Assessing the Promise and Pitfalls of ChatGPT for Automated CS1-driven Code Generation

Muhammad Fawad Akbar Khan*†, Max Ramsdell*†, Erik Falor, and Hamid Karimi†
Department of Computer Science, Utah State University
Emails:{khan@usu.edu; a02237674@usu.edu; erik.falor@usu.edu; hamid.karimi@usu.edu}

ABSTRACT

This paper undertakes a thorough evaluation of ChatGPT's code generation capabilities, contrasting them with those of human programmers from both educational and software engineering standpoints. The emphasis is placed on elucidating its importance in these intertwined domains. To facilitate a robust analysis, we curated a novel dataset comprising 131 code-generation prompts spanning five categories. The study encompasses 262 code samples generated by ChatGPT and humans, with a meticulous manual assessment methodology prioritizing correctness, comprehensibility, and security using 14 established code quality metrics. Noteworthy strengths include ChatGPT's proficiency in crafting concise, efficient code, particularly excelling in data analysis tasks (93.1% accuracy). However, limitations are observed in handling visual-graphical challenges. Comparative analysis with human-generated code highlights ChatGPT's inclination towards modular design and superior error handling. Machine learning models effectively distinguish ChatGPT from human code with up to 88% accuracy, indicating detectable coding style disparities. By offering profound insights into ChatGPT's code generation capabilities and limitations through quantitative metrics and qualitative analysis, this study contributes significantly to the advancement of AI-based programming assistants. The curated dataset and methodology establish a robust foundation for future research in this evolving domain, reinforcing its importance in shaping the future landscape of computer science education and software engineering. Code and data are available here: https://github.com/DSAatUSU/ChatGPT-promises-and-pitfalls

Keywords

ChatGPT, GPT3.5 Turbo, Code Generation, Computer Science Education, Python, Code Metrics, Machine Learning

*Co-first authors and equal contributions.
†Data Science and Applications Lab @ USU
https://dsa.cs.usu.edu

M. F. A. Khan, M. Ramsdell, E. Falor, and H. Karimi. Assessing the promise and pitfalls of chatgpt for automated cs1-driven code generation. In B. Paaßen and C. D. Epp, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, pages 83–95, Atlanta, Georgia, USA, July 2024. International Educational Data Mining Society.

© 2024 Copyright is held by the author(s). This work is distributed under the Creative Commons Attribution NonCommercial NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license.
https://doi.org/10.5281/zenodo.12729778

1 Introduction

Emerging technologies and changing societal demands have driven the digital transformation in education [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. A significant catalyst in this digital revolution is the rise of Generative Pre-trained Transformer (GPT) models [11], specifically ChatGPT [12] created by OpenAI. GPT is an advanced language model tailored for conversational applications such as question-answering and code generation. Built upon the GPT-3.5 series architecture, ChatGPT boasts an impressive 175 billion parameters and has been fine-tuned using reinforcement learning with human feedback. This extensive training enables ChatGPT to produce responses that closely resemble human language based on context comprehension. The ongoing conversation history has garnered substantial interest within the software engineering community. This interest is primarily attributed to ChatGPT's reported capability in realizing a longstanding aspiration in software engineering: the automatic repair of software with minimal human intervention [13]. These reported outcomes suggest that ChatGPT holds transformative potential for the field, indicating a promising future for LLM-driven software engineering and AI programming assistant tools. However, further research is imperative to precisely delineate the extent of LLM capabilities specifically for generating programs.

While the utilization of AI-based code generation presents promising prospects for enhancing productivity and automating various software development tasks, it introduces critical considerations. The process of code generation by LLMs faces several challenges, such as the need to ensure functional correctness, comprehensibility, and security of the generated code. Additionally, the primary concern regarding ChatGPT-generated code pertains to its correctness. Given that two vastly different code snippets can possess semantic equivalence, conventional NLP metrics like BLEU score [14] lack reliability in the context of program synthesis. Ideally, we aspire to validate the accuracy of ChatGPT-generated solutions for any input formally. However, validating domain-specific problems using techniques like translation validation [15, 16] is already a formidable task, and constructing a universal verifier with absolute certainty for arbitrary problems, including those in code benchmarks, presents an even greater challenge.

This paper focuses on conducting an extensive and systematic evaluation of code generated by ChatGPT, with a particular emphasis on assessing its correctness, comprehensibility, and security. We have chosen ChatGPT, a prominent and widely recognized LLM, to serve as a representative example of LLMs. Many studies that focus on generating pro-

grams with ChatGPT tend to assess its performance using outdated benchmark data available prior to 2022–See Section 2. This outdated data might have inadvertently influenced the training data for ChatGPT. Additionally, none of these studies conduct a comparative analysis between ChatGPT-generated code and human-written code, effectively highlighting the limitations of using the ChatGPT model for code generation. This potential bias in experimental design raises concerns about the applicability of the reported results to new and unforeseen challenges. Furthermore, the existing literature lacks a comprehensive examination of ChatGPT's capabilities in code generation, emphasizing the need for further investigation in this area. By conducting this comprehensive analysis, we aim to contribute to advancing ChatGPT-based code generation techniques, potentially enhancing the broader field of AI and LLM-based code generation. In this study, we aim to provide insightful answers to the following fundamental questions:

- ☆ **Functional Accuracy**: Can ChatGPT outperform humans in generating functional and highly accurate code?
- ☆ **Code Understandability**: Is ChatGPT capable of producing code that is more understandable than that of humans?
- ☆ **Code Security**: Can ChatGPT demonstrate superior capabilities in generating code that is more secure compared to human-generated code?
- ☆ **ChatGPT code detection**: Is it possible to develop a prediction model that can reliably distinguish between ChatGPT-generated and human-generated code reliably, achieving a significant level of accuracy?

Our contributions that followed in this study are summarized below.

- ❒ **Constructing a New Dataset:** We curate a diverse dataset of 131 prompts across five categories and 25 subcategories featuring human-written code. This dataset forms the foundation for a robust comparative analysis, assessing the efficacy of code-generation algorithms against human-coded solutions.
- ❒ **Comprehensive Code Evaluation:** Using ChatGPT3.5 Turbo, a widely recognized LLM, we systematically evaluate generated code, prioritizing correctness, understandability, maintainability, and security. This evaluation employs 14 interpretable code quality metrics.
- ❒ **Comparative Analysis with Human Code:** We conduct a comparative analysis between ChatGPT-generated and human-written code across 131 prompts in five categories, revealing both limitations and strengths in ChatGPT's code generation across different categories of code.
- ❒ **Critical Analysis of Prompts:** Our research emphasizes the influence of prompt quality on ChatGPT's code generation capabilities through case studies, providing key considerations for prompt design, an emerging skill in LLMs.
- ❒ **ChatGPT Code Detection:** Using introduced features, we develop machine learning (ML) models to classify ChatGPT code versus human code. Additionally, we conduct a reliability test to ensure the model's generalizability. To the best of our knowledge, this is the first ML ChatGPT code detection model.

Significance in Computer Science Education and Software Engineering

The integration of large language model (LLM) code generation models, exemplified by ChatGPT, into software engineering and education is inevitable. This research holds broader significance, providing valuable insights into the evolving capabilities of AI-based code generation tools. As large language models progress rapidly, it becomes imperative for the software engineering and computer science education communities to develop a nuanced understanding of their promise and limitations. Our work systematically evaluates ChatGPT's code generation proficiency, comparing it to human learners or programmers, thereby delineating the current state of its abilities and its potential as a true programmer assistant.

The findings, particularly relevant to the field of education, address the increasing prevalence of intelligent tutoring systems and AI teaching assistants. It is critical to precisely determine their competencies and shortcomings in automating programming education. This analysis facilitates the integration of LLM-based technologies into the education system, offering data-driven guidance on ChatGPT's reliability in grading programming assignments, providing feedback, and assisting students. Educators can leverage our insights to make informed decisions about incorporating ChatGPT, considering its limitations, such as challenges with visual and advanced tasks.

Moreover, the introduced dataset and comparative analysis methodology lay the groundwork for future learning analytics research assessing AI tutors. As educational institutions blend online tools with in-person instruction, robust evaluations of AI's pedagogical effectiveness become imperative. This research pioneers techniques to gauge automated programming tutors' capacities across languages, problem categories, and metrics like correctness. By promoting prudent AI adoption and advancing evaluation methods, this research delivers significant broader impacts for the learning analytics community, underscoring its relevance in shaping the future of computer science education.

2 Related Work

Since its inception, GPT-based LLMs have been getting incredible attention in the research community mainly because of their exceptional abilities in Natural Language Processing and Code Generation [13]. Various studies have attempted to evaluate GPT code generation capabilities and address their shortcomings [17, 18]. Fan et al. [19] systematically investigated whether automated program repair (APR) techniques, including Codex, can rectify incorrect solutions generated by Codex for LeetCode problems. Xia et al. [20] conducted a comprehensive study involving the direct application of nine state-of-the-art Language Models (LLMs) for Automated Program Repair (APR). Their evaluation encompassed various approaches for utilizing LLMs in APR, such as entire patch fixes, code chunk fixes and single-line fixes. Hendricks et al. [21] introduced a benchmark for Python programming problems called "craftAPPS". They evaluated the code generation performance of several GPT-

based variant models by fine-tuning them with the craftAPPS dataset. Dong et al. [22] introduced the idea of a software development lifecycle and put forth a self-collaboration framework. This framework utilizes distinct GPT conversations in various roles, such as analyst, developer, and tester, to collaborate in the code generation process. Liu et al. [23] analyzed various code quality issues associated with GPT-based code generation. However, most of these studies utilize a publically available dataset, for example, LeetCode problems [24] and CWE (Common Weakness Enumeration) scenarios (CWE's code scenarios) as provided in [25]. The challenge with existing code datasets lies in their lack of customization to effectively evaluate GPT's code generation capabilities. For instance, these datasets may neglect scenarios involving the creation of visually intensive, graphical, or drawing-oriented programs. Furthermore, the effectiveness of the GPT for code generation is poorly understood, and the generation performance could be heavily influenced by the choice of prompt [26]. Therefore, using an outdated prompt dataset such as (e.g., OpenAICookbook [27], and PromptBase [28]) may not be an effective option. Prompts should be custom engineering, offering sufficient information to GPT while leveraging its dialog ability. Additionally, conducting preliminary analyses of the prompts can yield deeper insights into GPT's code-generation abilities, potentially leading to more suitable prompts for an enhanced dataset. The datasets for analysis should be thoroughly tailored to encompass a comprehensive spectrum of programming aspects, spanning various categories and subcategories. This approach can effectively uncover vulnerabilities and limitations within GPT's code generation capabilities.
Against this backdrop, our research stands out. We introduce a meticulously crafted prompt dataset spanning five distinct categories, offering a richer evaluation canvas than generic benchmarks. By juxtaposing ChatGPT code with human-generated code, we attain a deeper comprehension. Our use of 14 code quality metrics and novel machine learning models for ChatGPT code detection sets a new standard in the domain. Our holistic approach, emphasizing rigorous benchmarks and thorough methodologies, brings forth novel insights into ChatGPT's code generation capabilities. Consequently, our contributions lay a robust groundwork that can spur further innovations in AI-driven programming tools.

3 Data Collection and Curation Process

To maximize ChatGPT's performance, we conducted our study using Python3, a highly expressive programming language. A study done by Zhijie Liu et al. verified that ChatGPT is better at generating Python3 code in terms of understandability, functionality, and security metrics compared to 4 other languages C++, C, JavaScript, and Java [18].

3.1 Programming Prompt Conceptualization

To pinpoint the focal area for prompt collection, we conducted a preliminary investigation, thoroughly exploring different facets of ChatGPT's code generation capabilities. This investigation utilized assignments from the *"Introduction to Python Programming"* course (CS1) at a major US public university as ChatGPT's initial prompts. Tailored for computer science undergraduates without prior programming experience, these assignments covered diverse programming concepts, including visual and drawing problems, algorithms, data structures, loops, and object-oriented programming. The insights from the preliminary prompts guided our decision-making on categories and subcategories essential for a comprehensive analysis of ChatGPT's code-generation capabilities, as depicted in Figure 1. To construct this framework, we gathered prompts from diverse online platforms, including Github, Medium, GeekforGeeks, and others.

- ➪ **Algorithms and Data Structures (ADS):** This category features prompts of varying difficulty levels in *algorithms and data structures*, evaluating ChatGPT's ability to devise solutions for sorting, searching, recursion, optimization, arrays, linked lists, trees, and graphs. This category assesses its understanding of mathematics and computer science concepts.

- ➪ **Data Analysis (DA):** This category includes prompts of diverse complexity in *data analysis*, assessing ChatGPT's capacity to generate code for data cleaning, manipulation, visualization, and statistical analysis. The goal is to evaluate ChatGPT's proficiency in addressing real-world data tasks.

- ➪ **Mathematics (M):** This category incorporates prompts of diverse complexity in *mathematics*, covering basic geometry, trigonometry, arithmetic, algebra, calculus, and advanced topics. This category evaluates ChatGPT's ability to generate code solutions for a range of mathematical problems.

- ➪ **Object Oriented (OO):** This category includes prompts of varying complexity within *object-oriented programming*, addressing tasks related to class design, inheritance, polymorphism, encapsulation, and design patterns. The aim is to assess ChatGPT's ability to generate code adhering to object-oriented principles.

- ➪ **Visual Graphical Drawing (VGD):** The last category focuses on ChatGPT's ability to generate code for *visual patterns, drawing, and graphical* challenges. This category includes tasks ranging from turtle graphics patterns to visualizing directions, complex designs, GUIs, pixel art, and image processing. Given ChatGPT's potential challenges in this domain (described in Section 5), evaluating its code generation proficiency here is crucial.

3.2 Collection of Prompts

A total of 131 prompts were collected, and a prompt template was designed for consistency, featuring four essential components: **1) Preamble:** Sets up the environment and directories and defines the GPT model's role and context. **2) Prompt:** Specifies the task and programming language for code generation. **3) Output Formatting:** Details output requirements such as Python package use, commenting, and single-file output, with no command-line arguments and optional user input. **4) Exporting:** Saves the generated code as a .py file in the correct directory. A shell script was used to automate code collection from the prompts. Out of the total prompts, 60 were sourced online, and 71 were specifically crafted for this study, predominantly in the *visual-graphica-drawing* and *algorithms-data-structures* categories, representing 23% and 26% of the prompts, respectively.

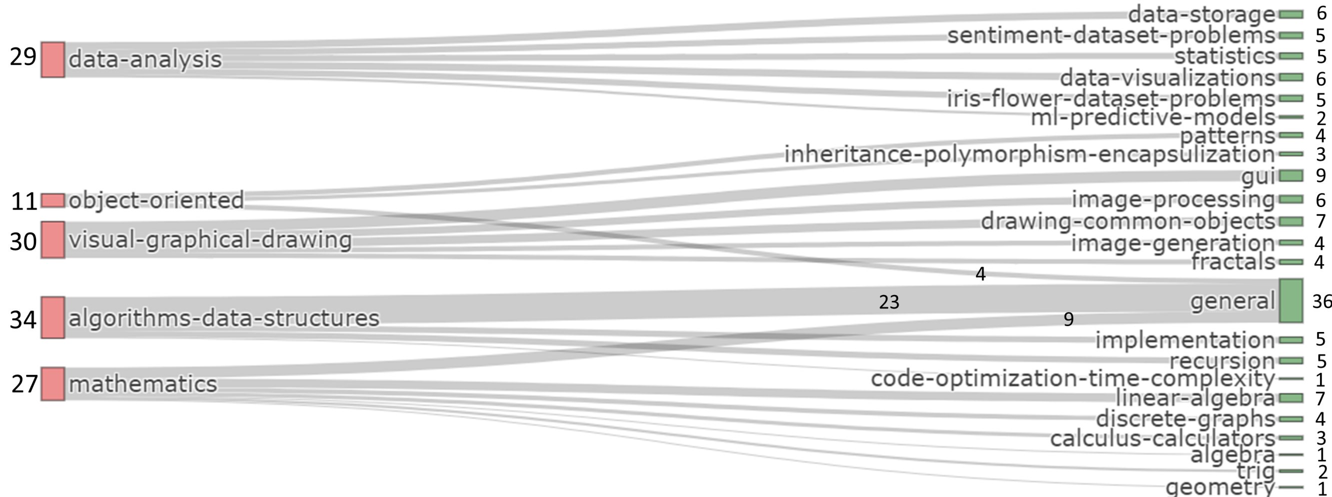


Figure 1: Categories and subcategories for which data and prompts were collected. The numbers show the total number of prompts collected for that category or subcategory.

3.3 ChatGPT Code Generation

To streamline the process, we developed a shell script that uses OpenAI's GPT API to harness the GPT-3.5 Turbo model (July 2023) to automatically generate code solutions for each of the 131 prompts. To ensure the quality and functionality of the code generated from the prompts, we began by manually evaluating a few sample codes. We then iteratively refined the prompts to ensure that ChatGPT performs to its fullest potential before proceeding to generate code for all 131 prompts. Following the successful validation of our automated process, we efficiently generated code solutions for all 131 prompts, resulting in a total of 131 code files. Subsequently, we organized the obtained code into separate *.py* files and meticulously structured them within nested directories to ease subsequent analysis.

3.4 Human Code Collection

To create the human-coded dataset for all 131 prompts, we engaged six computer science major individuals, each responsible for selecting prompts based on their preference. They were instructed to produce code solutions independently in their unique coding styles, with strict guidance not to utilize AI assistance tools, including ChatGPT, GitHub Copilot, or Kite. However, they could access online resources to collect ideas to generate the code. Approximately 80 prompts were meticulously crafted through manual coding in this manner. In order to diversify the dataset to encompass a broader range of coding styles, an additional approximately 51 codes were sourced from various online platforms. These codes were meticulously curated to ensure their authors were human programmers. Particular emphasis was placed on retrieving code snippets from the most recent (after *Sept 2021*) or non-public sources whenever feasible to guarantee that the code had not been incorporated into the training set of ChatGPT 3.5.

4 Methodology

Utilizing a dataset of 262 code samples (131 from ChatGPT and 131 from humans), we performed a collaborative effort for a manual evaluation of each code. The focus was primarily on assessing functionality. A customized test suite was meticulously designed for each prompt. This approach facilitated a comprehensive examination of the code's capabilities and helped identify potential limitations in ChatGPT's coding capabilities. For codes resulting in compilation errors, runtime errors, or incorrect output, a corrective process ensued, spanning up to 10 rounds. Task-related information was incrementally provided within the prompts with each successive round to guide the code generation process. Codes that remained unsuccessful even after the 10-round correction process were categorized as incorrect. We did not impose any time or round limits for human-generated code, allowing for the completion and execution of all codes, consistently yielding correct outputs. For ChatGPT-generated code, the authors manually assessed the functional correctness of codes. In the subsequent sections, we elaborate on the code metrics utilized in our analysis and elucidate the methodology employed for training the machine learning algorithm.

4.1 Code Analytical Metrics

To comprehensively assess ChatGPT's code compared to human-generated code, we employed 14 well-established programming metrics commonly utilized in the research literature. These metrics serve a dual purpose, enabling us to not only gain insights into the coding style of ChatGPT and humans but also facilitate in-depth examinations of specific coding behaviors, thus enhancing our observations. This approach allows for a thorough evaluation of functionality, understandability, and security.

1. **Cyclomatic Complexity:** Cyclomatic Complexity quantifies the number of linearly independent paths in code, offering a pure complexity measurement. Lower complexity, indicative of fewer branches, aligns to achieve comprehensive code coverage. This metric is frequently employed to determine the number of paths necessary for full testing coverage. Additionally, it has been observed that Cyclomatic Complexity correlates strongly with Source Lines of Code (SLoC) and may share similar predictive capabilities.

2. **Halstead Metrics:** Halstead metrics are derived from the code's count of operators (e.g., '+' or 'int') and operands (e.g., variable names or numbers). These metrics aim to quantify the "physical" properties of code, similar to how physical matter is characterized by mass and volume. **1) Halstead Difficulty**: Measures the code's readability and understanding. **2) Halstead Effort**: Estimates the effort required to write the code. **3) Halstead Volume**: Reflects the program's size, including operator and operand counts. **4) Halstead Time**: Predicts the time needed to develop the program. **5) Halstead Bugs**: Estimates potential bug count, aiding debugging efforts.

3. **Source and Logical Lines of Code (SLoC and LLoC):** The physical number of lines of code. Logical Lines of Code (LLoC) takes a file's total lines of code and removes whitespace lines and comment lines—the number of statements in a program. For example, a line with a print statement after an if statement on the same line would have 1 SLoC and 2 LLoC.

4. **Difference of SLoC LLoC:** Since the difference between SLoC and LLoC can provide insight into code quality. Higher quality code will have lower difference values. Therefore, we add that as a metric.

5. **Number of Lines:** Total number of lines in the code. That is, counting code, new lines, and comments. This is different from LLoC and SLoc.

6. **Number of Comments, Functions, and Classes :** The total number of comments in the code. This counts the comments by counting the start of each comment. The number of Functions count the number of functions used in code. The number of Classes counts the number of classes used in code.

7. **Maintainability Index:** The Maintainability Index is a software metric that measures how maintainable and comprehensible a software system is. It considers factors such as code size, complexity, and coupling, providing a numerical score that reflects the ease with which developers can understand, modify, and maintain the codebase. A higher Maintainability Index indicates better code maintainability.

Using these code metrics, we embark on an in-depth analysis using scientific visualization to identify the nuance difference between human-generated and ChatGPT-generated code.

4.2 ChatGPT Code Detection

Utilizing 14 code metrics, we trained seven machine learning algorithms—including Decision Trees (DT), Random Forest (RF), and K-Nearest Neighbors (KNN)—to discern between ChatGPT and human code. The typical training procedure involved:

- **Train-Test Split:** An 80-20% ratio ensured 210 codes in the training set and 52 in the test set from both ChatGPT and humans.

- **Hyperparameters Tuning:** Parameters were determined through grid search based on literature and experimentation.

- **Training:** Each algorithm was trained with optimized hyperparameters, and performance was gauged using standard metrics. To mitigate randomness, models were trained multiple times with varied seeds.

4.2.1 Reliability Tests.

We conducted reliability tests to evaluate the classifiers' real-world applicability:

- **Train-Test Split Ratio Test:** We experimented with train-test ratios from 10% to 100% in 5% increments.

- **Per-category Performance Test:** Checked model performance across categories to prevent overfitting to any single category.

- **Gaussian Noise Test:** The classifiers' resilience was assessed by incrementally adding Gaussian noise, ranging from 0 to 1 in 0.01 steps, to the standard deviation of the noise generator.

4.2.2 Feature Analysis.

The 14 features across 262 data instances were analyzed to discern coding style variations. Two methods were adopted:

- **Random Forest-based Approach:** Evaluated importance through impurity decrease metrics.

- **Feature Permutation-based Technique:** Assessed feature relevance by modifying performance metrics after feature permutation.

By averaging results from both methods over 1000 iterations, we gained a robust understanding of feature importance, balancing the strengths and weaknesses of each method.

5 Experimental Results

This section presents experimental results delineating ChatGPT deep code analysis and prediction.

5.1 Functional Accuracy

Since we didn't impose any time and attempt limits on Humans, functionally correct codes for all 131 prompts were generated. For ChatGPT, After the assessment of 131 codes for functional accuracy, the results for the 131 prompts were categorized into three classes:

1. Correct and Compilable

These codes were not only accurate in fulfilling the task description but also provided the required output across the test suite. Among the accurately generated prompts, a substantial majority were classified under the Data Analysis category, with 93.1% of the prompts being correct. This highlights ChatGPT's robust comprehension of concepts related to predictive modeling, data storage, statistics, and data visualizations. Out of the total 16 failures, 7 (23.3%), 3 (8.8%), 3 (11.11%), 2 (6.8%), and 1 (9%) occurred in the VGD, ADS, M, DA, and OO categories, respectively. All Human and ChatGPT codes are provided in the GitHub repository.

Listing 1: An example of error handling, code commenting, and debugging information style

CHATGPT

```python
def compress_file(file_path):
    """
    Compresses a text file using zlib compression algorithm.

    :param file_path: path to the text file
    """
    try:
        with open(file_path, 'rb') as file:
            input_data = file.read()
            compressed_data = zlib.compress(input_data, level=zlib.Z_

        with open(file_path + '.compressed', 'wb') as file:
            file.write(compressed_data)

        print('Compression successful. Compressed file saved as', file_
    except FileNotFoundError:
        print('File not found.')
    except Exception as e:
        print('An error occurred during compression:', str(e))
```

HUMAN

```python
def compress_file(filename):
    archiveName = input("Input zip archive name: ")
    with zipfile.ZipFile(archiveName, 'w') as zip:
        zip.write(os.path.join('./', filename))
        zip.close()
```

Listing 2: An Example of Functional Programming

CHATGPT

```python
# Connect to the database
def connect_to_database():
    # ...

# Create table if not exists
def create_table(connection):
    # ...

# Insert names into the table
def insert_names(connection, first_names,
    # ...

# Get names from user
def get_names():
    # ...

def main():
    # ...

# Call the main function
if __name__ == '__main__':
    main()
```

HUMAN

```python
# ...
def register_name():
    # ...

register_name()
```

Listing 3: Making Chessboard example

CHATGPT: Initial Attempt

```python
def drawRectangle(x, y, width, height):
    # ...
    turtle.forward(width)
    turtle.left(90)
    turtle.forward(height)
    turtle.left(90)
    turtle.forward(width)
    turtle.left(90)
    turtle.forward(height)
    turtle.penup()
```

Initial Attempt Output

CHATGPT: Final Attempt

```python
def drawRectangle(x, y, width, height):
    turtle.penup()
    turtle.goto(x, y)
    turtle.pendown()
    turtle.begin_fill()

    for _ in range(4):
        turtle.forward(width)
        turtle.right(90)

    turtle.end_fill()
```

Final Attempt Output

Figure 2: Codes and outputs for the presented case studies.

2. Incorrect and Compilable.
Codes falling into this category were deemed uncompilable for various reasons, such as compile errors, type errors, time limit exceeded errors or runtime errors. This category includes compilable codes that produce incorrect output. It also includes instances of codes with infinite loops resulting in a "time limit exceeded" condition. In another scenario, we observed that overloading ChatGPT with excessive details and imposing numerous conditions could adversely impact its code generation capabilities. This leads to ChatGPT desperately trying to solve the problem by generating compilable code yet with incorrect output. We provide insights about this category through Case Studies 1,2 and 3.

3. Incorrect and Uncompilable.
Non-compilable codes, marked by compile or runtime errors persisting even after 10 rounds of incremental correction attempts, revealed ChatGPT's struggle in adapting to outdated methods and functions. The model's lack of awareness of changes beyond September 2021 renders it a liability for coding tasks with newer contexts or requirements involving recently released packages. Additionally, the generated code may employ outdated methods, potentially leading to slower performance and suboptimal memory management. To provide more detail about the nature of this category, we present Case Study 4.

Conclusion.
In conclusion, our evaluation encompassed three key aspects of ChatGPT's coding capabilities. In generating *Correct and Compilable* code, ChatGPT exhibited proficiency with a commendable 93.1% accuracy, notably emphasizing modularity through a higher number of functions. However, when tackling *"Incorrect but Compilable"* codes, the model faced challenges in tasks requiring advanced visual imagination and multi-layered problem-solving, showcasing limitations in creativity and optimal code generation. Notably, ChatGPT lacks the ability to leverage visual feedback for quick error detection, hindering accuracy in drawing and visual problem-solving. Lastly, the examination of *"Incorrect and Uncompilable"* codes highlighted the model's struggle with outdated methods, emphasizing the importance of continual model updates to adapt to evolving coding practices. While ChatGPT's limitations were evident, successful corrective adjustments demonstrated the model's potential for improvement with targeted interventions, offering valuable insights for future advancements in natural language programming models.

5.2 Code Understandability, Maintainability and Security

In evaluating the potential and limitations of GPT-3.5 concerning code understandability, maintainability, and security, we applied the metrics introduced in Section 4.1 to all code samples. Subsequently, to facilitate comparison and discussion, we created box-and-whisker plots and radar plots (refer to Figures 3 and 4). These visualizations offer insights into the understandability, maintainability, and security of both ChatGPT and human codes. Analyzing programming styles, our observations are as follows:

❒ **Functional Programming:** The plots highlight ChatGPT's significant use of functions, particularly in the M and ADS categories, showcasing its inclination towards functional programming. In contrast, humans tend to minimize unnecessary functional usage. This distinction is evident in the spider plot, with a notable contrast in the number of functions used by ChatGPT and humans, especially in the M and ADS categories. However, this difference is less apparent in the OO, DA, and VGD categories, suggesting a nuanced response influenced by the emphasis on functional programming in the prompts. Additionally, as programs increase in size and complexity, humans typically introduce more functions. The strategic use of functional programming enhances ChatGPT's coding efficiency, leading to fewer Logical Lines of Code (LLoC) and a higher maintainability index.

❒ **Lines of Code and Comments** Significant disparities in the distribution of the Number of Lines of Code and Comments are evident across the entire dataset and within each category. Generally, humans tend to produce more lines of code and fewer comments, exhibiting sporadic behavior. In contrast, ChatGPT consistently generates concise and efficient code with succinct comments, typically not exceeding one line. This distinction is more pronounced in the VGD and ADS categories, containing the most challenging prob-

Case Study 1 (Drawing Chessboard): In our preliminary investigation, ChatGPT was tasked with replicating a Python script assignment where CS students at a large public university in the US created a chessboard using the Turtle module. The assignment was in a 3-page PDF file with detailed instructions about the task implementation and execution. Despite providing detailed prompts with instructions on how to implement and execute the task, ChatGPT consistently produced incorrect but compilable code, struggling with nuances like the chessboard border placement of missing tiles and misalignment of tiles and the box. Interestingly, simplifying the prompt to a concise instruction,"*Create Python code to generate an 8x8 chessboard using the Turtle Python package*," resulted in accurate code on the first attempt. This suggests that, akin to humans, ChatGPT excels at smaller and straightforward tasks. Notably, the performance improvement parallels the success of human students given a similar concise prompt for a smaller chessboard task, underscoring the impact of task framing on language model performance.

Case Study 2 (ChatGPT Directional Dyslexia): When addressing issues in the ChatGPT output, it became apparent that while ChatGPT correctly followed the prescribed steps, it encountered challenges in aligning the tiles within the chessboard bounding box or rectangle. A chessboard consists of tiles inside a rectangular box. To create a tile, we utilized the commands `pen.right()` and `pen.forward()` repeated four times. However, a problem arose as ChatGPT struggled to discern how using `pen.right()` or `pen.left()` to construct the tiles (black squares) would impact their alignment within the box–See Listing 3 Code and Output in Figure 2. Notably, the use of `pen.right()` resulted in the first row of tiles being positioned below and outside the chessboard, and this alignment issue was rectified by employing `pen.left()`.

Case Study 3 (Generating Sprite): Contextual meaning is crucial for ChatGPT to provide accurate output. For instance, a prompt instructing the machine to create a script generating "Sprites" for video games resulted in a script producing a flat image. Recognizing the misunderstanding, we modified the term "Sprite" to "pixel art image," leading to a script generating images resembling static on an old television—technically correct as a form of "random pixel art image." This highlights the significance of providing contextual framing for prompts, as ChatGPT may struggle with words carrying double meanings without such context, unlike humans familiar with the associated concepts.

Case Study 4 (Outdate Methods): ChatGPT relies on training data up to September 2021, potentially leading to outdated knowledge of packages. For example, when tasked to create a script using "sklearn" for a predictive model, ChatGPT chose the "Boston Housing Dataset," unaware it had been deprecated since sklearn's 1.2 update in December 2022. This limitation means ChatGPT might generate code with deprecated or nearly deprecated features, necessitating caution. In contrast, while not always the latest, human-written code can offer more recent, relevant, and reliable solutions. Despite initial failures, corrective adjustments, such as changing `load_boston()` to `fetch_california_housing()`, were deemed correct, showcasing a nuanced evaluation approach.

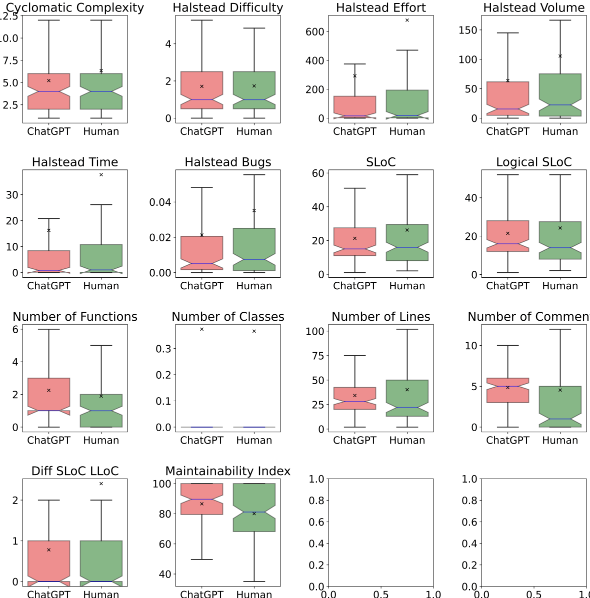


Figure 3: This box and whisker plot illustrate key values of analytical metrics: Median (line within the box), Mean ('x' symbol), Minimum (lower whisker), Maximum (upper whisker), Lower Quartile (bottom of the box), and Upper Quartile (top of the box) for each metric.

lems. Notably, ChatGPT tends to provide more comments for OO, M, and DA categories while offering fewer comments for VGD and ADS categories. This pattern arises from humans' tendency to avoid extensive comments and longer codes, particularly when confronted with the complexity of challenges in VGD and ADS categories.

❒ **Complexity of Code (LoC):** ChatGPT consistently generates code with higher cyclomatic complexity, particularly in the challenging M, ADS, and VGD categories, showcasing its ability to produce intricate solutions. This is evident in the relatively lower disparity between physical and logical lines of code, highlighting ChatGPT's efficiency in crafting concise codes using advanced Python concepts such as list comprehensions, functional programming, inheritance, polymorphism, generators, iterators, decorators, and sophisticated data structures. Notably, its strength lies in compact code generation across most categories, except for VGD and ADS, where performance was subpar, leading to more incorrect solutions. In comparison, human programmers may avoid these advanced techniques due to perceived complexity. A cross-category analysis reveals ChatGPT's weaknesses in visual, graphical, or drawing problems.

❒ **Halstead Metrics:** Additionally, we evaluated the quality of the generated code using Halstead metrics, including difficulty, effort, volume, and time. Humans showed an anomalously high standard deviation of difficulty, volume, effort, and bugs for the mathematics category, which means some prompts were poorly executed by humans compared to ChatGPT. Overall, the ChatGPT code had lower means for all Halstead metrics than humans, specifically for M, ADS, and VGD categories. This showed that ChatGPT was able to provide efficient code for these categories. However, it

Table 1: Performance of various classification models in predicting ChatGPT code

Model	Class	Precision	Recall	F1	Accuracy
RF	ChatGPT Human	83% 91%	92% 81%	87% 86%	87%
DT	ChatGPT Human	83% 95%	96% 81%	89% 88%	88%
RUSBoost	ChatGPT Human	79% 83%	85% 77%	81% 80%	81%
GNB	ChatGPT Human	51% 67%	96% 8%	67% 14%	52%
MLP	ChatGPT Human	51% 56%	85% 19%	64% 29%	52%
XGB	ChatGPT Human	69% 80%	85% 62%	76% 70%	73%
KNN	ChatGPT Human	61% 75%	85% 46%	71% 57%	65%

also performed poorly in terms of functional accuracy in the VGD and ADS categories.

❒ **Maintainability Index:** Furthermore, ChatGPT's code exhibits enhanced maintainability across all categories from a maintainability and security perspective. This is also characterized by fewer bugs measured by the Halstead Bugs metric and the implementation of high-quality error-handling techniques. These techniques encompass robust *exception handling*, the adoption of Testing and Test-Driven Development (TDD) practices, a strong emphasis on functional programming principles, and proficient memory management strategies.

Conclusion. Comparing code understandability, maintainability, and security between ChatGPT and humans, metric distribution plots reveal that ChatGPT tends to generate code with higher cyclomatic complexity, excelling in crafting concise code using advanced Python concepts. ChatGPT displays proficiency in list comprehension, functional programming, and sophisticated data structures. Evaluation with Halstead metrics consistently indicates that ChatGPT produces higher-quality code, exemplified by lower difficulty and time scores while maintaining lower effort and volume scores compared to human-generated code. From a maintainability and security perspective, ChatGPT exhibits enhanced qualities, showcasing robust error handling, commenting, and debugging output. Despite potential biases in the instructions, the observations emphasize ChatGPT's ability to produce code with advanced constructs and quality practices.

5.3 ChatGPT Code Detection using Machine Learning

This section presents our findings concerning hyperparameter tuning for the seven classification algorithms, their classification performance, and their resilience during reliability testing.

5.3.1 Code Detection Performance.

Table 1 demonstrates the performance of ChatGPT code detection using different machine learning models and across different measures. The Decision Trees (DT) algorithm exhibited exceptional performance with an accuracy of 88%, closely followed by the Random Forest (RF) algorithm, which achieved an accuracy of 87%. Since the dataset is perfectly

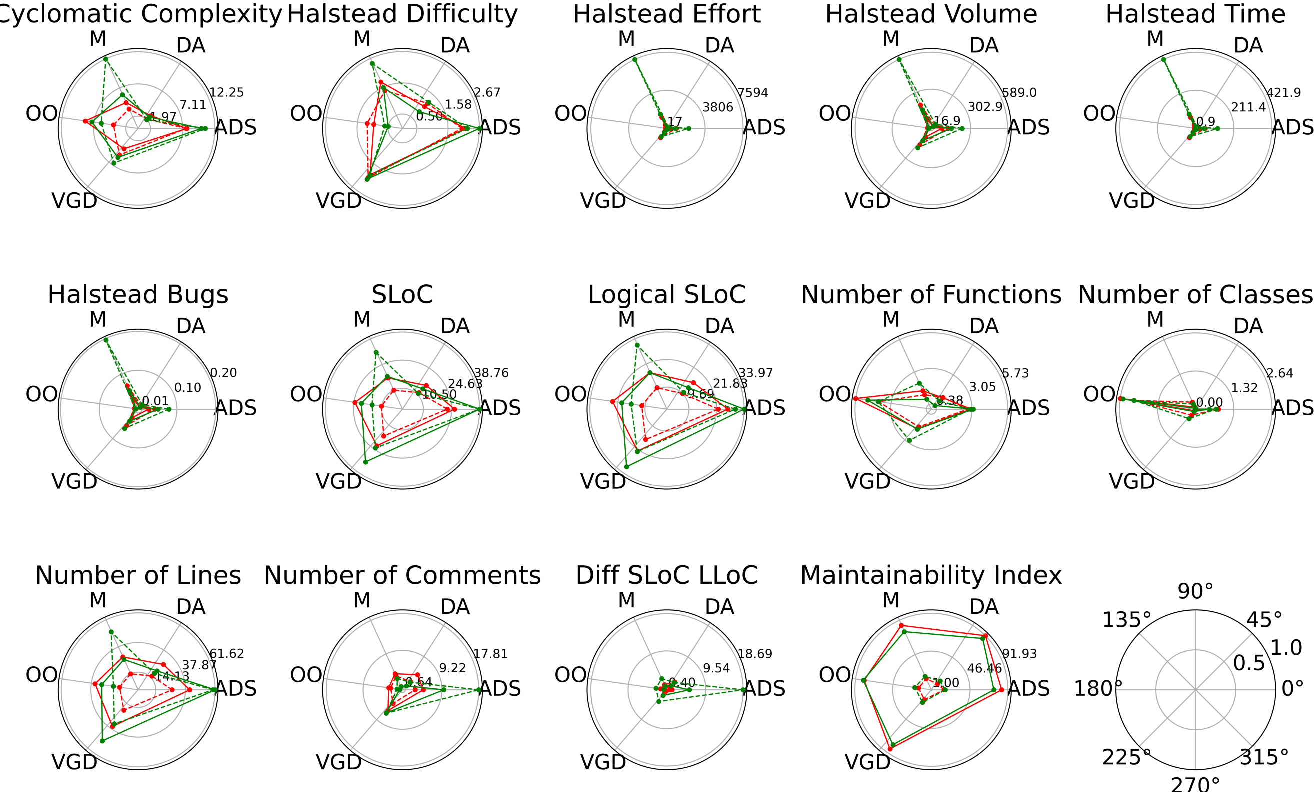


Figure 4: The mean (solid line) and standard deviation (dotted line) for each metric across categories, providing a comprehensive view of ChatGPT (red) and human (green) coding styles.

balanced, the F1 and Weighted F1 scores align closely. It is intriguing that RUSBoost, typically employed for imbalanced datasets, delivered outstanding results with an accuracy of 81%

5.3.2 *Reliability Tests*

The results of the three reliability tests provided insight into the model's generalizability. For this analysis, we only used the top three best-performing algorithms: Random Forest (RF), Decision Trees (DT), and Random UnderSampling and Boosting (RUSBoost). Tests were performed using the best hyperparameter. The following observations were made from each test.

❒ **Train-Test Split Ratio Test.** Our analysis of the impact of varying train-test split ratios on the performance of three models revealed distinct patterns. The RF and RUSBoost models exhibited stability across different splits, with RF peaking at an 87% Weighted F1 Score at an 80% train split, as illustrated in Figure 5. The DT model, however, showed greater variability and inconsistent performance as the train split size increased, which may be due to the DT's tendencies to overfit the data and the effects of random sampling post-train split decision. A notable performance dip for DT between 40-50% train split sizes suggests concerns regarding its generalizability. Remarkably, RUSBoost outperformed the RF classifier at several split ratios, although RF maintained the highest performance at the 80% split. This underscores the efficacy of the conventional 80-20% train-test split for training classifiers. All models displayed a performance decline beyond an 80% train split, likely attributable to the reduced size of the test set, emphasizing the balance needed between training and testing datasets for optimal model evaluation.

❒ **Per-category Performance Test.** In this test, we investigated the comparative performance of three machine learning models—RF, DT, and RUSBoost—across various categories as presented on the x-axis in Figure 6. The RF model consistently delivered strong results, particularly excelling in categories such as "Algo Data Struct," "Mathematics," and "Object Oriented," where it achieved perfect scores in some tests. However, the DT displayed variable effectiveness; it performed well in "Mathematics" but was less reliable in other categories. Notably, it exhibited multiple misclassifications in "Human" codes, demonstrating a tendency to overfit GPT-generated codes. RUSBoost distinguished itself with high performance in the "Mathematics" category; however, it generally performed less effectively across other categories. All models faced challenges in the "Algo Data Struct" and "Visual Graph Draw" categories, where the codes are complicated to distinguish, as evident from the metrics comparison in Figure 4. This analysis underscores the necessity of selecting an appropriate model based on the specific demands of each category. Furthermore, the varied performance of all three models across different categories suggests that an ensemble approach could potentially improve overall performance.

❒ **Gaussian Noise Test.** In assessing model resilience against external perturbations, we introduced Gaussian noise to the

Case Study 5 (Error Handling, Code Commenting, and Debugging Information): In many of the prompts, ChatGPT was seen to perform excellent error handling, commenting, and debugging output, as shown in Listing 1 in Figure 2. In that example, fairly indicative of the differences between many of the scripts, it is easy to see that the error handling code is more comprehensive, there are more comments, and the debugging output is higher quality in the ChatGPT written script than the human one. Focusing on error handling, ChatGPT wrote excellent error handling code, taking into account not just a general case but also the specific *FileNotFoundError* exception as well. The human code was written without any error handling. Also note that if you remove the error handling, comments, and debugging output, the GPT code would be a single line longer than the human code. In this situation, ChatGPT prioritizes error handling and debugging output to make the code more useable and understandable despite not having been specifically instructed to do so. We can also see it outperforming the human code in the number of comments explaining the code. We can note that the debugging output could be considered its own form of comment, further promoting the idea that ChatGPT is proficient at writing code that follows general best practices for writing high-quality and understandable code.

Case Study 6 (Functional Programming): It was seen that ChatGPT used functions at a significantly greater rate than humans. The code in Listing 2 of Figure 2 showcases this use of functions. It is easy to see that the program generated by ChatGPT has 5 different functions, each performing a different task, while the human code has a single function. We can see here that GPT used functions to promote understandability and reusability in its code, significantly contrasting with the human code in this case. This was often the case in most of the ChatGPT codes, while humans tend to code function only when necessary.

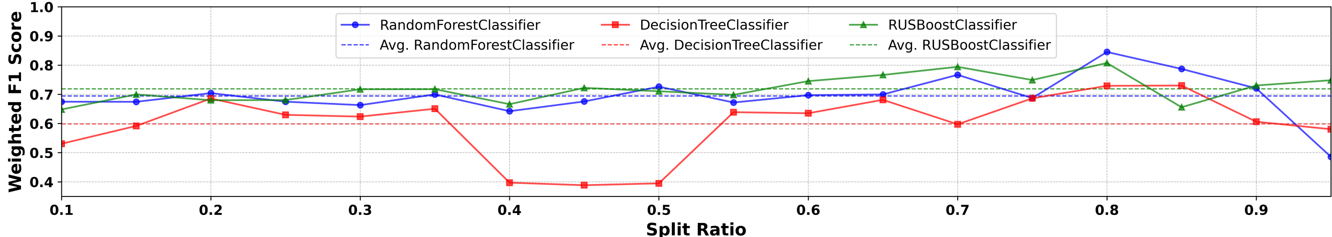


Figure 5: ML model performance on the test set for different train split sizes (x-axis).

test data, incrementally varying the noise's standard deviation from 0 to 1 in 0.01 steps to test stability. The Random Forest (RF) model maintained consistent performance up to a noise standard deviation of 0.5, beyond which it showed sensitivity, as depicted in Figure 7. Despite this, the RF and RUSBoost models exhibited notable noise tolerance, with RF outperforming others in resilience, underscoring its robustness and reliability amidst data perturbations. In contrast, the DT model demonstrated considerable performance fluctuations, highlighting its vulnerability to overfitting and limited adaptability.

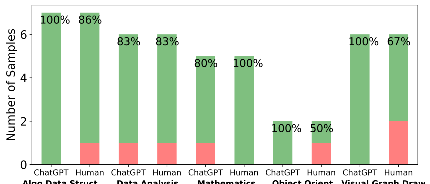
Based on the results of the three tests, we conclude that the Random Forest (RF) algorithm shows robustness in terms of random sampling of changing train set size, addition of random Gaussian noise with varying standard deviation, and cross-category performance across the test set.

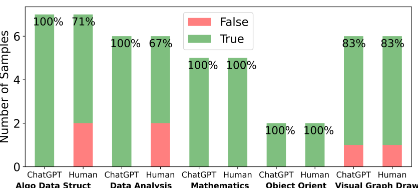
5.4 Feature Analysis

As depicted in Table 2, the *Number of Comments* and *Lines of Code* stand out as pivotal features, echoing the insights discussed in Section 5.1. While these metrics hold importance, particularly for RF models, our exploration delves deeper into other metrics of higher significance. Among these, the *Maintainability Index* proves most paramount in predicting source code authorship. This index encompasses metrics such as *Cyclomatic Complexity, Source Lines of Code*, and *Halstead Volume*. Interestingly, the aggregate influence of the maintainability index surpassed the individual impacts of both *Source Lines of Code* and *Cyclomatic Complexity*. Regarding maintainability, ChatGPT-generated code consistently exhibited superior performance compared to human-written code. This superiority is likely attributable to its lower *Halstead Volume*, reduced count of source lines, and minimized *Cyclomatic Complexity*. On the lower end of the significance spectrum lie metrics such as *Cyclomatic Complexity*, *Number of Classes*, *Difference between Logical and Source Lines of Code*, and *Halstead Difficulty*. Their diminished importance might be a result of their strong correlation with dominant features like *Source Lines of Code (SLoC)* and *Logical SLoC*, making them somewhat

Table 2: Feature Importance of the 14 coding metrics using the dual feature importance approach

Feature	Forest Importance	Perm Importance	Mean Importance
Number of Comments	0.172082	0.298360	0.235221
Number of Lines	0.123837	0.179766	0.151802
Maintainability Index	0.099583	0.178798	0.139190
Number of Functions	0.088670	0.167553	0.128111
Logical SLoC	0.085149	0.122872	0.104010
SLoC	0.084966	0.122422	0.103694
Halstead Time	0.049588	0.073684	0.061636
Halstead Volume	0.048599	0.074152	0.061376
Halstead Bugs	0.048450	0.074172	0.061311
Halstead Effort	0.049134	0.072983	0.061058
Cyclomatic Complexity	0.048372	0.070053	0.059212
Halstead Difficulty	0.045021	0.070368	0.057695
Diff SLoC LLoC	0.045856	0.065013	0.055435
Number of Classes	0.010693	0.014973	0.012833





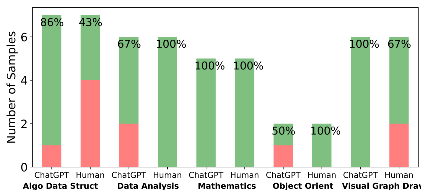


Figure 6: ML model performance for different categories (x-axis) in the test set. From left to right: Random Forest, Decision Tree, RUSBoost.

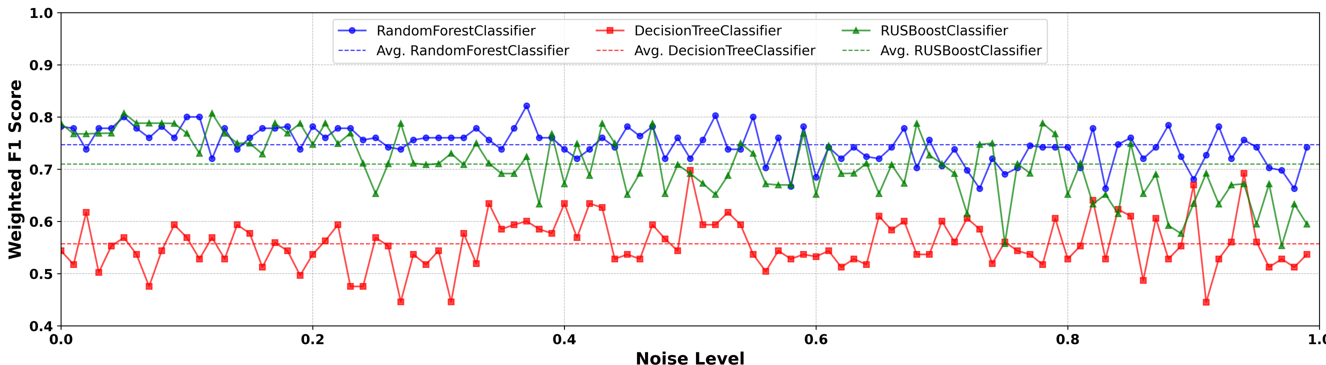


Figure 7: ML model performance on the test set across the different amounts of Gaussian noise (x-axis).

redundant in the eyes of the algorithms, especially when other correlated metrics are present. A noteworthy point pertains to the *Number of Classes* metric, ranking as the least important for determining authorship. The reason was both ChatGPT and human participants adhered to writing an identical number of classes, constrained by the specific requirements set in the prompts. The singular choice of Python as the language and the limitation to a single file size probably swayed this outcome. It suggests potential disparities if the prompts weren't bound to one file or if a language more inclined towards class utilization was chosen.

6 Conclusion

Summary

This paper comprehensively evaluated ChatGPT's code generation capabilities compared to human programmers. We curated a novel dataset of 131 prompts and analyzed 262 code solutions using quantitative metrics and qualitative techniques. Our findings reveal ChatGPT's strengths in efficiently generating concise, modular code with advanced constructs. However, limitations arose in visual-graphical challenges. The comparative analysis highlighted ChatGPT's superior error handling and maintainability while uncovering distinct coding style differences. Additionally, we developed highly accurate machine learning models to classify ChatGPT versus human code.

Scope and Limitations

Our study, while providing valuable insights into ChatGPT's automated code generation capabilities, faces several limitations. These include the small dataset with limited instances per category and the lack of comparison of our proposed model with baseline AI-generated detection systems and human quality measures. Additionally, the temporal fixation on a specific ChatGPT version and the limited scope of Python programming languages may restrict the generalizability of our findings. Despite these constraints, our study underscores the need for future research to address these limitations for a more comprehensive understanding of AI-driven code generation technologies.

Future Work

Our contributions establish a robust foundation to advance AI programming assistants. The curated dataset provides a benchmark for rigorous LLM evaluations. The quantitative and qualitative analyses deliver nuanced insights into current capabilities and limitations. Our ML models pioneer techniques to detect AI-generated code automatically. Several promising directions remain for future work. As LLMs evolve rapidly, prompt engineering techniques should be refined to enhance performance. Testing ChatGPT's abilities across diverse languages and integrating code context could reveal further insights. Evaluating code generation alongside complementary tasks like summarization and documentation may be worthwhile. Investigating neural architecture optimizations and continual learning to bolster visual reasoning could address critical weaknesses. Overall, this work helps chart a course toward trustworthy and human-aligned AI programming assistants.

Acknowledgment

This work is supported by the National Science Foundation (NSF) under grant number EES-2321304.

7 References

[1] Soheila Farokhi, Aswani Yaramal, Jiangtao Huang, Muhammad Fawad Akbar Khan, Xiaojun Qi, and Hamid Karimi. Enhancing the performance of automated grade prediction in mooc using graph representation learning. In *2023 IEEE 10th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10. IEEE, 2023.

[2] Muhammad Fawad Akbar Khan, John Edwards, Paul Bodily, and Hamid Karimi. Deciphering student coding behavior: Interpretable keystroke features and ensemble strategies for grade prediction. In *2023 IEEE International Conference on Big Data (BigData)*, pages 5799–5808. IEEE, 2023.

[3] Muhammad Fawad Akbar Khan, Max Ramsdell, Erik Falor, and Hamid Karimi. Assessing the promise and pitfalls of chatgpt for automated code generation. *arXiv preprint arXiv:2311.02640*, 2023.

[4] Hamid Karimi, Jiangtao Huang, and Tyler Derr. A deep model for predicting online course performance. *Association for the Advancement of Artificial Intelligence*, 2020.

[5] Hamid Karimi, Tyler Derr, Jiangtao Huang, and Jiliang Tang. Online academic course performance prediction using relational graph convolutional neural network. ERIC, 2020.

[6] Hamid Karimi, Tyler Derr, Kaitlin T. Torphy, Kenneth A. Frank, and Jiliang Tang. A roadmap for incorporating online social media in educational research. volume 121, pages 1–24, 2019.

[7] Hamid Karimi, Tyler Derr, Kaitlin Torphy, Kenneth A Frank, and Jiliang Tang. Towards improving sample representativeness of teachers on online social media: A case study on pinterest. pages 130–134. Artificial Intelligence in Education: 21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part II 21, 2020.

[8] Kaitlin Torphy Knake, Hamid Karimi, Sihua Hu, Kenneth A Frank, and Jiliang Tang. Educational research in the twenty-first century: Leveraging big data to explore teachers' professional behavior and educational resources accessed within pinterest. *The Elementary School Journal*, 122(1):86–111, 2021.

[9] Hamid Karimi, Jiliang Tang, Xochitl Weiss, and Jiangtao Huang. Automatic identification of teachers in social media using positive unlabeled learning. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 643–652, 2021.

[10] Sakshi Solanki, Kiana Kheiri, Marissa A Tsugawa, Hamid Karimi, et al. Leveraging social media analytics in engineering education research. In *2023 ASEE Annual Conference & Exposition*, Baltimore , Maryland, June 2023. ASEE Conferences. https://peer.asee.org/43472.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] OpenAI. Chatgpt: A large-scale generative model for conversations, 2023. Accessed: Date you accessed the blog post.

[13] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023.

[14] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[15] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

[16] George C Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83–94, 2000.

[17] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023.

[18] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *arXiv preprint arXiv:2308.04838*, 2023.

[19] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[20] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023.

[21] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[22] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, 2023.

[23] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *arXiv preprint arXiv:2307.12596*, 2023.

[24] LeetCode. Leetcode. https://leetcode.com. Accessed: August 10, 2023.

[25] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[26] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. Improving chatgpt prompt for code generation. *arXiv preprint arXiv:2305.08360*, 2023.

[27] OpenAI. Openai cookbook, 2023. Accessed on Date of access.

[28] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE, 2021.