# Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers

Yuling Shi[1], Hongyu Zhang[2], Chengcheng Wan[3], Xiaodong Gu[1*]

[1]Shanghai Jiao Tong University, [2]Chongqing University, [3]East China Normal University

Accepted at ICSE 2025

# Large Language Models for Code

- LLMs like Codex and ChatGPT have been trained on vast code repositories, enabling them to generate code that closely resembles human-written code.

# Plagiarism and Authenticity Concerns

- These models have shown significant promise in automating software engineering tasks, but also blur the line between human and machine authorship.
- Students and interviewees may use these LLMs to generate code solutions, raising concerns about plagiarism and code authenticity.

# Machine-Generated Code Detection

- Previous methods like DetectGPT have been successful in detecting machine-generated text by analyzing the likelihood score discrepancies between original and perturbed texts.
- However, these methods yield suboptimal results when applied to code due to the strict syntactic rules that govern programming languages.
- There is a need for a detection method that captures code structure and style to detect machine-generated code effectively.

# Contribution

- We present the first comprehensive empirical analysis of machine- and human-authored code, focusing on lexical diversity, conciseness, and naturalness.
- We propose DetectCodeGPT, a novel zero-shot method for detecting machine-generated code that leverages stylized perturbations to capture the stylistic differences between machine- and human-authored code.
- We evaluate DetectCodeGPT on a diverse set of CodeLLMs and demonstrate its effectiveness in distinguishing between machine- and human-authored code.

# Empirical Analysis: Study Design

- We conducted a comparative analysis of machine- and human-authored code, focusing on 3 aspects:
  - **Lexical diversity**: the range of vocabulary, including variable names, functions, and reserved words.
  - **Conciseness**: the number of tokens and lines of code, reflecting verbosity and complexity.
  - **Naturalness**: how predictably a token appears in a given context, using token likelihood and rank.

# Empirical Analysis: Dataset

- **Source:** 10,000 Python functions sampled from the CodeSearchNet corpus.
- **Model:** We engaged the CodeLlama(7B) as the code generation model.
- **Prompt:** We use the function signatures and their accompanying function-level comments as prompts like in HumanEval.

# Results on Lexical Diversity

Table II: Top 50 tokens from human- and machine-authored code from CodeLlama

| Rank | Human-Authored Tokens | Machine-Authored Tokens |
|---|---|---|
| 1–10 | **. ( ) = ’ , : self ”** [ | **. - , ( ) self : ” ’ if** |
| 11–20 | **] if return** in for not None 0 1 == | **= return** not [ def **raise** isinstance ] == path |
| 21–30 | else + is name { } path data **raise** - | name 0 **__class__ __name__** None os { / } % |
| 31–40 | try * os len format get and True value isinstance | else **TypeError** str ‘ __init__ is > // the |
| 41–50 | args key % np i x kwargs except False or | in 1 ; value kwargs #include + __str__ for **ValueError** |

> **Finding 1:** LLM's code focuses more on exception handling and object-oriented principles, indicating a bias towards error prevention and standard programming practices.

# Results on Lexical Diversity

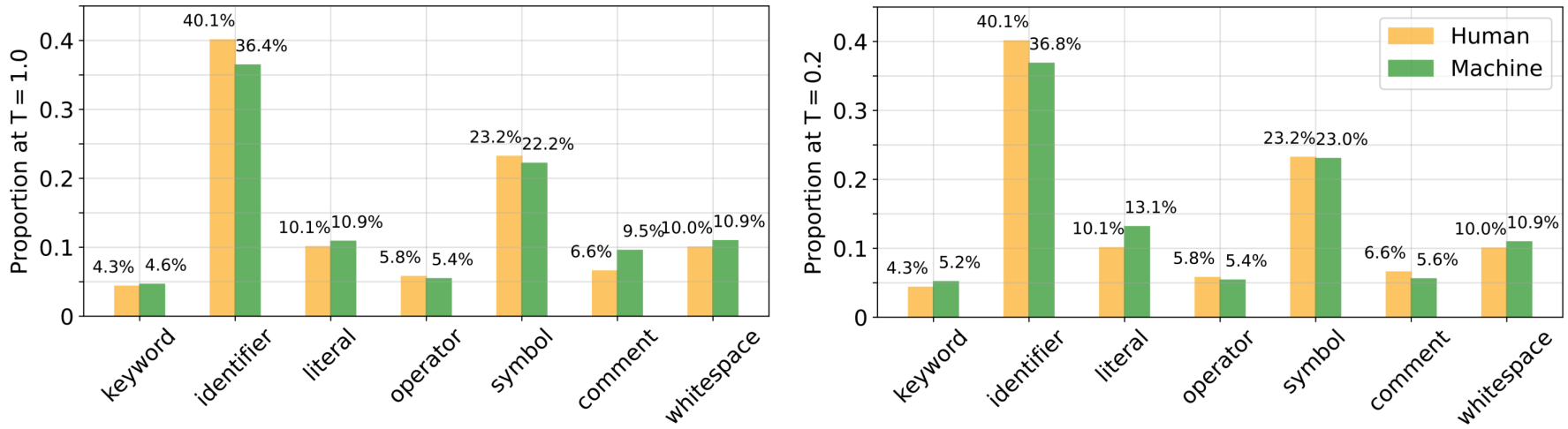
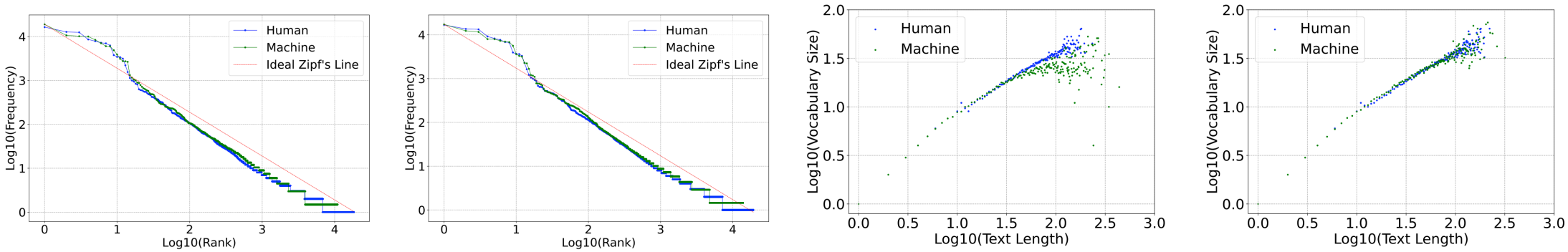



Figure 1: Syntax element distribution of the code corpus

> **Finding 2:** LLM's code tends to use fewer identifiers and more literals, with an increase in comments at higher temperatures, reflecting a more direct approach to data processing and a greater emphasis on documentation.

# Results on Lexical Diversity

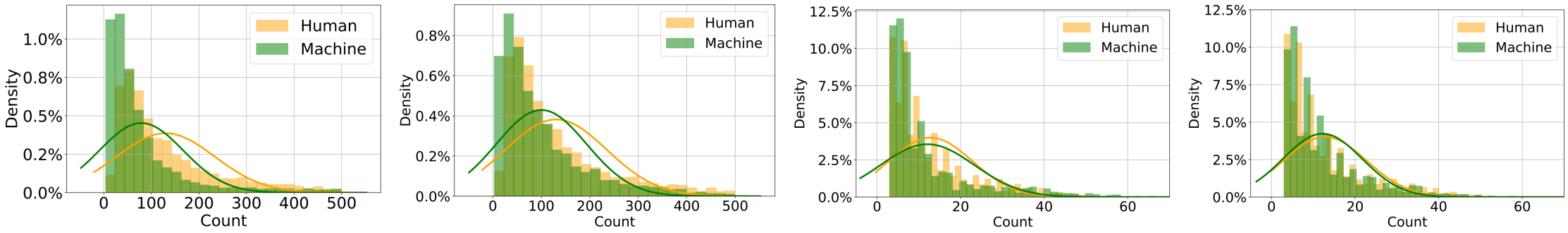



(a) Zipf's Law ($T = 0.2$) (b) Zipf's Law ($T = 1.0$) (c) Heaps' Law ($T = 0.2$) (d) Heaps' Law ($T = 1.0$)

Figure 2: Comparison of Zipf's and Heaps' laws on machine- and human-authored code

> **Finding 3:** LLM's code shows a preference for a limited set of frequently-used tokens, in contrast to the richer diversity in human code, suggesting a more formulaic approach to coding.

# Results on Conciseness




(a) Number of tokens ($T = 0.2$) (b) Number of tokens ($T = 1.0$) (c) Number of lines ($T = 0.2$) (d) Number of lines ($T = 1.0$)

Figure 3: Distribution of code length for machine- and human-authored code

> **Finding 4:** LLM's code is generally more concise, with fewer tokens and lines, possibly due to training biases towards efficiency, while human code is more varied, reflecting individual stylistic choices.

# Results on Naturalness

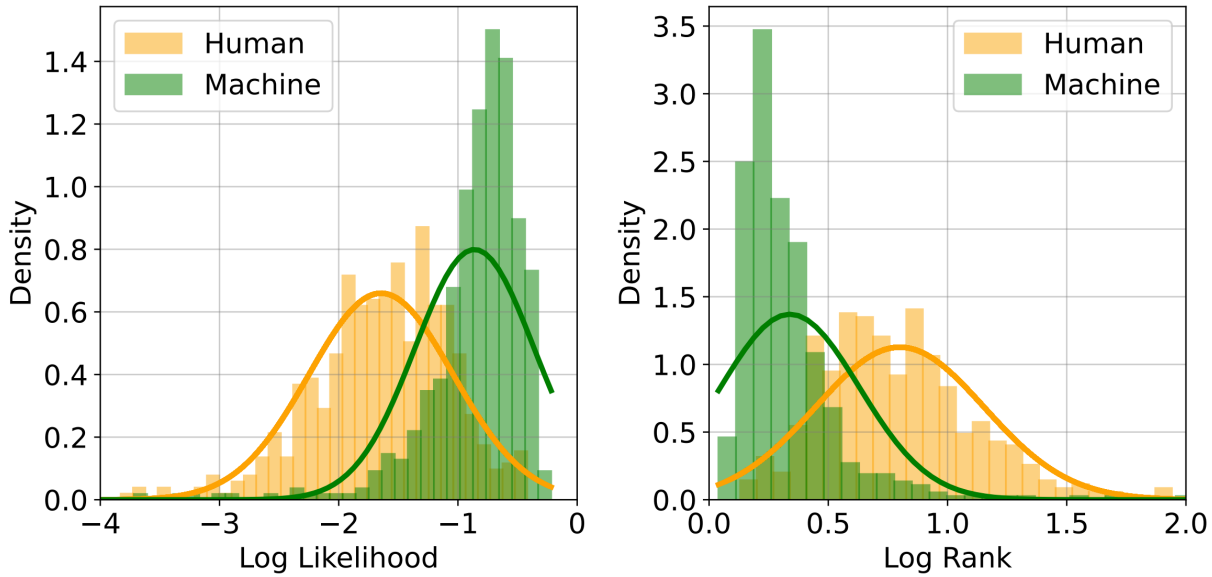



Figure 4: Distribution of naturalness scores

> **Finding 5:** LLM's code exhibits higher "naturalness" compared to human code, with the most significant differences in stylistic tokens like comments and whitespaces, indicating a more standardized and predictable coding style in machines.

# Results on Naturalness

Table III: The naturalness of different categories of syntax elements. Statistical significance $p < 0.001$.

| Category | Log Likelihood | | | Log Rank | | |
|---|---|---|---|---|---|---|
| | Machine | Human | Δ | Machine | Human | Δ |
| keyword | -1.701 | -2.128 | 0.428 | 0.837 | 1.053 | 0.217 |
| identifier | -0.459 | -0.874 | 0.415 | 0.163 | 0.378 | 0.215 |
| literal | -0.506 | -1.364 | 0.858 | 0.152 | 0.630 | 0.479 |
| operator | -0.938 | -1.835 | 0.897 | 0.367 | 0.872 | 0.504 |
| symbol | -0.868 | -1.639 | 0.771 | 0.321 | 0.781 | 0.460 |
| comment | -1.503 | -3.028 | 1.525 | 0.608 | 1.610 | 1.002 |
| whitespace | -1.131 | -2.740 | **1.609** | 0.429 | 1.441 | **1.012** |
| ALL | -0.827 | -1.658 | 0.831 | 0.319 | 0.811 | 0.492 |

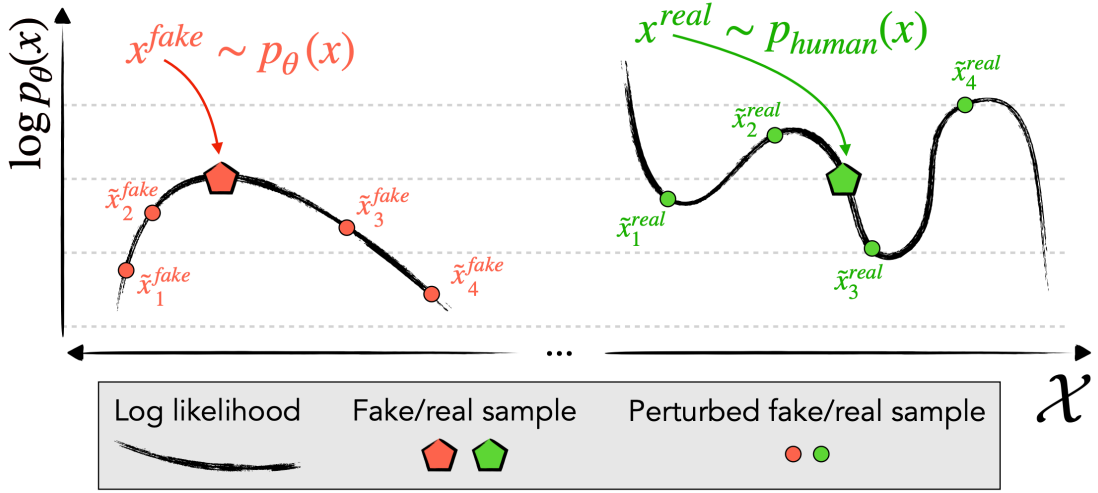
> **Finding 5:** LLM's code exhibits higher "naturalness" compared to human code, with the most significant differences in stylistic tokens like comments and whitespaces, indicating a more standardized and predictable coding style in machines.

# Machine-Generated Code Detection Problem

- **Zero-Shot**: Identifying whether code is machine-generated without collecting labeled training data.
- **Approach**: Utilizing the 'white box' setting to evaluate code probability $\log p_\theta(x)$ and leveraging generic pre-trained models for perturbation.
- **Goal**: Effectively classify machine-generated code from human-written code without the need for labeled training data.

# DetectCodeGPT Methodology

- **Observation**: Machine code differs significantly in its use of stylistic tokens compared to human-authored code.
- **Hypothesis**: By perturbing code with stylized changes:
  - The naturalness of human-written code won't change much because of its inherent randomness.
  - Machine-generated code will exhibit more significant changes in naturalness since it's usually more natural.



# DetectCodeGPT Methodology

- **Idea**: After perturbing code with stylized changes, we can detect machine-generated code by observing the changes in naturalness.
- **Naturalness**: We use the NPR (Normalized Perturbed LogRank) score to quantify the naturalness of code:

$$\mathbf{NPR}\left(x, p_\theta, q\right) \triangleq \frac{\mathbb{E}_{\tilde{x}\sim q(\cdot|x)} \log r_\theta\left(\tilde{x}\right)}{\log r_\theta(x)}$$

where $\log r_\theta(x)$ is the logarithm of the rank order of text $x$ sorted by likelihood under model $p_\theta$, and $\tilde{x}$ is a perturbed version of $x$ from the perturbation function $q$.

# Perturbation Strategy

- **Space Insertion**: Identify locations in the code to insert spaces. The number of spaces at each location $c$ is sampled from a Poisson distribution:

$$N_{spaces}(c) \sim P(\lambda_{spaces})$$

- **Newline Insertion**: Select lines to insert newlines. The number of newlines $l$ is also determined by a Poisson distribution:

$$N_{newlines}(l) \sim P(\lambda_{newlines})$$

> Will be much faster than using an LLM to Mask-Fill the code like in DetectGPT.

# Example of Perturbation

- **Before Perturbation**:

```
def calculate_tax(income,tax_rate):
    if income>10000:
        return income*0.2
    else:
        return income*0.1
```

- **After Perturbation**:

```
def calculate_tax(income, tax_rate):

    if income  > 10000:
        return income*0.2

    else:
        return income * 0.1
```

> Such perturbation captures human's randomness in code writing.

# Experimental Setup

- **Metric:** We use the AUROC score.

  > This metric eliminates the need for a fixed threshold and provides a comprehensive evaluation of the detection performance.

- **Dataset:** We used the CodeSearchNet and The Stack datasets.

  > Both dataset represent a diverse set of code snippets from real-world applications, ensuring the generalizability of our method.

# RQ1: Detection Performance

Table IV: Performance (AUROC) of various detection methods. Statistical significance $p < 0.001$.

| Dataset | Code LLM | Detection Methods | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $\log p(x)$ | Entropy | Rank | Log Rank | DetectGPT | LRR | NPR | GPTSniffer | DetectCodeGPT |
| CodeSearchNet ($T = 0.2$) | Incoder (1.3B) | 0.9810 | 0.1102 | 0.8701 | 0.9892 | 0.4735 | 0.9693 | 0.8143 | 0.9426 | **0.9896** |
| | Phi-1 (1.3B) | 0.7881 | 0.4114 | 0.6409 | 0.7513 | 0.7210 | 0.4020 | 0.7566 | 0.3855 | **0.8287** |
| | StarCoder (3B) | 0.9105 | 0.2942 | 0.7585 | 0.9340 | 0.6949 | 0.9245 | 0.9015 | 0.7712 | **0.9438** |
| | WizardCoder (3B) | 0.9079 | 0.2930 | 0.7556 | 0.9120 | 0.6450 | 0.7975 | 0.8677 | 0.7433 | **0.9345** |
| | CodeGen2 (3.7B) | 0.7028 | 0.4411 | 0.7328 | 0.7199 | 0.6051 | 0.7997 | 0.6177 | 0.5327 | **0.8802** |
| | CodeLlama (7B) | 0.8850 | 0.3174 | 0.7265 | 0.9016 | 0.8212 | 0.8332 | 0.5890 | 0.7496 | **0.9095** |
| CodeSearchNet ($T = 1.0$) | Incoder (1.3B) | 0.7724 | 0.4167 | 0.7797 | 0.7876 | 0.6258 | 0.7427 | 0.6801 | 0.6761 | **0.7882** |
| | Phi-1 (1.3B) | 0.6118 | 0.4588 | 0.5709 | 0.6299 | 0.7492 | 0.4528 | 0.7912 | 0.4158 | **0.8365** |
| | StarCoder (3B) | 0.6574 | 0.4844 | 0.6987 | 0.6822 | 0.6505 | **0.7050** | 0.6751 | 0.6299 | 0.6918 |
| | WizardCoder (3B) | 0.8319 | 0.3363 | 0.7273 | 0.8338 | 0.5972 | 0.6965 | 0.7516 | 0.7068 | **0.8392** |
| | CodeGen2 (3.7B) | 0.4484 | 0.6263 | 0.6584 | 0.4632 | 0.4797 | 0.5530 | 0.5208 | 0.4024 | **0.6798** |
| | CodeLlama (7B) | 0.6463 | 0.4855 | 0.6759 | 0.6656 | 0.6423 | 0.6768 | 0.6515 | 0.6442 | **0.7239** |
| The Stack ($T = 0.2$) | Incoder (1.3B) | 0.9693 | 0.1516 | 0.8747 | 0.9712 | 0.6061 | 0.9638 | 0.8571 | 0.9291 | **0.9727** |
| | Phi-1 (1.3B) | 0.8050 | 0.4318 | 0.6766 | 0.7622 | 0.7295 | 0.4022 | 0.8106 | 0.4640 | **0.8578** |
| | StarCoder (3B) | 0.9098 | 0.3077 | 0.7843 | **0.9329** | 0.6824 | 0.9135 | 0.9233 | 0.7715 | 0.9274 |
| | WizardCoder (3B) | 0.9026 | 0.3196 | 0.7963 | 0.9010 | 0.6385 | 0.7742 | 0.8574 | 0.7794 | **0.9243** |
| | CodeGen2 (3.7B) | 0.7171 | 0.4051 | 0.7930 | 0.7301 | 0.5288 | 0.7604 | 0.5670 | 0.4520 | **0.8513** |
| | CodeLlama (7B) | 0.8576 | 0.3565 | 0.7366 | 0.8793 | 0.8087 | 0.8358 | 0.5436 | 0.7619 | **0.8852** |
| The Stack ($T = 1.0$) | Incoder (1.3B) | 0.7310 | 0.4591 | 0.7673 | 0.7555 | 0.6124 | 0.7446 | 0.6787 | 0.6846 | **0.7833** |
| | Phi-1 (1.3B) | 0.7841 | 0.4205 | 0.6666 | 0.7475 | 0.6718 | 0.4106 | 0.7755 | 0.4984 | **0.8376** |
| | StarCoder (3B) | 0.6333 | 0.5025 | 0.7010 | 0.6609 | 0.5896 | 0.7080 | 0.6638 | **0.7243** | 0.6890 |
| | WizardCoder (3B) | 0.8293 | 0.3459 | 0.7484 | 0.8223 | 0.6377 | 0.6436 | 0.7929 | 0.7766 | **0.8384** |
| | CodeGen2 (3.7B) | 0.4816 | 0.6046 | 0.5631 | 0.4956 | 0.4337 | 0.5740 | 0.5178 | 0.4265 | **0.6595** |
| | CodeLlama (7B) | 0.5929 | 0.5260 | 0.6451 | 0.6091 | 0.6116 | 0.6365 | 0.6226 | **0.7494** | 0.6660 |
| Average | - | 0.7649 | 0.3961 | 0.7228 | 0.7724 | 0.6357 | 0.7050 | 0.7178 | 0.6507 | **0.8308** |

> DetectCodeGPT significantly outperformed state-of-the-art techniques, with an improvement of 7.6% in terms of AUC.

# RQ2: Different Perturbation Strategies

Table V: Performance of different perturbation strategies

| Perturb. Type | MLM | Newline | Space | Newline&Space |
|---|---|---|---|---|
| $T = 0.2$ | 0.5436 | 0.8703 | 0.8639 | **0.8852** |
| $T = 1.0$ | 0.6226 | 0.6453 | 0.6504 | **0.6660** |

> The combined perturbation strategy of newline and space insertions in DetectCodeGPT proves to be the most effective.

# RQ3: Impact of Perturbation Count

Table VI: Impact of varying the number of perturbations

| #Perturbations | 10 | 20 | 50 | 100 | 200 |
|---|---|---|---|---|---|
| $T = 0.2$ | 0.6537 | 0.8825 | 0.8852 | **0.8855** | 0.8846 |
| $T = 1.0$ | 0.5558 | 0.6584 | 0.6660 | 0.6660 | **0.6662** |

> A relatively small number of perturbations (i.e. 20) is sufficient for DetectCodeGPT to achieve robust detection performance, highlighting the efficiency of the method.

# RQ4: Cross-Model Detection


Figure 6: Cross-model detection performance

> It proved to be a model-free and robust method against model discrepancies, making it viable for real-world applications.

# Case Study

```
import os
import re

if not os.path.isdir(path):
        raise ValueError('%s is not a directory' % path)

if not isinstance(include, (list, tuple)):
        include = [include]
if not isinstance(exclude, (list, tuple)):
        exclude = [exclude]

if not show_all:
        exclude.append(r'\.pyc$')

...
```

Truth: 🤖
log $p(x)$: 🧑
Log Rank: 🧑
DetectCodeGPT: 🤖

(a) Example 1

```
args_names = []
args_names.extend(function_spec.args)
if function_spec.varargs is not None:
        args_names.append(function_spec.args)

args_check = {}

for arg_name in arg_specs.keys():
        if arg_name not in args_names:
                args_check[arg_name] = self.check(
                        arg_specs[arg_name], arg_name
                )
return args_check
```

Truth: 🧑
log $p(x)$: 🤖
Log Rank: 🤖
DetectCodeGPT: 🧑

(b) Example 2

```
save_test = random() > 0.8
audio = load_audio(fn)
num_chunks = len(audio)//chunk_size

listener.clear()

for i, chunk in enumerate(chunk_audio(audio, chunk_size)):
        print('\r' + str(i * 100./num_chunks) + '%')
        buffer = update((buffer[len(chunk):], chunk))
        conf = listener.update(chunk)
```

Truth: 🧑
log $p(x)$: 🤖
Log Rank: 🤖
DetectCodeGPT: 🧑

(c) Example 3

```
# If num is 0.
if (n == "0"):
        return 0

# Count sum of digits under mod 9
answer = 0
for i in range(0, len(n)):
        answer = (answer + int(n[i])) % 9

# If digit sum is multiple of 9, answer
# 9, else remainder with 9.
if(answer == 0):
        return 9
else:
        return answer % 9
```

Truth: 🧑
log $p(x)$: 🤖
Log Rank: 🤖
DetectCodeGPT: 🤖

(d) Example 4

Figure 5: Examples of machine- and human-authored code snippets with corresponding predictions.

> DetectCodeGPT effectively captures coding style nuances through stylized perturbation to detect machine-generated code.

# Future Directions

- Our work on DetectCodeGPT is just the beginning, and there are numerous avenues for future research and development.
    - **Language:** Investigating the differences in code patterns across other programming languages and their impact on detection performance.
    - **Performance:** Improving the detection of machine-generated code at higher generation randomness levels like $T = 1.0$.
    - **Attack:** Design adversarial attacks to evade detection by DetectCodeGPT.
    - **Defense:** Studying defense strategies against attacks in the context of machine-generated code detection.

# Conclusion

- Through a rigorous empirical analysis, we identified distinct patterns in machine- and human-authored code.
- DetectCodeGPT leverages the insights from this analysis to perturb the codes with stylized changes, and detect machine-generated code effectively.
- This work contributes to maintaining the integrity and authenticity of software development, ensuring transparency in code authorship.

# Thank you for your attention

Our data and code are available at https://github.com/YerbaPage/DetectCodeGPT

Feel free to reach out for any question or collaboration at yuling.shi@sjtu.edu.cn