Gotta Catch'em All!: Multi-Generator and Multi-Lingual Benchmark for Detecting LLM-Generated Code Snippets

Anonymous ACL submission

Abstract

The advent of commercial services based on large language models (LLMs), such as ChatGPT and Copilot, has significantly enhanced software development productivity. Despite their widespread adoption and benefits, concerns regarding the security vulnerabilities of LLM-generated code snippets, potential copyright and licensing infringements, and academic cheating. Recognizing the importance of detecting LLM-generated code snippets, we introduce the first benchmark, DeCo (Detecting Code generated by LLMs), aimed at addressing these challenges. DeCo comprises a dataset of 246K samples across four programming languages: C, C++, Java, and Python, generated by two commercial LLMs, ChatGPT and Gemini-Pro, and two open-source, code-specialized LLMs, WizardCoder and DeepSeek-Coder. We formulate two key tasks based on the DeCo: (1) binary detection to discern whether a given code snippet was written by a human or an LLM, and (2) multi-class detection to identify the specific generator among humans and the four LLMs. We conduct extensive experiments evaluating 13 detection methods on the DeCo dataset.

1 Introduction

The proliferation of LLMs has ushered in a new era of technological advancements, particularly in the realms of generating natural language texts and code snippets (Zan et al., 2022). Commercial services such as ChatGPT and Copilot, which are based on LLMs, have become essential tools for improving software development productivity and aiding in the learning process for students studying programming. Despite their widespread adoption and numerous benefits, these advancements also come with their own set of challenges. As the utility of LLMs continues to expand, so do concerns surrounding the security vulnerabilities (Pearce et al., 2022; Sandoval et al., 2023) inherent in LLM-generated code snippets. Issues such as potential copyright[1] and licensing[2] infringements have emerged as pressing challenges within the community. Additionally, due to the risk of students cheating on exams or committing acts of plagiarism in assignments using ChatGPT, some universities have restricted the use of ChatGPT[3]. These concerns highlight the urgent need for mechanisms capable of detecting LLM-generated code snippets (Wang et al., 2023a; Lee et al., 2023; Nguyen et al., 2023; Yang et al., 2023).

In response to these growing concerns, we present DeCo, a pioneering benchmark specifically designed to address the challenges of identifying LLM-generated code snippets. DeCo is a benchmark consisting of 246K code snippets written in four major programming languages: C, C++, Java, and Python. We construct DeCo using two commercial LLMs, ChatGPT and Gemini-Pro, and two code-specialized open-source LLMs, WizardCoder (Luo et al., 2023) and DeepSeek-Coder (Guo et al., 2024). First, we collect entirely human-written code snippets by gathering software projects from Github repositories created before the emergence of code-generating LLMs. Then, based on the collected human-written code snippets, we generate LLM-written code snippets using four LLMs. We formulate two key tasks based on the DeCo benchmark: (1) **LLM-generated code detection.** This task aims to discern whether a given code snippet was written by a human or LLM, and (2) **Identifying the authors of code snippets.** This task seeks to identify the specific generator among humans and the four LLMs (*i.e.*, authorship attribution).

We conduct extensive experiments evaluating

[1] https://felixreda.eu/2021/07/github-copilot-is-not-infringing-your-copyright/

[2] https://www.techtarget.com/searchsoftwarequality/news/252526359/Developers-warned-GitHub-Copilot-code-may-be-licensed

[3] https://www.universityworldnews.com/post.php?story=20230222132357841

13 detection methods on DeCo benchmark. Experimental results demonstrate that state-of-the-art methods struggle to distinguish between human-written code snippets and LLM-generated code snippets. In addition, we conduct the following analyses: (1) an examination of linguistic features in code snippets written by humans and code snippets generated by LLM, and (2) an exploration of the impact of natural language comments in the task of identifying the authors of code snippets. Our contributions can be summarized as follows:

- We construct the first benchmark, DeCo, for LLM-generated code snippets detection. DeCo consists of 246K samples across four programming languages, generated using four distinct LLMs.
- Based on the DeCo benchmark, we formulate two tasks: (1) LLM-generated code detection, and (2) Identifying the authors of code snippets.
- We conduct extensive experiments evaluating 13 detection methods including state-of-the-art methods on the DeCo benchmark.
- Through analysis, we identify distinct linguistic features between human-written code snippets and those generated by LLMs.

2 Related Work

2.1 Datasets for Detecting Machine-Generated Text

Guo et al. (2023) constructed the HC3 dataset, which consists of 40K questions and their answers from human experts and ChatGPT. The HC3 dataset is composed of both English and Chinese, and its questions cover open-domain, financial, medical, and psychological areas. Su et al. (2023b) broadened the scope of the HC3 dataset, originally centered on question-answering, by developing the HC3 Plus dataset, which incorporates data suited for summarization, translation, and paraphrasing tasks. Wang et al. (2024) constructed the M4 dataset that consists of multi-generator, multi-domain, and multi-lingual corpus.

2.2 Methods for Detecting Machine-Generated Text

Watermarking Methods Watermarking methods (Kirchenbauer et al., 2023; Kuditipudi et al., 2023) enable the effective detection of machine-generated text by injecting detectable patterns into the generated text. These methods require direct intervention in the decoding process of the language model to leave a watermark in the generated text.

White-Box Zero-shot Detection Methods White-box zero-shot detection methods detect machine-generated text based on the log probability (Mitchell et al., 2023) or perplexity (Hans et al., 2024) calculated from the given text, without any additional training process. White-box detection methods require full or partial access to the generator. Recently, Mireshghallah et al. (2024) reported that using a surrogate model as a detection can achieve satisfactory detection performance.

Black-Box Zero-shot Detection Methods Black-box zero-shot detection methods only require the target text. Zhu et al. (2023) proposed a paraphrasing-based method using ChatGPT. Based on the hypothesis that machine-generated text aligns better with the generation logic and statistical patterns learned by ChatGPT than human-written text, they proposed a method to identify the original text as machine-generated if the paraphrased text is similar to the original text.

2.3 Methods for Detecting Machine-Generated Code Snippet

Lee et al. (2023) pointed out that the quality of generated code snippets is degraded when watermarking methods are applied to the process of code generation. They suggested removing low-entropy segments at injecting watermarks. Yang et al. (2023) proposed DetectGPT4Code, a modification of the existing machine-generated text detection method, DetectGPT (Mitchell et al., 2023), using a code-specialized surrogate model. Wang et al. (2023a) evaluated whether existing machine-generated text detectors are effective at detecting machine-generated code snippets. They reported that the existing detectors exhibit diminished efficacy in identifying code snippets in contrast to their performance in detecting natural language texts. Previous studies conducted experiments using existing datasets for code-related downstream tasks, considering the code snippets included in these datasets to have been written by humans. However, we cannot be certain that all code snippets present in existing datasets were authored by humans. (Wang et al., 2023a). Therefore, we collect code snippets from GitHub repositories created before the emergence of LLMs.

DeCo-Binary	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder	Total
C	25,907	16,855	12,151	8,081	5,288	**68,282**
C++	14,441	5,077	10,850	5,689	3,579	**39,636**
Java	23,903	7,234	19,293	12,614	9,216	**72,260**
Python	21,545	5,699	13,789	13,796	11,101	**65,930**
Total	**85,796**	**34,865**	**56,083**	**40,180**	**29,184**	**246,108**

DeCo-Multi	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder	Total
C	457	457	457	457	457	**2,285**
C++	385	385	385	385	385	**1,925**
Java	1,494	1,494	1,494	1,494	1,494	**7,470**
Python	1,935	1,935	1,935	1,935	1,935	**9,675**
Total	**4,271**	**4,271**	**4,271**	**4,271**	**4,271**	**21,355**

Table 1: Data statistics of DeCo-Binary and DeCo-Multi.

3 DeCo Dataset

We present DeCo, the first benchmark for LLM-generated code detection. DeCo features the following characteristics.

Large-Scale: DeCo consists of a total of 246K human-written and LLM-generated code snippets.

Multi-Lingual: DeCo consists of code snippets written in four programming languages: C, C++, Java, and Python.

Multi-Generator: When constructing the DeCo dataset, we utilize the following four LLMs to generate LLM-generated code snippets: 1) Commercial LLMs: ChatGPT and Gemini-Pro; 2) Open-source, code-specialized LLMs: WizardCoder and DeepSeek-Coder.

Two Tasks: The DeCo benchmark consists of two subsets supporting the following two tasks: 1) DeCo-Binary: A dataset for binary detection tasks determining whether a given code snippet was written by a human or an LLM; 2) DeCo-Multi: A dataset for multi-class classification tasks predicting which generator wrote a given code snippet.

3.1 Human-Written Code Snippets Collection

We collect code snippets which are fully human-written by gathering code snippets from GitHub repositories created before the emergence of LLMs. Specifically, we collect repositories created between January 1, 2019, at 00:00 and January 1, 2020, at 00:00[4] , which have the MIT license and contain code snippets written in C, C++, Java, and Python. Afterward, we retain only the code snippets from the collected ones that consist solely of ASCII characters and have token counts ranging from 50 to 950. Additionally, we remove code snippets with licenses other than Apache, BSD, and MIT.

[4]We consider GPT-3, released in June 2020, as the first LLM with code generation capabilities.

3.2 LLM-Generated Code Snippets Construction

We construct LLM-generated code snippets by providing human-written code snippets and instructing the LLMs to rewrite them. We prompt the following LLMs to construct LLM-generated code snippets: 1) **OpenAI ChatGPT**[5]: is a conversational AI model, known for its exceptional ability to understand context and generate natural responses during conversations; 2) **Google Gemini-Pro**[6]: is a multi-modal generative AI model that was unveiled in 2023; 3) **WizardCoder**[7]: is an instruction fine-tuned open-source LLM, demonstrating superior performance across various code-related downstream tasks; 4) **DeepSeek-Coder**[8]: is an open-source LLM trained on project-level source code corpus using a fill-in-the-blank pre-training objective. We provide each LLM with a human-written code snippet and instruct it to revise the code snippet while maintaining the original programming language and functionality of the code snippet. The prompt we used can be found in Figure 1 of Appendix A. We observe the characteristics of LLMs and the types of errors they produce during the code generation process. We discuss these in Appendix B and C, respectively.

3.3 Data Filtering and Cleaning

If the code snippets generated by LLMs are less than 30% of the length of the original human-written code snippets, we consider such code snippets to have different functionalities from human-written code snippets and therefore remove them. We compute the similarity between LLM-generated code snippets and human-written code snippets based

[5]GPT-3.5-turbo-1106
[6]Gemini 1.0
[7]WizardCoder-33B-V1.1
[8]DeepSeek-Coder-33B-instruct

on the longest common subsequence (LCS) and then calculate percentiles of this similarity. Among the code snippets generated by LLMs, those with similarity scores of 75th percentile or higher are considered to have undergone minimal modification from human-written code snippets, and are thus removed. Some LLM-generated code snippets have a prefix such as `cpp` at the beginning, indicating the programming language of the code snippet. As this can serve as a clue to identify code snippets generated by LLMs, we use regular expressions to remove such prefixes. The regular expressions used to remove prefixes can be found in Figure 9 of Appendix D. For data anonymization, we remove email addresses, URLs, and phone numbers included in code snippets using regular expression. The regular expressions we used for data anonymization can be found in Figure 10 of Appendix D.

3.4 Data Annotation

We collect a total of 85,796 human-written code snippets and based on these, we construct a total of 160,312 LLM-generated code snippets. Based on a total of 246,108 code snippets, we construct DeCo-Binary and DeCo-Multi.

DeCo-Binary Since DeCo-Binary is a benchmark for the binary detection task, we annotate human-written code snippets with a label of 0 and LLM-generated code snippets with a label of 1. Examples of data from the DeCo-Binary benchmark can be found in the Appendix E.

DeCo-Multi We construct the DeCo-Multi dataset by selecting only the code snippets that all four LLMs have successfully modified. Therefore, the code snippets included in DeCo-Multi have a total of five authors, including humans, for the same code snippet. Since DeCo-Multi is a benchmark for multi-class classification tasks, we annotate code snippets generated by humans, ChatGPT, Gemini-Pro, WizardCoder, and DeepSeek-Coder with labels 0, 1, 2, 3, and 4, respectively. Table 1 presents the data statistics of DeCo-Binary and DeCo-Multi. We analyze the similarity between code snippets in the DeCo-Multi dataset using Stanford Moss[9] (Measure of software similarity), a tool for evaluating the similarity of code snippets. In this analysis, we explore both the similarity between human-written code snippets and those generated by LLMs, as well as the similarities among code snippets produced by different LLMs. The analysis results can be found in the Appendix F.

[9] https://theory.stanford.edu/~aiken/moss/

4 Experimental Settings

4.1 Task Definition

We explore the following two tasks based on the DeCo-Binary dataset and the DeCo-Multi dataset.

Task 1: LLM-Generated Code Detection. This taks is a binary detection task aimed at determining whether a given code snippet was written by a human or generated by an LLM.

Task 2: Code Generator Classification. This task is a multi-class classification task aimed at predicting the generator that wrote the given code snippet. To the best of our knowledge, we are the first to explore authorship attribution tasks targeting code domain.

4.2 Binary Detection Methods for Task 1

For the LLM-generated code detection task, we employ 10 black-box zero-shot detection methods. Since white-box detection methods and watermarking methods cannot be applied to commercial LLMs such as ChatGPT and Gemini-Pro[10], we exclude them from the experiments. Black-box zero-shot detection methods are divided into the following three categories.

Metric-based Detection Metric-based detection methods employ pre-trained languages models to analyze text and derive distinctive features from it, such as the rank or entropy of individual words in a text based on preceding context. We use the following six metric-based detection methods: 1) Log-Likelihood (Solaiman et al., 2019); 2) Rank (Gehrmann et al., 2019); 3) Log-Rank (Mitchell et al., 2023); 4) Entropy (Gehrmann et al., 2019); 5) LRR (Su et al., 2023a); 6) Binoculars (Hans et al., 2024).

Perturbation-based Detection Perturbation-based detection methods assess alterations in the log probability of the model when introducing slight changes to the original text. We employ the following three perturbation-based detection methods: 1) DetectGPT (Mitchell et al., 2023); 2) NPR (Su et al., 2023a); 3) DetectGPT4Code (Yang et al., 2023).

Paraphrasing-based Detection Paraphrasing-based detection methods are based on the hypothesis that when both human-written and machine-generated texts are revised through an LLM, the

[10] For these commercial LLMs, we cannot obtain log probabilities or directly intervene in the decoding process.

machine-generated text undergoes less alteration compared to the human-written text. We employ BARTScore-CNN proposed by Zhu et al. (2023). Detailed information about the detection methods we used can be found in the Appendix G.

4.3 Classification Models for Task 2

We present three code generator classifiers by fine-tuning the following models using the DeCo-Multi dataset.

- **OpenAI Detector** (Solaiman et al., 2019): This model is trained using texts generated by GPT-2, capable of determining whether a given text was authored by GPT-2 or not.
- **ChatGPT Detector** (Guo et al., 2023): This model is trained on texts generated by ChatGPT, which are included in the HC3 dataset.
- **CodeBERT** (Feng et al., 2020): CodeBERT is a language model pre-trained on masked language modeling and replaced token detection tasks for six programming languages.

All three models are based on the RoBERTa-base (Liu et al., 2019) architecture. The OpenAI detector and ChatGPT detector are models trained on the task of distinguishing between human-written texts and machine-generated texts, while CodeBERT is a pre-trained model on a code corpus consisting of six programming languages. We initialize the models with their pre-trained checkpoints and fine-tune them on the DeCo-Multi dataset.

4.4 Evaluation Metrics

LLM-Generated Code Detection We use Area Under the Receiver Operating Characteristic curve (**AUROC**) as an evaluation metric for task 1. AUROC represents the area under the curve when plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings. AUROC provides a single scalar value that summarizes the performance of the detector across all possible classification thresholds.

Code Generator Classification For task 2, we employ the following two evaluation metrics: 1) **Macro F1 Score**: is a metric used to evaluate the performance of multi-class classification models. It calculates the harmonic mean of precision and recall for each class and then takes the average across all cases, giving equal weight to each class. 2) **Accuracy**: measures the ratio of correctly predicted samples to the total number of samples in the dataset.

4.5 Implementation Details

LLM-Generated Code Detection We experiment with BARTScore-CNN[11] and Binoculars[12] using their official implementations. The official implementation of BARTScore-CNN uses ChatGPT for paraphrasing, however, to reduce the cost of API usage, we opt for Gemini-Pro[13] instead. Based on the implementation of DetectGPT provided by MGTBench (He et al., 2023), we implement DetectGPT4Code, utilizing CodeT5+ 220M (Wang et al., 2023b) as the perturbation model and PyCodeGPT[14] as the surrogate model for measuring log probability. For the remaining detection methods, we utilize the implementations provided by MGTBench[15]. All the detection methods we use for task 1 are zero-shot detection methods. Therefore, we conduct evaluations on all samples in the DeCo-Binary dataset without training process. For the experiments of task 1, we use the NVIDIA RTX A6000 with 48GB of memory.

Code Generator Classification We divide the DeCo-Multi dataset into proportions of 8:1:1 to obtain the training, validation, and testing subsets. The model is trained using the training subset, and the model checkpoint from the epoch with the best performance on the validation subset is saved. We assess the model performance on the testing subset using this model checkpoint. We perform hyper-parameter search based on validation accuracy within the ranges of learning rates {1e-5, 2e-5, 5e-5}, and batch sizes {8, 16, 32}. We train the model using AdamW (Loshchilov and Hutter, 2019) as the optimizer. We conduct fine-tuning over 10 epochs. For the experiments of task 2, we use the NVIDIA RTX 3090 with 24GB of memory.

5 Experimental Results

5.1 LLM-Generated Code Detection

Table 2 shows the performance of LLM-generated code detection for 10 zero-shot binary detection methods. We report the performance of detection methods for each programming language and their overall performance across all programming lan-

[11] https://github.com/thunlp/LLM-generated-text-detection
[12] https://github.com/ahans30/Binoculars
[13] Gemini-Pro currently does not charge for API usage.
[14] https://huggingface.co/Daoguang/PyCodeGPT
[15] https://github.com/xinleihe/MGTBench

Zero-Shot Binary Detector	C	C++	Java	Python	Overall
Log-Likelihood (Solaiman et al., 2019)	54.14	56.54	56.29	55.55	55.63
Rank (Gehrmann et al., 2019)	57.06	56.46	55.99	54.10	55.90
Log-Rank (Mitchell et al., 2023)	54.91	56.06	55.99	55.09	55.51
Entropy (Gehrmann et al., 2019)	46.69	49.69	48.32	43.62	47.08
LRR (Su et al., 2023a)	46.52	53.34	51.98	49.66	50.37
Binoculars (Hans et al., 2024)	46.36	48.13	51.19	46.92	48.15
DetectGPT (Mitchell et al., 2023)	60.48	57.60	55.80	57.37	57.81
NPR (Su et al., 2023a)	62.01	57.73	55.70	57.63	**58.26**
DetectGPT4Code (Yang et al., 2023)	61.04	57.97	54.46	57.30	57.69
BARTScore-CNN (Zhu et al., 2023)	57.15	59.41	59.05	56.24	57.96

Table 2: **Task1) LLM-Generated Code Detection Performance.** We use AUROC as an evaluation metric. Detectors divided by horizontal lines are, from top to bottom, metric-based detectors, perturbation-based detectors, and a paraphrasing-based detector. The numbers in the rightmost column represent the average performance across four programming languages. We emphasize the highest overall performance in bold.

Classifier	C		C++		Java		Python		Overall	
	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC
OpenAI Detector	36.53	37.18	34.35	36.40	35.94	38.10	35.38	39.06	$35.55_{\pm 0.92}$	$37.69_{\pm 0.55}$
ChatGPT Detector	26.91	32.08	31.98	35.17	36.64	37.76	28.45	34.57	$30.99_{\pm 1.45}$	$34.89_{\pm 0.82}$
CodeBERT	42.51	43.23	39.04	41.63	36.83	37.94	34.08	36.86	$\mathbf{38.11}_{\pm 1.18}$	$\mathbf{39.91}_{\pm 0.92}$

Table 3: **Task2) Code Generator Classification Performance.** F1 and ACC represent macro F1 score and accuracy respectively. We report the average performance of experiments conducted with five different seeds. The numbers in the rightmost column represent the average performance across four programming languages. We emphasize the highest overall performance in bold.

guages. The detection methods in the Table 2 are divided into three groups based on horizontal lines, from top to bottom: metric-based detection methods, perturbation-based detection methods, and a paraphrasing-based detection method. Some detection methods show performance worse than random predictions (typically an AUROC of 50), and even the best-performing method only shows slightly better performance than random predictions. This highlights the difficulty of the task of detecting LLM-generated code snippets.

The perturbation-based detection methods and a paraphrasing-based detection method generally show superior performance compared to metric-based detection methods. Interestingly, despite using a code-specialized language model, DetectGPT4Code shows lower performance compared to DetectGPT, which uses a language model specialized for natural language texts. This suggests that simply replacing a language model specialized for natural language with a code-specialized one is not highly effective for detecting LLM-generated code snippets. BARTScore-CNN, which is a state-of-the-art (SOTA) methodology for detecting texts generated by LLMs, also shows unsatisfactory performance in detecting LLM-generated code snippets. Based on these experimental results, we emphasize the need for further research into methodologies for detecting LLM-generated code snippets.

Finding. SOTA methods for detecting texts generated by LLMs struggle to detect LLM-generated code snippets.

5.2 Code Generator Classification

Table 3 shows the performance of three classifiers on the task of classifying the generator of a code snippet. We report the average performance of experiments conducted with five different seeds. For overall results across four languages, we report both the average performance and the standard deviation. The standard deviations for individual programming languages can be found in the Appendix H (Table 6). Overall, CodeBERT, which is pre-trained on programming languages, shows superior performance compared to models pre-trained on the task of distinguishing between machine-generated and human-written texts. This shows that for the task of identifying the generator of a code snippet, it is beneficial for the model to first acquire knowledge about programming languages.

The confusion matrices for each model's predictions can be found in the Appendix I. Additionally, the embedding visualization analysis for the DeCo-Multi dataset can be found in the Appendix J. Through embedding visualization analysis, we observe distinct differences between code snippets written by humans and those generated by LLMs. However, there are no clear differences observed among code snippets generated by different LLMs. This highlights the challenge of classifying the generator of code snippets. If a specific LLM generates a significant number of vulnerable code snippets or produces copyrighted code snippets, it becomes even more crucial to identify code snippets generated by such an LLM. Therefore, we emphasize the need for research into classification methodologies for this task.

Finding. For the task of classifying the generator of code snippets, it is effective for models to be pre-trained on programming languages.

6 Analysis

6.1 Linguistic Analysis

We analyze the linguistic features of human-written code snippets and LLM-generated code snippets from the perspectives of density and average log-perplexity.

Density Density refers to how frequently various tokens appear within code snippets. Density is calculated as follows:

$$\text{Density} = 100 \times V/(L \times N), \quad (1)$$

where V represents the number of unique tokens used in all code snippets, L is the average number of tokens present in each code snippet, and N denotes the number of code snippets. A high density value indicates that a more diverse range of tokens have been used in code snippets.

Log-Perplexity Log-perplexity is a measure of how well a probability distribution predicts a sample. In the context of language models, it quantifies how confused the model is when predicting the next token. Log-perplexity is the average negative log-likelihood of all tokens in the given sequence. Log-perplexity is calculated as follows:

$$\text{Log-PPL}(T) = -\frac{1}{M}\sum_{i=1}^{M} \log_2 P(t_i|t_{i-1}, ..., t_1), \quad (2)$$

where M denotes the total number of tokens in the sequence and $P(t_i|t_{i-1}, ..., t_1)$ represents the probability of token t_i given the preceding tokens in the sequence. A lower log-perplexity indicates that the model is more confident in its predictions, whereas a higher log-perplexity suggests that the model is more uncertain. For calculating log-perplexity, we use CodeLlama 7B (Roziere et al., 2023). We calculate the log-perplexity for each code snippet, and then compute the average value.

Table 4 shows the results of the linguistic analysis for the DeCo-Multi dataset. In the case of density, code snippets written by humans show the lowest values for all four programming languages. This indicates that LLMs use a more diverse range of tokens than humans when writing code snippets. One possible reason for this is that humans tend to follow coding conventions[16] when writing code snippets. That is, when naming variables and functions, humans tend to adhere to coding conventions, which may result in more standardized naming rules than those followed by LLMs. The results of the linguistic analysis, including the average length (L) and vocabulary size (V), can be found in the Appendix K (Table 7).

For the three programming languages other than Python, code snippets written by humans show the lowest average log-perplexity values. This indicates that CodeLlama is more familiar with code snippets written by humans than those written by LLMs. This phenomenon can be attributed to the composition of the dataset utilized for the pre-training of CodeLlama, which potentially includes a greater volume of code snippets authored by humans compared to code snippets generated by LLMs. CodeLlama was released in August 2023, while ChatGPT was released in November 2022, WizardCoder in June 2023, Gemini-Pro in November 2023, and DeepSeek-Coder in January 2024. Therefore, it is unlikely that the data used for pre-training CodeLlama included code snippets generated by the four LLMs we utilized, given their respective release dates.

When looking at overall results for four programming languages, code snippets written by humans show the lowest density value and the lowest average log-perplexity value. Among the LLMs, code snippets written by Gemini-Pro exhibit the lowest density value and the lowest average log-perplexity value. Table 8 in the Appendix K presents the re-

[16] https://peps.python.org/pep-0008/

		Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Density	C	**18.70**	22.98	22.59	22.61	24.19
Avg. Log-Perplexity		**0.49**	0.51	0.50	0.54	0.55
Density	C++	**23.37**	27.04	26.64	27.18	28.17
Avg. Log-Perplexity		**0.48**	0.55	0.51	0.53	0.54
Density	Java	**19.02**	23.01	22.69	22.78	24.49
Avg. Log-Perplexity		**0.42**	0.45	0.44	0.47	0.47
Density	Python	**21.84**	27.01	26.00	25.70	28.15
Avg. Log-Perplexity		0.66	0.64	**0.63**	0.68	0.65
Density	Overall	**20.73**	25.01	24.48	24.56	26.25
Avg. Log-Perplexity		**0.51**	0.54	0.52	0.56	0.55

Table 4: Linguistic analysis on the DeCo-Multi dataset. In each row, the smallest number is highlighted in bold. We can see that, overall, code snippets written by humans have lower density values and smaller average log-perplexity values compared to those written by LLMs.

sults of linguistic analysis conducted after removing natural language comments from the DeCo-Multi dataset. We confirm that similar results are obtained even after the removal of comments.

Finding. There are distinct linguistic features between human-written code snippets and LLM-generated code snippets.

6.2 Impact of Natural Language Comments

We investigate the impact of natural language comments in the task of classifying the generator of code snippets. We conduct analysis using CodeBERT, which showed the highest performance in this task (analysis using the OpenAI Detector and ChatGPT Detector can be found in Table 9 of Appendix L). We remove natural language comments from the DeCo-Multi dataset (both training and evaluation data) and measure the performance of models. We report the average performance of experiments conducted with five different seeds. Table 5 compares the performance of the model on code snippets with natural language comments and code snippets without natural language comments. Through the experimental results, we observe that removing natural language comments from code snippets leads to an average decrease of 10.62%p in macro F1 score and 9.72%p in accuracy across four programming languages. This indicates that natural language comments are an important feature in the predictions of models for this task. The confusion matrices for the predictions of each model can be found in Figure 23 of Appendix L.

Finding. In the task of identifying the authors of code snippets, natural language comments are an important feature.

	NL	Macro F1	Accuracy
C	O	42.51	43.23
	X	27.65 (↓ 14.86%p)	30.03 (↓ 13.20%p)
C++	O	39.04	41.63
	X	30.35 (↓ 8.69%p)	31.58 (↓ 10.05%p)
Java	O	36.83	37.94
	X	27.63 (↓ 9.20%p)	32.15 (↓ 5.79%p)
Python	O	34.08	36.86
	X	24.35 (↓ 9.73%p)	27.00 (↓ 9.86%p)
Overall	O	**38.11**	**39.91**
	X	**27.49 (↓ 10.62%p)**	**30.19 (↓ 9.72%p)**

Table 5: We analyze the impact of natural language comments in the task of identifying the authors of code snippets. We experiment using CodeBERT, which exhibits the best performance.

7 Conclusion

We have addressed critical challenges associated with the deployment of LLMs in code generation by introducing the DeCo benchmark—a pioneering effort for detecting code snippets generated by LLMs. The DeCo benchmark, incorporating 246K code snippets across four programming languages and generated by both commercial and open-source LLMs, provides a foundation for this research. We propose two tasks based on the DeCo dataset: 1) Detection of LLM-generated code snippets; and 2) Classification of the authors who wrote the code snippets. We evaluate 13 detection methods, including SOTA methods, on the DeCo dataset. We find that SOTA methods for detecting LLM-generated texts struggle with detecting LLM-generated code snippets. Furthermore, we analyze the linguistic features of the DeCo dataset from the perspectives of density and average log-perplexity. We discover distinct differences between code snippets written by humans and those generated by LLMs. We hope our findings paves the way for more secure and trustworthy use of AI in coding applications.

Limitations

We propose the first benchmark for detecting LLM-generated code snippets, and perform various analyses, including the evaluation of different detection methods and linguistic analysis. However, there are some limitations to our research.

Low-resource Programming Languages We cover four major programming languages: C, C++, Java, and Python. However, we do not include low-resource programming languages such as Julia, Rust, and Lua in the DeCo dataset. Detecting LLM-generated code snippets in low-resource programming languages is also important, and we leave this as future work.

Various Sizes of LLMs We construct the DeCo dataset using commercial LLMs such as ChatGPT and Gemini-Pro, whose exact model sizes are unknown, along with open-source LLMs like WizardCoder and DeepSeek-Coder with a parameter size of 33B. However, recently, LLMs of various sizes (ranging from 2B to 70B) have been released. The code generation capabilities are expected to differ depending on the model size of LLMs, and thus, the characteristics of the generated code snippets are also expected to vary. Analyzing the code snippets generated by LLMs of different sizes is an important avenue for future research.

Detection Methods We experiment with various detection methods on the DeCo dataset, and discover that even SOTA methods struggle to detect LLM-generated code snippets. We confirm that even DetectGPT4Code, a method specifically designed for detecting LLM-generated code snippets, does not achieve satisfactory performance. We emphasize the necessity for active research on methodologies to detect LLM-generated code snippets.

Ethical Considerations

We discuss some ethical concerns and broader impacts of our research.

Licenses The code snippets comprising the DeCo dataset adhere to one of the licenses: Apache, BSD, or MIT.

Personal Information We anonymize the data by removing email addresses, URLs, and phone numbers included in the comments of the code snippets comprising the DeCo dataset.

Broader Impact Due to potential security vulnerabilities, copyright issues, and concerns about academic cheating in code snippets generated by LLMs, there is a need to develop systems to detect these code snippets. The DeCo dataset can be utilized to develop models for this purpose.

References

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. 2023. The Falcon Series of Open Language Models. *arXiv preprint arXiv:2311.16867*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP*.

Sebastian Gehrmann, Hendrik Strobelt, and Alexander Rush. 2019. GLTR: Statistical Detection and Visualization of Generated Text. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

Biyang Guo, Xin Zhang, Ziyuan Wang, Minqi Jiang, Jinran Nie, Yuxuan Ding, Jianwei Yue, and Yupeng Wu. 2023. How Close is ChatGPT to Human Experts? Comparison Corpus, Evaluation, and Detection. *arXiv preprint arXiv:2301.07597*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Abhimanyu Hans, Avi Schwarzschild, Valeriia Cherepanova, Hamid Kazemi, Aniruddha Saha, Micah Goldblum, Jonas Geiping, and Tom Goldstein. 2024. Spotting LLMs With Binoculars: Zero-Shot Detection of Machine-Generated Text. *arXiv preprint arXiv:2401.12070*.

Xinlei He, Xinyue Shen, Zeyuan Chen, Michael Backes, and Yang Zhang. 2023. MGTBench: Benchmarking Machine-Generated Text Detection. *arXiv preprint arXiv:2303.14822*.

John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. 2023. A Watermark for Large Language Models. In *International Conference on Machine Learning*.

Rohith Kuditipudi, John Thickstun, Tatsunori Hashimoto, and Percy Liang. 2023. Robust Distortion-free Watermarks for Language Models. *arXiv preprint arXiv:2307.15593*.

Taehyun Lee, Seokhee Hong, Jaewoo Ahn, Ilgee Hong, Hwaran Lee, Sangdoo Yun, Jamin Shin, and Gunhee Kim. 2023. Who Wrote this Code? Watermarking for Code Generation. *arXiv preprint arXiv:2305.15060*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint*.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *Proceedings of the 9th International Conference on Learning Representations*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568*.

Niloofar Mireshghallah, Justus Mattern, Sicun Gao, Reza Shokri, and Taylor Berg-Kirkpatrick. 2024. Smaller Language Models are Better Zero-shot Machine-Generated Text Detectors. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 2: Short Papers)*.

Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. DetectGPT: Zero-shot machine-generated text detection using probability curvature. In *International Conference on Machine Learning*.

Phuong T Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2023. Is this Snippet Written by Chatgpt? an Empirical Study with a CodeBERT-based Classifier. *arXiv preprint arXiv:2307.09381*.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of Github Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*.

Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. 2019. Release Strategies and the Social Impacts of Language Models. *arXiv preprint arXiv:1908.09203*.

Jinyan Su, Terry Zhuo, Di Wang, and Preslav Nakov. 2023a. DetectLLM: Leveraging Log Rank Information for Zero-Shot Detection of Machine-Generated Text. In *Findings of the Association for Computational Linguistics: EMNLP 2023*.

Zhenpeng Su, Xing Wu, Wei Zhou, Guangyuan Ma, and Songlin Hu. 2023b. HC3 Plus: A Semantic-Invariant Human ChatGPT Comparison Corpus. *arXiv preprint arXiv:2309.02731*.

Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*.

Jian Wang, Shangqing Liu, Xiaofei Xie, and Yi Li. 2023a. Evaluating AIGC Detectors on Code Content. *arXiv preprint arXiv:2304.05193*.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023b. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

Yuxia Wang, Jonibek Mansurov, Petar Ivanov, Jinyan Su, Artem Shelmanov, Akim Tsvigun, Chenxi Whitehouse, Osama Mohammed Afzal, Tarek Mahmoud, Toru Sasaki, Thomas Arnold, Alham Aji, Nizar Habash, Iryna Gurevych, and Preslav Nakov. 2024. M4: Multi-generator, Multi-domain, and Multilingual Black-Box Machine-Generated Text Detection. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Xianjun Yang, Kexun Zhang, Haifeng Chen, Linda Petzold, William Yang Wang, and Wei Cheng. 2023. Zero-shot Detection of Machine-Generated Codes. *arXiv preprint arXiv:2310.05103*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large Language Models Meet NL2Code: A Survey. *arXiv preprint arXiv:2212.09420*.

Biru Zhu, Lifan Yuan, Ganqu Cui, Yangyi Chen, Chong Fu, Bingxiang He, Yangdong Deng, Zhiyuan Liu, Maosong Sun, and Ming Gu. 2023. Beat LLMs at Their Own Game: Zero-Shot LLM-Generated Text Detection via Querying ChatGPT. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*.

A Prompt for Code Generation via LLMs

Figure 1 represents the prompt used for code generation with LLMs. We provide human-written code snippets (**[CODE]**) along with their programming languages (**[LANG]**), and instruct LLMs to modify the code snippets while maintaining their functionality and programming language.

````
I will give you a code snippet written in [LANG].
Please modify this code snippet while ensuring that the functionality
of the given code snippet is preserved.
Please make some changes to the code to avoid it being too similar to
the original.
The revised code snippet must be written in the same programming
language as the original.
Do not include any explanations about how you modified the code
snippet or add any arbitrary comments to the code snippet.
After modifying, prepend a prefix at the beginning of the revised
code snippet indicating its programming language.
For example, if the revised code snippet is written in [LANG], add
```[LANG]``` at the start.
The following code snippet is the one you need to modify.
-----------------------------------------------------------------
[CODE]
````

Figure 1: Prompt for code generation using LLMs. We utilize LLMs to generate code snippets by incorporating human-written code snippets (**[CODE]**) along with their respective programming languages (**[LANG]**) into this prompt template.

B Characteristics of LLMs during Code Generation

We describe several characteristics exhibited by LLMs during the code generation process:

- LLMs tend to confuse the C and C++ languages. We provide LLMs with human-written code snippets as input and instruct them to maintain the programming language of the given code snippet. However, LLMs sometimes confuse between the C and C++ languages. For example, even when the provided human-written code snippet is explicitly stated to be written in C, LLMs may generate a code snippet written in C++ instead.
- LLMs tend to generate short-length code snippets. As seen in Tables 7 and 8, code snippets generated by LLMs are generally shorter than those written by humans.
- LLMs tend to remove documentation located at the top of code snippets when modifying human-written code snippets.
- Among LLMs, ChatGPT tends to primarily modify variable names and function names when modifying human-written code snippets compared to other LLMs.

Figures 2 and 3 respectively illustrate examples of a code snippet written by a human and a code snippet written by ChatGPT.
````

```c
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <stdio.h>
#include <stdlib.h>     /* For atoi */
#include "gd.h"

/* A short program which converts a .png file into a .gd file, for
   your convenience in creating images on the fly from a
   basis image that must be loaded quickly. The .gd format
   is not intended to be a general-purpose format. */

int
main (int argc, char **argv)
{
    gdImagePtr im;
    FILE *in, *out;
    int x, y, w, h;

    if (argc != 7) {
        fprintf (stderr,
                 "Usage: gdparttopng filename.gd filename.png x y w h\n");
        exit (1);
    }
    in = fopen (argv[1], "rb");
    if (!in) {
        fprintf(stderr, "Input file does not exist!\n");
        exit (1);
    }

    x = atoi (argv[3]);
    y = atoi (argv[4]);
    w = atoi (argv[5]);
    h = atoi (argv[6]);
...
```

Figure 2: Example from the DeCo-Binary dataset. Human-written C code snippet.

```c
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <stdio.h>
#include <stdlib.h>     /* For atoi */
#include "gd.h"

int main (int argc, char **argv)
{
    gdImagePtr img;
    FILE *input, *output;
    int startX, startY, width, height;

    if (argc != 7) {
        fprintf (stderr,
                "Usage: convertToPng input.gd output.png startX startY width height\n");
        exit (1);
    }
    input = fopen (argv[1], "rb");
    if (!input) {
        fprintf(stderr, "Error: Input file is missing!\n");
        exit (1);
    }

    startX = atoi (argv[3]);
    startY = atoi (argv[4]);
    width = atoi (argv[5]);
    height = atoi (argv[6]);
...
```

Figure 3: Example from the DeCo-Binary dataset. ChatGPT-generated C code snippet.

C Types of Errors that Occur During Code Generation Using LLMs

We report on some types of errors that occur when generating code snippets using LLMs. We report the types of errors in descending order of frequency. Examples of each error type can be found in Figures 4, 5, 6, 7, and 8.

Case 1) Missing Code Snippets: This refers to cases where LLMs output only natural language descriptions without code snippets. LLMs produce summaries describing the functionality of human-written code snippets.

Case 2) Code Snippets Written in Different Programming Languages: This occurs when LLMs output code snippets written in a different programming language from human-written code snippets. Such cases are most common when human-written code snippets are written in C or C++.

Case 3) Miss Use of Quotation Marks: LLMs often omit closing quotation marks when writing comments. This can result in error code snippets. Such occurrences are more common when using open-source LLMs for code generation.

Case 4) Miss Use of Parentheses: LLMs often miss closing parentheses. This occurs more frequently when using open-source LLMs as well.

Case 5) Repetitive Output of Meaningless Characters: Occasionally, LLMs may output repetitive meaningless characters when generating code snippets. These cases are very rare, and commercial LLMs rarely exhibit this behavior.

```
In the given C code snippet, it is unclear what modifications are needed. However, if
you want to add error checking or some additional functionality, it would be helpful to
know specifically what these changes would be.
```

Figure 4: Error types occurring during code snippet generation by LLMs. Case 1) Missing Code Snippets.

````
C# code:
```csharp
using System;
using System.Collections.Generic;
using OneLayeredBucket; // This assumes you have a reference to the `OneLayeredBucket`
library or namespace. You need to replace this with your actual namespace if
different.
...
````

Figure 5: Error types occurring during code snippet generation by LLMs. Case 2) Code Snippets Written in Different Programming Languages.

```
...
class StackExchangeOAuth2Adapter(OAuth2Adapter):
    provider_id = StackExchangeProvider.id
    access_token_url = '[URL]
    authorize_url = '[URL]
    profile_url = '[URL]
...
```

Figure 6: Error types occurring during code snippet generation by LLMs. Case 3) Miss Use of Quotation Marks.
```

```
...
int cudaMemcpyDtoH(void *dst, const void *src, size_t count){
    cudaError_t err = cudaMemcpy(dst, src, count, cudaMemcpyDeviceToHost);
    return (err == cudaSuccess) ? 0 : -1;
}
end:
    free(in);
    free(out);
    return -1;
}
```

Figure 7: Error types occurring during code snippet generation by LLMs. Case 4) Miss Use of Parentheses.

```
GGGGGGGGGGGGGGGGCCCCCXXXXPPPPTTTTSSSSSOOOOIIIIIINNNN
class Solution {
    public:
        boolean isInterleave(String s1, String s2, String s3) {
            if(s1.length() + s2.length() != s3.length()) return false;
            if(s1.length() < s2.length()) return isInterleave(s2, s1, s3);
            boolean[] f = new boolean[s2.length() + 1];
            Arrays.fill(f, true);
            for(int j = 1; j <= s2.length(); ++j) {
                f[j] = f[j - 1] && s2.charAt(jGGGGGGG - 1) == s3.charAt(j - 1);
            }
            for(intGGGGGGGGGGGGGGGGGGGGGGGG i = 1; i <= s1.length(); ++i) {
                f[0] = f[0] && s1.charAt(i - 1) == s3.charAt(i + j - 1);
                for(int j = 1; j <= s2.length(); ++j) {
                    f[j] = (f[j] && s1.charAt(i - 1) == s3.charAt(i + j - 1))
                        || (f[j - 1] && s2.charAt(j - 1) == s3.charAt(i + j - 1));
                }
            }
            return f[s2.length()];
        }
};
```

Figure 8: Error types occurring during code snippet generation by LLMs. Case 5) Repetitive Output of Meaningless Characters.

D Data Filtering and Cleaning

Regular Expressions for Removing the PL Prefix

Figure 9 displays the regular expressions used to remove the prefixes indicating the programming languages of code snippets. LLMs often prepend a prefix indicating the programming language of the code snippet, such as cpp at the beginning of the code snippet. Such prefixes could introduce bias into the detection model, leading it to classify code snippets with the prefix as being authored by the LLM. Thus, we remove such prefixes using regular expressions.

```
regex = {
"c": [r'c\\n|C\n', r'c\\n(.*?)|C\\n(.*?)', r'c|C', r'c\n|C\n'],
"cpp": [r'cpp\\n|CPP\n|Cpp\\n|c\+\+\n|C\\+\\+\\n',
r'cpp\n(.?)|CPP\n(.?)|Cpp\n(.?)|c\+\+\n(.?)|C\+\+\n(.?)',
r'cpp|CPP|Cpp|c\\+\\+|C\\+\\+', r'cpp\\n|CPP\\n|Cpp\\n|c\\+\\+\\n|C\\+\\+\\n'],
"java": [r'java\n|JAVA\\n|Java\n', r'java\\n(.*?)|JAVA\\n(.*?)|Java\\n(.*?)',
r'java|JAVA|Java', r'java\n|JAVA\n|Java\n'],
"py": [r'python\\n|Python\n|PYTHON\\n', r'python\n(.?)|Python\n(.?)|PYTHON\n(.?)',
r'python|Python|PYTHON', r'python\\n|Python\\n|PYTHON\\n']
}
```

Figure 9: Regular expressions for removing the prefixes indicating the programming languages of code snippets.

Regular Expressions for Data Anonymization

Figure 10 displays the regular expressions for data anonymization. We use regular expressions to remove URLs, phone numbers, and email addresses included in the comments of code snippets.

```
import re
re.sub(r'http(s?):\/\/[^\r\n\t\f\v )\]\}]+', '[URL]')
re.sub(r'www\.\S+', '[URL]')
re.sub(r'^[\+]?[(]?[0-9]{3}[)]?[-\s\.]?[0-9]{3}[-\s\.]?[0-9]{4,6}$', '[PHONE NUMBER]')
re.sub(r'(?i)\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b', '[EMAIL]')
```

Figure 10: Regular expressions used for data anonymization by removing URL addresses, phone numbers, and email addresses.

E Examples of DeCo-Binary Benchmark

Figures 11 to 18 present examples of data from the DeCo-Binary benchmark. Sections marked with a red box represent areas removed by the LLM during the modification of the human-written code snippet. Those marked with a pink box indicate additions made by the LLM. Meanwhile, the areas highlighted with a yellow box show where the LLM has modified the human-written code snippet.

```
#ifndef TAPENADE
#include <math.h>
#endif
#define Max(x,y) fmax(x,y)
#define Min(x,y) fmin(x,y)
#define Heaviside(x) ((x>=0)?1.0:0.0)

#define u(x,xx,xxx) u[x][xx][xxx]
#define c(x,xx,xxx) c[x][xx][xxx]
#define u_1(x,xx,xxx) u_1[x][xx][xxx]
#define u_2(x,xx,xxx) u_2[x][xx][xxx]
void wave3d(double* u_vec, double* c_vec, double* u_1_vec, double* u_2_vec, double D,
    int n) {
  double (*u)[n][n] = (double (*)[n][n]) u_vec;
double (*c)[n][n] = (double (*)[n][n]) c_vec;
double (*u_1)[n][n] = (double (*)[n][n]) u_1_vec;
double (*u_2)[n][n] = (double (*)[n][n]) u_2_vec;

    int i;
    int j;
    int k;
    #pragma omp parallel for private(k,j,i)
    for ( i=1; i<=n - 2; i++ ) {
        for ( j=1; j<=n - 2; j++ ) {
            for ( k=1; k<=n - 2; k++ ) {
                u(i,j,k) += D*(-6*u_1(i, j, k) + u_1(i, j, k - 1) + u_1(i, j, k + 1) +
                u_1(i, j - 1, k) + u_1(i, j + 1, k) + u_1(i - 1, j, k) +
                u_1(i + 1, j, k))*c(i, j, k) + 2.0*u_1(i, j, k) - u_2(i, j, k);
            }
        }
    }
}
```

Figure 11: Example from the DeCo-Binary dataset. C code snippet written by human.

```
#include <math.h>
#define Max(x,y) fmax((x),(y))
#define Min(x,y) fmin((x),(y))
#define Heaviside(x) ((x)>=0?1.0:0.0)

typedef double (*array_3d)[n][n];

void wave3d(double* u_vec, double* c_vec, double* u_1_vec, double* u_2_vec, double D,
    int n) {
    array_3d u = (array_3d) u_vec;
    array_3d c = (array_3d) c_vec;
    array_3d u_1 = (array_3d) u_1_vec;
    array_3d u_2 = (array_3d) u_2_vec;

    int i, j, k;

    #pragma omp parallel for private(k,j,i)
    for ( i=1; i<=n - 2; i++ ) {
        for ( j=1; j<=n - 2; j++ ) {
            for ( k=1; k<=n - 2; k++ ) {
                u[i][j][k] += D*(-6*u_1[i][j][k] + u_1[i][j][k - 1] + u_1[i][j][k + 1]
                + u_1[i][j - 1][k] + u_1[i][j + 1][k] + u_1[i - 1][j][k]
                + u_1[i + 1][j][k])*c[i][j][k] + 2.0*u_1[i][j][k] - u_2[i][j][k];
            }
        }
    }
}
```

Figure 12: Example from the DeCo-Binary dataset. C code snippet generated by DeepSeek-Coder.

```cpp
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>
#include <math.h>

int main(int argc, const char* argv[]){
  cv::Mat img = cv::imread("imori.jpg", cv::IMREAD_COLOR);

  int width = img.rows;
  int height = img.cols;

  cv::Mat out = cv::Mat::zeros(height, width, CV_8UC3);

  for (int j = 0; j < height; j++){
    for (int i = 0; i < width; i++){
      for (int c = 0; c < 3; c++){
    out.at<cv::Vec3b>(j,i)[c] = (uchar)(floor((double)img.at<cv::Vec3b>(j,i)[c] / 64)
                                * 64 + 32);
      }
    }
  }

  //cv::imwrite("out.jpg", out);
  cv::imshow("answer", out);
  cv::waitKey(0);
  cv::destroyAllWindows();

  return 0;
}
```

Figure 13: Example from the DeCo-Binary dataset. C++ code snippet written by human.

```cpp
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>
#include <math.h>

int main(int argc, const char* argv[]){
  cv::Mat inputImage = cv::imread("input.jpg", cv::IMREAD_COLOR);

  int imgWidth = inputImage.rows;
  int imgHeight = inputImage.cols;

  cv::Mat outputImage = cv::Mat::zeros(imgHeight, imgWidth, CV_8UC3);

  for (int y = 0; y < imgHeight; y++){
    for (int x = 0; x < imgWidth; x++){
      for (int c = 0; c < 3; c++){
        outputImage.at<cv::Vec3b>(y,x)[c] =
                        (uchar)(floor((double)inputImage.at<cv::Vec3b>(y,x)[c] / 64) *
                                64 + 32);
      }
    }
  }

  //cv::imwrite("output.jpg", outputImage);
  cv::imshow("result", outputImage);
  cv::waitKey(0);
  cv::destroyAllWindows();

  return 0;
}
```

Figure 14: Example from the DeCo-Binary dataset. C++ code snippet generated by DeepSeek-Coder.

```java
package dzhuang.pedyield.tracker;

public class Position {
    // objects position
    public double x; // x of the middle point
    public double y; // y of the middle point
    public double w;
    public double h;

    public Position(double x, double y, double w, double h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }

    public Position(Position position) {
        this.x = position.x;
        this.y = position.y;
        this.w = position.w;
        this.h = position.h;
    }
}
```

Figure 15: Example from the DeCo-Binary dataset. Java code snippet generated by human.

```java
import java.awt.geom.Rectangle2D;

public class Position extends Rectangle2D.Double {
    public Position(double x, double y, double w, double h) {
        super(x, y, w, h);
    }

    public Position(Position position) {
        super(position.x, position.y, position.w, position.h);
    }
}
```

Figure 16: Example from the DeCo-Binary dataset. Java code snippet generated by Gemini-Pro.

```python
"""
An sorted array of integers was rotated an unknown number of times.

Given such an array, find the index of the element in the array in faster than linear
time.
If the element doesn't exist in the array, return null.
For example, given the array [13, 18, 25, 2, 8, 10] and the element 8,
return 4 (the index of 8 in the array).
You can assume all the integers in the array are unique.
"""
from typing import Optional


def find_index(arr: list, k: int) -> Optional[int]:
    start, end = 0, len(arr) - 1

    while start <= end:
        mid: int = (start + end) >> 1

        if arr[mid] == k:
            return mid
        elif arr[mid] > k:
            if k < arr[0] and arr[mid] > arr[0]:
                start = mid + 1
            else:
                end = mid - 1
        else:
            if k > arr[0] and arr[mid] < arr[0]:
                end = mid - 1
            else:
                start = mid + 1
    return None


if __name__ == "__main__":
    arr: list = [13, 18, 25, 2, 8, 10]

    assert find_index(arr, 13) == 0
    assert find_index(arr, 18) == 1
    assert find_index(arr, 25) == 2
    assert find_index(arr, 2) == 3
    assert find_index(arr, 8) == 4
    assert find_index(arr, 10) == 5
    assert find_index(arr, 1) is None
```

Figure 17: Example from the DeCo-Binary dataset. Python code snippet written by human.

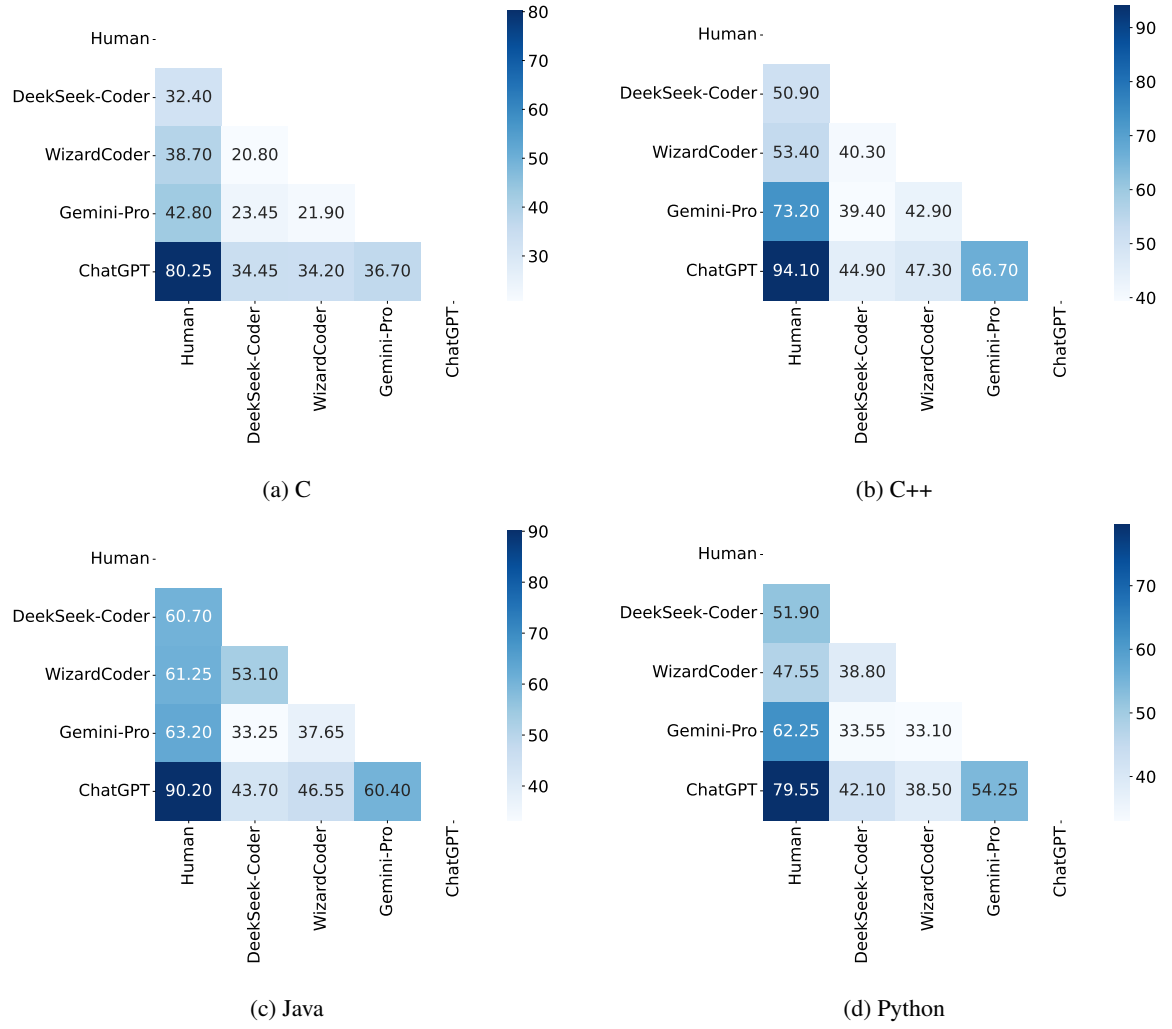
```python
from typing import Optional

def find_index(arr: list, k: int) -> Optional[int]:
    start, end = 0, len(arr) - 1

    while start <= end:
        mid = (start + end) >> 1

        if arr[mid] == k:
            return mid
        elif arr[mid] < arr[0] != (k < arr[0]):
            if arr[mid] < k:
                start = mid + 1
            else:
                end = mid - 1
        else:
            if arr[mid] < k:
                start = mid + 1
            else:
                end = mid - 1
    return None

if __name__ == "__main__":
    arr = [13, 18, 25, 2, 8, 10]

    assert find_index(arr, 13) == 0
    assert find_index(arr, 18) == 1
    assert find_index(arr, 25) == 2
    assert find_index(arr, 2) == 3
    assert find_index(arr, 8) == 4
    assert find_index(arr, 10) == 5
    assert find_index(arr, 1) is None
```

Figure 18: Example from the DeCo-Binary dataset. Python code snippet generated by WizardCoder.

F Similarity Between Code Snippets Written by Various Generators

We analyze the similarity between code snippets written by various authors using Stanford Moss, a tool used for measuring software similarity, which is also utilized in source code plagiarism detection. We randomly sample 400 code snippets from the DeCo-Multi dataset (100 code snippets per programming language) and utilize them for analysis. Figure 19 presents the results of similarity analysis using Moss. We can observe that code snippets generated by commercial LLMs (ChatGPT and Gemini-Pro) are more similar to code snippets written by humans compared to those generated by open-source LLMs (WizardCoder and DeepSeek-Coder). Due to the user-friendly interfaces such as web UIs and APIs, many users tend to use commercial LLMs more than open-source LLMs. Therefore, the analysis results showing that commercial LLMs generate code snippets more similar to human-written ones than open-source LLMs further underscore the importance of the task of detecting LLM-generated code snippets. When comparing between LLMs, we can observe that the similarity between code snippets generated by ChatGPT and Gemini-Pro is the highest.

Finding. Code snippets generated by commercial LLMs are more similar to human-written code snippets than those generated by open-source LLMs.



(a) C (b) C++

(c) Java (d) Python

Figure 19: The similarity between code snippets written by various authors, measured using Stanford Moss, a tool for measuring software similarity. The analysis results show that ChatGPT writes code snippets most similarly to humans, followed by Gemini-Pro. Both of these LLMs are commercial LLMs.

G Details on Binary Detection Methods

The details on binary detection methods for task 1 are as follows.

Metric-based Detection

- **Log-Likelihood**: This approach uses a language model to calculate the log probability of each word in a given text and then takes the average. If the average log-probability value is high, the given text is considered to be LLM-generated.
- **Rank**: By evaluating the absolute rank of each word within a text based on its preceding context, we can calculate a score for the text by averaging these rank values. A lower score suggests the text is more likely to have been generated by LLMs.
- **Log-Rank**: This approach applies logarithmic function to the rank value of each word in the given text.
- **Entropy**: This approach takes the average entropy value of each word, given its preceding context. Texts generate by LLMs tend to exhibit lower entropy scores.
- **LRR (Log-Likelihood Log-Rank Ratio)**: This approach utilizes both log-likelihood and log-rank.
- **Binoculars**: This approach utilizes two language models to calculate cross-perplexity, and based on this, it detects LLM-generated text. Binoculars utilizes the Falcon 7B and Falcon-Instruct 7B (Almazrouei et al., 2023) models to compute cross-perplexity.

Perturbation-based Detection

- **DetectGPT**: DetectGPT evaluates how the model's log probability function responds to slight modifications made to the original text. The underlying idea is that text produced by an LLM typically resides at a local optimum of the model's log probability function. Consequently, any small changes to text generated by the LLM are likely to result in a decreased log probability compared to the original text.
- **NPR (Normalized Log-Rank Perturbation)**: This approach is based on the observation that when slight perturbations are applied, the log-rank score of text written by LLMs increases more than that of text written by humans.
- **DetectGPT4Code**: DetectGPT4Code is a modified version of DetectGPT that uses models specialized for code to apply perturbations to a given code snippet and to calculate the log probability of the code snippet.

Paraphrasing-based Detection

BARTScore-CNN is based on the assumption that ChatGPT makes fewer modifications to text written by LLMs than to text written by humans. It involves editing the given text with ChatGPT and then calculating the similarity to the original text using BARTScore. The calculated similarity score is used to detect LLM-generated text.

H Experimental Results of Task 2

Table 6 represents the experimental results of task 2 (Code Generator Classification). We report the average performance and standard deviation of experiments across five different seeds. Overall, among the three classification models, CodeBERT exhibits the most superior performance.

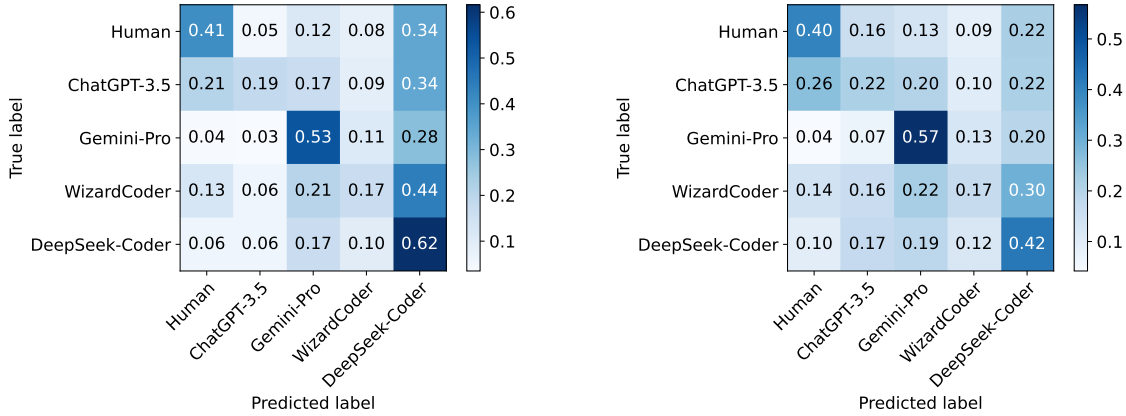
	OpenAI Detector		ChatGPT Detector		CodeBERT	
	Macro F1	Accuracy	Macro F1	Accuracy	Macro F1	Accuracy
C	$36.53_{\pm 0.73}$	$37.18_{\pm 0.99}$	$26.91_{\pm 2.62}$	$32.08_{\pm 1.12}$	$42.51_{\pm 1.60}$	$43.23_{\pm 1.73}$
C++	$34.35_{\pm 0.64}$	$36.40_{\pm 0.64}$	$31.98_{\pm 0.98}$	$35.17_{\pm 1.10}$	$39.04_{\pm 1.38}$	$41.63_{\pm 0.88}$
Java	$35.94_{\pm 1.23}$	$38.10_{\pm 0.19}$	$36.64_{\pm 0.69}$	$37.76_{\pm 0.59}$	$36.83_{\pm 1.16}$	$37.94_{\pm 0.52}$
Python	$35.38_{\pm 1.10}$	$39.06_{\pm 0.37}$	$28.45_{\pm 1.51}$	$34.57_{\pm 0.48}$	$34.08_{\pm 0.56}$	$36.86_{\pm 0.57}$
Overall	$35.55_{\pm 0.92}$	$37.69_{\pm 0.55}$	$30.99_{\pm 1.45}$	$34.89_{\pm 0.82}$	$38.11_{\pm 1.18}$	$39.91_{\pm 0.92}$

Table 6: Task2) Code Generator Classification Performance. We report the average performance and standard deviation of experiments using five different seeds.

I Confusion Matrices for the Experimental Results of Task 2

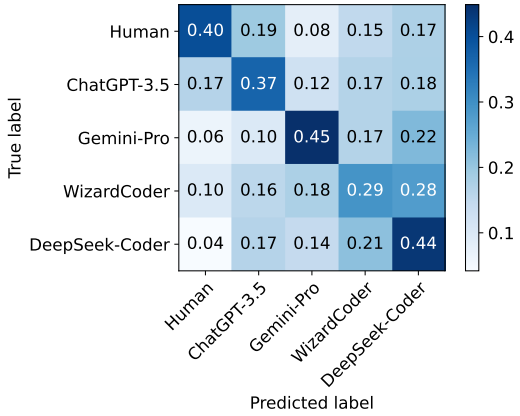
Figure 20 displays the confusion matrices of model predictions for the test split of the DeCo-Multi dataset (overall results for four programming languages). In all three classification models, predicting the generator of the given code snippet as DeepSeek-Coder is the most common scenario. We observe that CodeBERT shows a significantly more balanced prediction across each class compared to the other two classification models.

Finding. In the task of predicting the author of a given code snippet, CodeBERT provides more balanced predictions than the OpenAI Detector and ChatGPT Detector.



(a) OpenAI Detector

(b) ChatGPT Detector



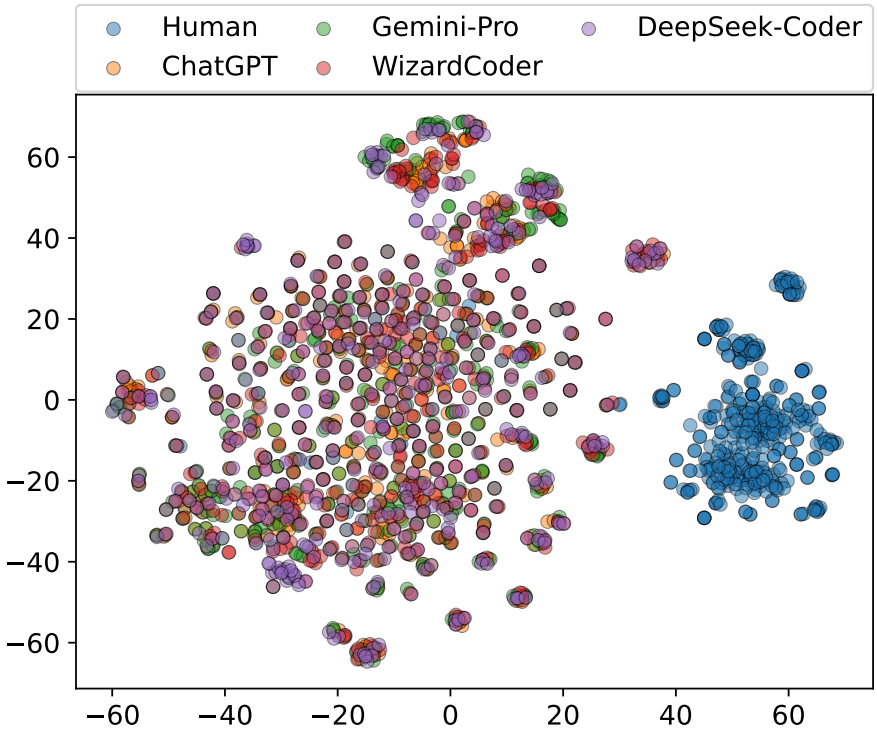
(c) CodeBERT

Figure 20: Confusion matrices of model predictions for the test split of the DeCo-Multi dataset. These provide overall results for the four programming languages. CodeBERT makes much more balanced predictions compared to the other two models.

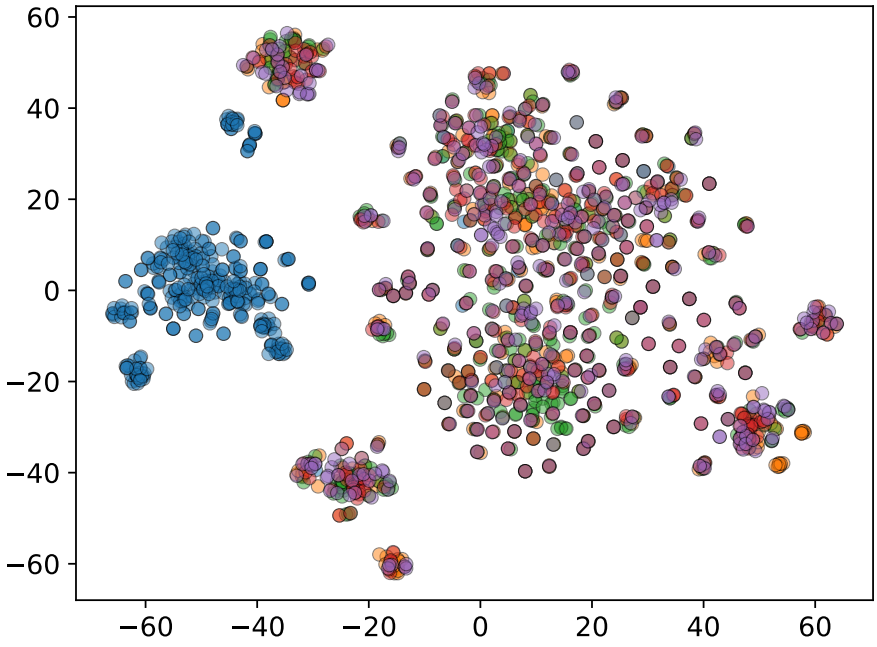
J Embedding Analysis

Figures 21, and 22 respectively visualize the embeddings of code snippets from the DeCo-Multi dataset for each programming language. We visualize the embeddings obtained using CodeLlama 7B with t-SNE (Van der Maaten and Hinton, 2008). For all four programming languages, embeddings of code snippets written by humans and LLMs are clearly distinguishable, but there is almost no distinction between embeddings of code snippets written by LLMs. This demonstrates the difficulty of predicting which LLM authored a given code snippet, underscoring the need for research into effective classification models for this task.

Finding. There is a clear distinction between the embeddings of human-written code snippets and those written by LLMs, but there is little distinction among the embeddings of code snippets written by different LLMs.

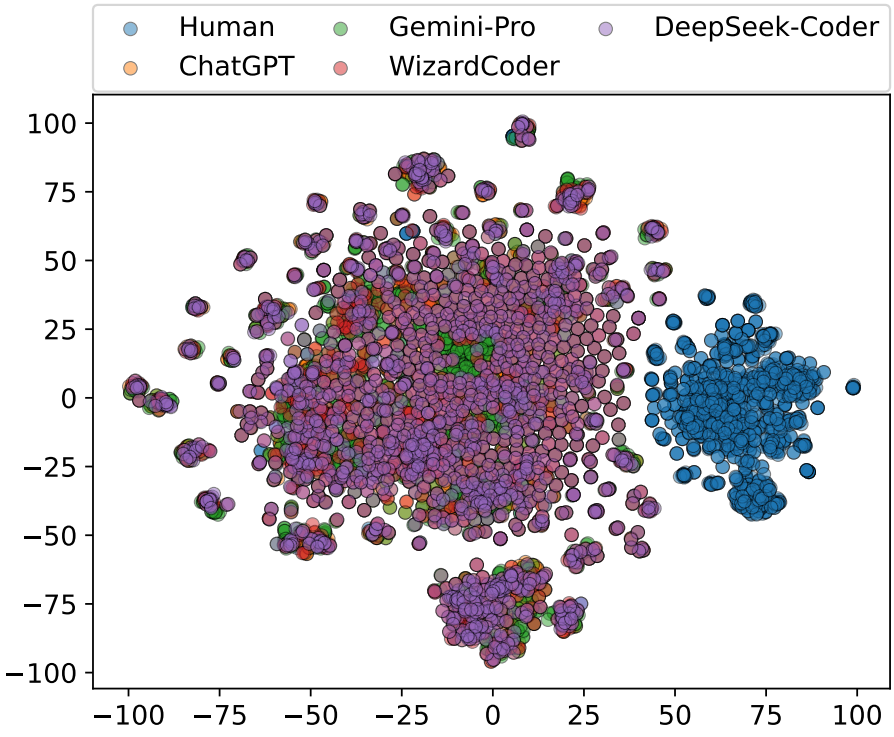


(a) C

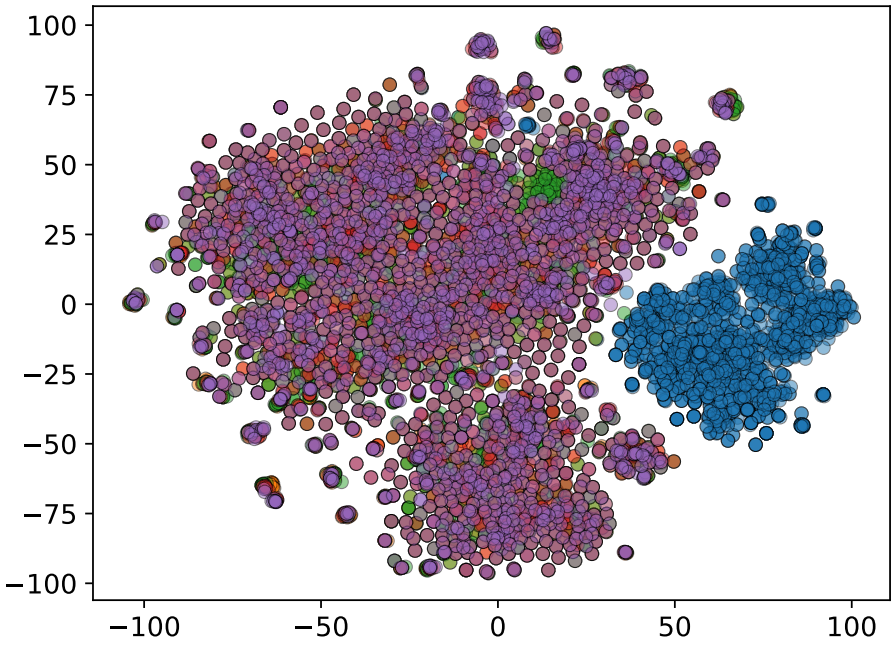


(b) C++

Figure 21: Analysis of embeddings for code snippets from the DeCo-Multi dataset. We visualize embeddings obtained using the CodeLlama 7B model. While embeddings are clearly distinguishable between humans and LLMs, there is little distinction among LLMs.



(a) Java



(b) Python

Figure 22: Analysis of embeddings for code snippets from the DeCo-Multi dataset. We visualize embeddings obtained using the CodeLlama 7B model. While embeddings are clearly distinguishable between humans and LLMs, there is little distinction among LLMs.

K Linguistic Analysis

Tables 7 and 8 respectively show the results of linguistic analysis for the DeCo-Multi dataset and the results of linguistic analysis for the DeCo-Multi dataset after removing natural language comments. Table 8 reveals that excluding the case of Java, human-written code snippets have the lowest density and the lowest average log-perplexity across all languages.

Finding. Regardless of the presence of natural language comments, code snippets authored by humans generally exhibit lower density values and lower average log-perplexity values compared to code snippets authored by LLMs.

C	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Vocabulary Size	21,003	19,394	16,924	18,125	17,188
Avg. Length	245.66	184.63	163.89	175.36	155.43
Density	18.70	22.98	22.59	22.61	24.19
Avg. Log-Perplexity	0.49	0.51	0.50	0.54	0.55

C++	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Vocabulary Size	18,056	16,980	15,925	16,361	14,841
Avg. Length	200.60	163.05	155.22	156.30	136.81
Density	23.37	27.04	26.64	27.18	28.17
Avg. Log-Perplexity	0.48	0.55	0.51	0.53	0.54

Java	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Vocabulary Size	62,998	59,817	57,077	54,332	50,465
Avg. Length	221.68	173.96	168.37	159.59	137.92
Density	19.02	23.01	22.69	22.78	24.49
Avg. Log-Perplexity	0.42	0.45	0.44	0.47	0.47

Python	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Vocabulary Size	98,985	91,574	91,162	87,627	81,044
Avg. Length	234.12	175.17	181.14	176.15	148.77
Density	21.84	27.01	26.00	25.70	28.15
Avg. Log-Perplexity	0.66	0.64	0.63	0.68	0.65

Table 7: Linguistic analysis on the DeCo-Multi dataset. We can see that, overall, code snippets written by humans have lower density values and smaller average log-perplexity values compared to those written by LLMs.

C	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Vocabulary Size	15,215	16,193	14,492	14,651	14,224
Avg. Length	287.95	285.15	269.91	257.91	239.67
Density	11.56	12.42	11.74	12.43	12.98
Avg. Log-Perplexity	0.38	0.40	0.40	0.41	0.43

C++	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Vocabulary Size	14,725	15,201	14,117	14,131	13,487
Avg. Length	285.19	139.10	253.71	257.48	240.78
Density	13.41	28.38	14.45	14.25	14.54
Avg. Log-Perplexity	0.38	0.51	0.39	0.39	0.40

Java	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Vocabulary Size	53,513	55,362	52,468	49,592	48,638
Avg. Length	147.68	145.19	144.92	136.74	131.39
Density	24.25	25.52	24.23	24.27	24.77
Avg. Log-Perplexity	0.42	0.46	0.44	0.47	0.47

Python	Human	ChatGPT	Gemini-Pro	WizardCoder	DeepSeek-Coder
Vocabulary Size	81,587	83,115	80,915	76,989	75,772
Avg. Length	147.45	142.30	140.04	132.77	130.84
Density	28.59	30.18	29.85	29.96	29.92
Avg. Log-Perplexity	0.59	0.62	0.60	0.63	0.63

Table 8: Linguistic analysis on the DeCo-Multi dataset after removing natural language comments. We can see that, overall, code snippets written by humans have lower density values and smaller average log-perplexity values compared to those written by LLMs.

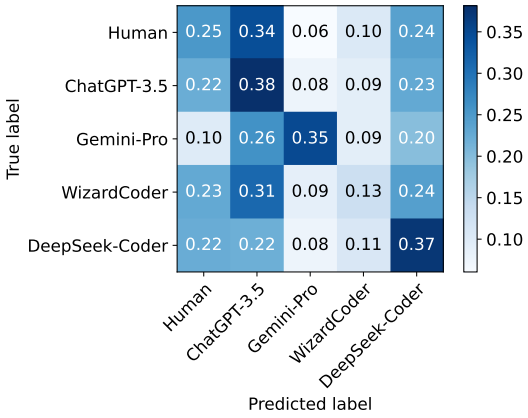
L Impact of Natural Language Comments

Table 9 presents the results of the analysis on the impact of natural language comments in predicting the author of a given code snippet. Both the OpenAI Detector and the ChatGPT Detector experience significant performance degradation when natural language comments are removed. This indicates that natural language comments are an important feature in classifying the author of a code snippet. Figure 23 displays the confusion matrices of model predictions for the test split of the DeCo-Multi dataset after removing natural language comments (overall results for four programming languages).

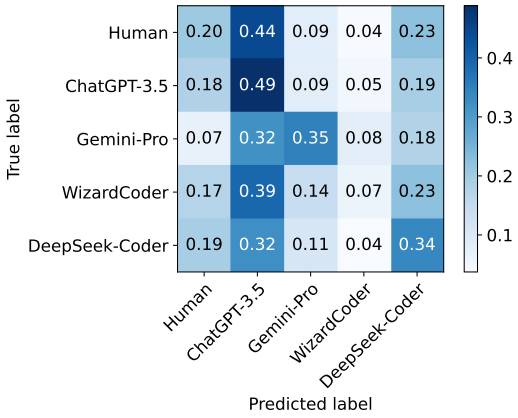
Finding. In the task of predicting the author of a given code snippet, natural language comments are an important feature.

	NL	OpenAI Detector		ChatGPT Detector	
		Macro F1	Accuracy	Macro F1	Accuracy
C	O	36.53	37.18	26.91	32.08
	X	27.79 (↓ 8.74%p)	29.78 (↓ 7.40%p)	18.62 (↓ 8.29%p)	27.06 (↓ 5.02%p)
C++	O	34.35	36.40	31.98	35.17
	X	24.64 (↓ 9.71%p)	28.40 (↓ 8.00%p)	19.27 (↓ 12.71%p)	25.22 (↓ 9.95%p)
Java	O	35.94	38.10	36.64	37.76
	X	31.77 (↓ 4.17%p)	33.86 (↓ 4.24%p)	28.06 (↓ 8.58%p)	31.86 (↓ 5.90%p)
Python	O	35.38	39.06	28.45	34.57
	X	21.66 (↓ 13.72%p)	27.02 (↓ 12.04%p)	21.17 (↓ 7.28%p)	27.54 (↓ 7.03%p)
Overall	O	**35.55**	**37.68**	**30.99**	**34.89**
	X	**26.46** (↓ **9.09%p**)	**29.76** (↓ **7.92%p**)	**21.78** (↓ **9.21%p**)	**27.92** (↓ **6.97%p**)

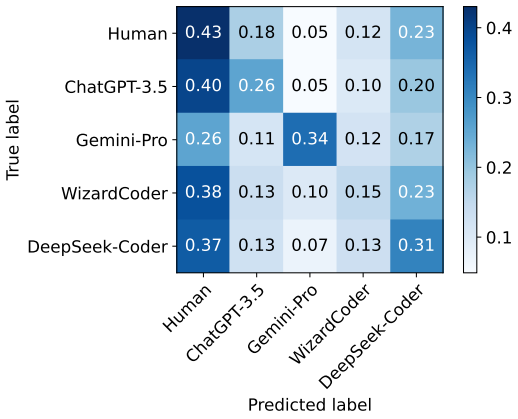
Table 9: We analyze the impact of natural language comments in the task of identifying the authors of code snippets. We report the average performance of experiments conducted with five different seeds.



(a) OpenAI Detector



(b) ChatGPT Detector



(c) CodeBERT

Figure 23: Confusion matrices of model predictions for the test split of the DeCo-Multi dataset after removing natural language comments. These provide overall results for the four programming languages.