have been foundational, with studies like those by Solaiman et al. (Solaiman et al. 2019) employing them to differentiate GPT-2 outputs from human writing. Log probability-based methods have been introduced, with Solaiman (Solaiman et al. 2019) using TGM to evaluate total log probability and GLTR (Gehrmann, Strobelt, and Rush 2019) offering statistical techniques to pinpoint differences between GPT-2 and human content. The DetectGPT (Mitchell et al. 2023) employs log probabilities produced by the targeted model and random perturbations from a generic pre-trained language model (like T5). Additionally, leveraging pre-trained language models through fine-tuning has emerged as a dominant strategy. For instance, GROVER (Zellers et al. 2019) utilizes a linear classifier on its base model to outclass other detectors, emphasizing the importance of public availability of such generators for research. Meanwhile, the fine-tuned RoBERTa model (Uchendu et al. 2020) achieves approximately 95% accuracy in detecting GPT-2-generated web pages, benefiting from its inherent bidirectional representations. Kirchenbauer et al. (Kirchenbauer et al. 2023) propose a watermarking framework for large language models. This method embeds signals into generated text that remain algorithmically detectable, yet are imperceptible to humans.

Discussion

In this section, we recognize the inherent limitations of our current methodology and explore the potential avenues for future research.

Limitations: A key limitation is the dependency on a single generative model (i.e. OpenAI's text-davinci-003), a mainstream GPT-3.5 variant trained on code. We haven't tested our approach across a diverse set of code generation models. This singular reliance could introduce biases. As the DetectGPT research suggests, when the generation and PPL calculation models are identical, detection accuracy might be slightly improved. Another limitation is our balanced dataset, the balanced dataset provides a fair baseline, illustrating model performance without the influence of class imbalance. In contrast, the unbalanced dataset reflects potential scenarios the model may encounter in real-world settings, where data is often imbalanced.

Future Work: To address the limitations identified, future research will focus on evaluating the proposed approach across a broader spectrum of code generation models, mitigating potential biases. We also plan to delve deeper into scenarios with real-world data imbalances, refining our methodology to ensure consistent and reliable detection accuracy. Besides, our approach currently depends on code rewriting, which is both time-consuming and resource-intensive. To address these challenges, we're considering the adoption of supervised models. The goal is to train a supervised model that can efficiently detect AI-generated code by recognizing specific perplexity patterns instead of just relying on code embeddings. By integrating deep learning with log probability techniques, we anticipate improved detection of AI-generated code.

Conclusion

In conclusion, we present the AIGCode Detector, an innovative tool adept at pinpointing AI-generated code assignments using perplexity analysis and targeted perturbations. Our evaluations highlight its superiority over current detectors. Our tool not only serves as a strong defense for academic integrity but also promotes the judicious use of AI in programming education. In future pursuits, we anticipate refining this tool by exploring its efficiency across a wider range of AI models and real-world applications.

References

Amazon Web Services. 2022. AWS CodeWhisperer. Code suggestions system by Amazon Web Services using a deep learning AI trained on AWS code and documentation.

Black, S.; Biderman, S.; Hallahan, E.; Anthony, Q.; Gao, L.; Golding, L.; He, H.; Leahy, C.; McDonell, K.; Phang, J.; et al. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

Chowdhery, A.; Narang, S.; Devlin, J.; Bosma, M.; Mishra, G.; Roberts, A.; Barham, P.; Chung, H. W.; Sutton, C.; Gehrmann, S.; et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

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

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

GitHub. 2021. GitHub Copilot. Code auto-completion system developed by GitHub using their private dataset of public GitHub repositories and trained via Codex from OpenAI.

Guo, B.; Zhang, X.; Wang, Z.; Jiang, M.; Nie, J.; Ding, Y.; Yue, J.; and Wu, Y. 2023. How Close is ChatGPT to Human Experts? Comparison Corpus, Evaluation, and Detection. *arXiv preprint arxiv:2301.07597*.

Kirchenbauer, J.; Geiping, J.; Wen, Y.; Katz, J.; Miers, I.; and Goldstein, T. 2023. A watermark for large language models. *arXiv preprint arXiv:2301.10226*.

Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; and Stoyanov, V. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Mitchell, E.; Lee, Y.; Khazatsky, A.; Manning, C. D.; and Finn, C. 2023. Detectgpt: Zero-shot machine-generated text detection using probability curvature. *arXiv preprint arXiv:2301.11305*.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2022. text-davinci-003. https://platform.openai.com/docs/guides/gpt/completions-api. Accessed: 2023-10-01.

Puri, R.; Kung, D. S.; Janssen, G.; Zhang, W.; Domeniconi, G.; Zolotov, V.; Dolby, J.; Chen, J.; Choudhury, M.; Decker, L.; et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 1035.

Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I.; et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8): 9.

Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; and Liu, P. J. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140): 1–67.

Solaiman, I.; Brundage, M.; Clark, J.; Askell, A.; Herbert-Voss, A.; Wu, J.; Radford, A.; Krueger, G.; Kim, J. W.; Kreps, S.; et al. 2019. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203*.

Tian, E.; and Cui, A. 2023. GPTZero: Towards detection of AI-generated text using zero-shot and supervised methods.

Uchendu, A.; Le, T.; Shu, K.; and Lee, D. 2020. Authorship attribution for neural text generation. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, 8384–8395.

Wang, B.; and Komatsuzaki, A. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax. Accessed: 2023-10-10.

Wang, Y.; Wang, W.; Joty, S.; and Hoi, S. C. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Writer.com. 2023. The Writer AI Detector. https://writer.com/ai-content-detector/. Accessed: 2023-10-01.

Xu, F. F.; Alon, U.; Neubig, G.; and Hellendoorn, V. J. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10.

Xue, L.; Constant, N.; Roberts, A.; Kale, M.; Al-Rfou, R.; Siddhant, A.; Barua, A.; and Raffel, C. 2020. mT5: A massively multilingual pre-trained text-to-text transformer. *arXiv preprint arXiv:2010.11934*.

Zellers, R.; Holtzman, A.; Rashkin, H.; Bisk, Y.; Farhadi, A.; Roesner, F.; and Choi, Y. 2019. Defending against neural fake news. *Advances in neural information processing systems*, 32.

Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Wang, Z.; Shen, L.; Wang, A.; Li, Y.; et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.