Thesis Report

ABSTRACT

In recent years, Artificial Intelligence has become more sophisticated, useful, and almost necessary for a wide range of tasks that are performed in many very different aspects of our society. It is improving at such a fast rate that many professions are partially or being replaced by it, and this raises the question: How similar is Artificial Intelligence's work to human work? What are the differences between them? In the field of informatics, these questions have become very interesting as a considerable amount of code is today written or guided by Artificial Intelligence such as ChatGPT, Copilot, or DeepSeek. In the last few months, I have been researching this topic by analyzing different types of code from the GitHub platform, written in these 9 common programming languages: Python, JavaScript, Java, Typescript, C++, Golang, Php, C and C#. By using the "AI code detector" tool from OpenAI, I then compared Artificial Intelligence-written code with human-written code. By analyzing patterns such as indentation, variable naming, and code structure, the detector was able to provide a confidence level about the probability of the code being written by Artificial Intelligence. The main objective of my work is to have a deeper understanding of the simi- varieties and differences between Artificial Intelligence and human-written code and to provide a valuable dataset of code written by Artificial Intelligence. Ode and to provide a valuable dataset of code written by Artificial Intelligence.

1. Introduction

Nowadays, Artificial Intelligence is increasingly important in our everyday tasks, and coding is no exception. Artificial Intelligence like ChatGPT is widely used in this field; it's mainly used to generate code or automate some repetitive tasks and processes, but it can also be used to guide and help during code writing and/or understanding.

This innovation, of course, does not only simply improve the developers' lives but it also comes with some risks. Using AI to generate code may bring bugs or inefficiencies in the code as AI usually works better with shorter and technical tasks rather than with longer chunks of code that require semantic understanding.

Building on this growing presence of AI in software development, it becomes essential to evaluate how AI-generated code compares to code written by human developers. As AI tools are increasingly integrated into professional and educational workflows, understanding their coding patterns, strengths, and limitations is crucial. This thesis aims to explore the extent to which AI-generated code differs from human-written code in terms of structure, clarity, maintainability, and best practices. By analyzing both qualitative and quantitative aspects of code, this study hopes to shed light on the evolving dynamics between human programmers and artificial intelligence and the potential implications for future coding standards, education, and software engineering practices.

1.1. Report Structure

In Section 2: "Related Work," I will explore and critically evaluate existing research that investigates the similarities and differences between human-written and AI-generated code. Several recent studies have attempted to assess the quality, readability, and maintainability of code produced by large language models. Some works focus more on patterns and common programming idioms, while others investigate code correctness and vulnerability rates. Furthermore, I will examine studies from the software engineering and machine learning communities that discuss the implications of AI assistance in professional and educational contexts.

In Section 3: "Study Design," I will dive deeper into the question: "To what extent is AI-generated and manually written code different?" To approach this, I first curated two datasets—one consisting of code written entirely by human developers, and another composed of code generated by AI systems (primarily ChatGPT and Copilot). I focused on commonly used programming languages and ensured comparable code functionality across both sets. I then identified specific dimensions of analysis, including code structure, naming conventions, comment usage, indentation and formatting consistency, function length, and others. The study also considers contextual factors such as code origin (e.g., educational, professional, open source) and task complexity to ensure a fair comparison.

In Section 4: "Results Discussion," I interpret the results of the comparative analysis between human-written and AI-generated code. Key patterns emerged that underscore the distinct tendencies of each source. For instance, AI-generated code tended to use more generic variable and function names, often closely reflecting the prompt given, while human-written code showed more context-aware and meaningful naming conventions. In terms of structure, AI code exhibited consistent indentation and formatting due to the model's training on style-conformant code, whereas human code showed greater stylistic variability. Furthermore, AI-generated code was more likely to include inline comments. However, some AI-generated solutions also demonstrated elegance in handling edge cases, suggesting that, in certain scenarios, the AI mimics best practices learned from large codebases. The discussion also touches on the limitations of AI-generated code, including occasional lack of modularity, redundancy, and superficial understanding of context. Overall, the findings highlight differences that have implications for code review practices, AI tool adoption, and programming pedagogy.

2. Related Work

The paper [1] examines how AI-generated code compares to human-written code in terms of software metrics and bug occurrences. The authors focus on Java solutions to 90 LeetCode problems, equally split across easy, medium, and hard levels. They generated code using GitHub Copilot and selected the top-voted human-written solutions from LeetCode. Static code analysis tools such as SpotBugs and PMD were used to identify bugs, while software metrics were extracted using the Understand tool. Four metrics were prioritized: AvgCountLineCode, AvgCyclomatic, CountLineCodeDecl, and CountLineCodeExe. The study aimed to answer three questions: (1) What relationships exist between metrics and bugs in AI vs. human code? (2) How do code size and complexity differ? (3) How does bug density compare? Two key metric-bug relationships were found in both types of code. The bug `DLS_DEAD_LOCAL_STORE` positively correlated with both `CountLineCodeDecl` and `CountLineCodeExe`. The bug `AvoidLiteralsInIfCondition` correlated with `AvgCyclomatic`.

AI-generated code tended to be longer than human-written code in most cases. Cyclomatic complexity was evenly distributed between both. Bug density was generally lower in AI-generated code, though correctness was often an issue, especially with harder problems. Only 5 of 30 hard problems were correctly solved by AI. In contrast, human-written code was assumed to be correct. The study suggests that AI-generated code may have less errors in syntax but more limited in logic for complex tasks. The authors note that correctness was not directly factored into the bug-metric correlation analysis. They acknowledge threats to validity, such as reliance on LeetCode problems and a single AI tool. The study provides insights into strengths and weaknesses of AI coding and helps inform developers about when and how AI can be trusted in software development.

The authors of the paper "FormAI: A Dataset of 112K Verified Compilable C Programs Generated by GPT-3.5 for Security Analysis" [2]created the FormAI dataset, consisting of 112,000 compilable C programs generated using GPT-3.5-turbo. These programs were written to perform a wide variety of tasks with diverse complexity. Each generated program was designed to be independent and self-contained, without requiring external input or dependencies. To assess security, the team used the ESBMC (Efficient SMT-based Bounded Model Checker), a formal verification tool. ESBMC checks each program for vulnerabilities such as buffer overflows, integer overflows, null pointer dereferencing, and more. The tool labels each vulnerability with type, line number, function name, and CWE (Common Weakness Enumeration) identifier. Over 197,800 vulnerabilities were found in total across the dataset. 51.24% of the programs were found to contain at least one vulnerability. The most common issues were Buffer Overflow, Arithmetic Overflow, Array Bounds Violations, and Dereference Failures. On average, each vulnerable file contained multiple issues, highlighting the compounding risk in AI-generated code. The study shows that even simple prompts to LLMs can produce syntactically correct but unsafe code. A dedicated effort was made to ensure code diversity using prompt combinations from 200 task types and 100 coding styles. About 98% of programs successfully compiled with the GNU C compiler; ESBMC classified 106,138 programs. The dataset enables detailed classification and benchmarking of static and dynamic vulnerability scanners. It can also support training machine learning models in secure code generation and vulnerability prediction. The team discovered and helped fix bugs in ESBMC, Clang, and CBMC using the generated code. The authors conclude that LLMs like GPT-3.5 frequently produce vulnerable code, necessitating caution and improved safeguards when using AI for software development.

The paper "Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues" [3] conducts a thorough empirical study on the reliability and quality of code generated by ChatGPT (GPT-3.5) for programming tasks. The authors created a benchmark of 2,033 LeetCode problems (easy, medium, hard) and used ChatGPT to generate 4,066 code snippets in Python and Java. They first evaluated the correctness of the generated code using LeetCode's test suites. About 66% of Python and 69% of Java solutions passed all tests. However, accuracy declined with increasing task difficulty and for tasks introduced after 2021—likely due to the model's limited training data. They found common code issues, even in correct solutions. Around 47% of the programs had maintainability issues, and 27% had incorrect outputs. Static analysis tools (e.g., Pylint, Flake8, PMD, Checkstyle) revealed issues like unused variables, poor structure, and excessive complexity. Runtime and compilation errors were also cataloged, especially for newer or more complex problems. Results showed that ChatGPT could self-repair 20–60% of the issues, with static feedback being more effective for maintainability bugs and simple prompts performing better for logic or performance issues. However, repairs sometimes introduced new issues, especially when using vague prompts. The paper concludes that while ChatGPT shows potential for code generation, its output often requires human oversight and revision, especially for complex or newer tasks.

The authors of "Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation" [4]evaluates the correctness and quality of code generated by ChatGPT (GPT-3.5), GPT-4, and Google Bard. The authors selected three complex Java programming problems from university exams to ensure they were not part of the LLMs' training data. The generated code was tested using 32 custom test cases and analyzed with SonarCloud to assess metrics like lines of code, cyclomatic complexity, cognitive complexity, and code smells. GPT-4 achieved the best results, passing 29/32 test cases, followed by GPT-3.5 with 28/32, and Bard with 22/32. Bard failed to produce a correct solution for one entire problem. GPT-4 also produced less complex code. None of the solutions showed bugs, vulnerabilities, or duplications. All models required minimal but necessary additional prompting to make the code runnable. The study highlights that while LLMs can generate functional code, human oversight remains essential. GPT-4 showed moderate advantages over the other models in both correctness and quality. The paper concludes that LLMs adhere fairly well to coding standards but cannot yet fully replace human developers.

Instead, in the paper "Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers"[5], the authors investigates the security implications of using generative AI, specifically ChatGPT, compared to traditional developer resources like StackOverflow. The authors selected 108 Java code snippets from SO that addressed security-related questions and then used the same questions to prompt ChatGPT (GPT-3.5-turbo) to generate equivalent code. After compiling the snippets from both sources, they analyzed them using GitHub's CodeQL, a static analysis tool that detects vulnerabilities based on Common Weakness Enumerations (CWEs). The results showed that ChatGPT-generated code contained 248 vulnerabilities, while StackOverflow code had 302, marking a statistically significant 20% reduction in the number of vulnerabilities for ChatGPT. Additionally, ChatGPT produced 19 different CWE types compared to 22 in SO, and the overlap in specific vulnerabilities between the two sources was only 25%. Both platforms had a total of 274 unique vulnerabilities, with the most common being related to weak cryptographic practices and resource management issues. Interestingly, ChatGPT code had more instances of hard-coded credentials, whereas SO code had more issues with pseudo-random number generation. Despite ChatGPT's relatively better performance, the authors emphasized that neither source can be trusted blindly, as both contribute to insecure code. They advocate for the consistent use of secure development practices, such as static analysis tools, to mitigate risks. The study also raises questions about why LLMs like ChatGPT, trained on internet data including SO, might produce less vulnerable code.

3. Study Design

3.1. Data Collection

As the starting point for the practical phase of my research, I explored and evaluated effective methods for identifying code that was likely generated by artificial intelligence. This is an inherently complex and rapidly evolving area of study, primarily because AI models—particularly those designed to assist with programming—are continuously improving in their ability to produce code that closely resembles human-written logic in both structure and style. The boundaries between human-generated and AI-generated code are becoming increasingly blurred, which makes the challenge of distinguishing between the two both technically demanding.

To ground my investigation in real-world data, I turned to GitHub, which serves as one of the largest and most diverse repositories of publicly available source code globally. With millions of developers contributing to open-source and private projects alike, GitHub offers a rich landscape for studying trends in software development and for collecting samples of code that may reflect AI influence.

My initial focus was on developing a reliable strategy for retrieving code samples from GitHub that were highly likely to have been authored, or at least assisted, by AI tools. I reviewed various approaches, such as keyword-based filtering (e.g., searching for commit messages that explicitly mention tools like Copilot or ChatGPT).

This phase of the research required not only technical implementation but also critical judgment in defining what constitutes "AI-generated" code and how confident we can be in that classification. It laid the groundwork for the subsequent stages of the project, where the gathered code samples would be further analyzed using static analysis tools and compared against a control group of human-authored code. By starting with a careful and methodologically sound collection process, I aimed to ensure that the analysis that followed would be based on meaningful, high-quality data.

3.1.1 AI-generated code

To identify code potentially generated with the assistance of AI tools, I began by selecting a set of indicative key phrases, or n-grams, that are commonly used in commit messages referencing AI-assisted coding. These included phrases such as "used ChatGPT to", "Copilot to generate", and similar expressions. The idea was to target commits where developers explicitly mention using AI tools to write code, allowing for a high-precision match with human-reported AI usage.

Using the GitHub API, I developed a mining pipeline that queried the entire public GitHub commits database for matches with these selected n-grams. For every commit whose message contained one of the phrases, I retrieved detailed metadata and stored it in a structured format. Each result was saved as a line in a .jsonl (JSON Lines) file, with each line corresponding to a single commit. For each commit, I stored the following fields:

- Link_to_commit: a direct URL to the commit on GitHub,
- n-gram matched: the specific phrase from my list that triggered the match,
- n_lines_longer_change: the total number of lines added in the commit compared to lines removed,
- n_files_impacted: the number of files modified by the commit,
- longest_added_chunk: the size (in lines) of the longest contiguous block of newly added code.

To assess the effectiveness of the OpenAI AI Code Detector, I selected a random sample of 20 commits from the dataset, each of which included phrases in the commit message suggesting the use of AI tools like ChatGPT or GitHub Copilot. I extracted the largest added code chunk from each commit and submitted it to the detector, which returned a confidence score estimating the likelihood that the code was AI-generated.

The results showed that most commits received high confidence scores, typically ranging from 85% to 98%, confirming that the detector successfully recognized the AI-generated nature of the code in cases where developers had explicitly mentioned using AI tools. The tool also explained the patterns and the metrics that resulted in that specific confidence level. This alignment suggests that, when AI use is disclosed, the detector tends to agree and can provide strong supporting evidence.

However, a few commits received moderate to low scores (e.g., in the range of 10% to 60%). These lower scores may reflect several factors: the code could have been heavily modified by the developer after initial generation, the code style might resemble conventional human-written patterns, or the n-gram match may have captured a false positive (i.e., the commit mentioned ChatGPT but the code was not actually AI-generated).

Overall, these results demonstrate that the AI Code Detector generally aligns well with human-reported AI usage in commit messages, but its confidence can vary depending on the structure and clarity of the code. This highlights the tool's usefulness in combination with metadata-based filtering (like n-gram search), while also pointing to its limitations in ambiguous or edge cases.

3.1.2 Manually written code

To better understand the limitations of the OpenAI AI Code Detector, I conducted a second experiment using a control group of commits that were guaranteed to be human-written. Specifically, I randomly selected 20 commits that were pushed before the year 2020, a period before the widespread availability of tools like ChatGPT or GitHub Copilot. This ensured that the selected code could not have been generated using AI assistance.

I followed the same procedure as in the previous experiment: for each commit, I extracted the largest added code block and submitted it to the AI Code Detector. Surprisingly, the results showed that a significant number of these commits still received high confidence scores, with several exceeding 85%, and a few even reaching into the 90%+ range.

While some commits were correctly classified as low-confidence (e.g., scores as low as 3%, 4%, 12%, and 18%), the presence of so many false positives—human-written code flagged as likely AI-generated—raises important concerns about the reliability of the detector, especially when used in isolation.

This inconsistency suggests that the AI Code Detector can mistake conventional or highly regular coding styles (which are common in many real-world projects) for AI-generated patterns. As a result, it struggles to consistently distinguish between actual AI-generated code and human-written code that simply follows common templates or boilerplate logic.

Due to this limitation, I decided to supplement the detector's output with rule-based code analysis using Semgrep, a static analysis tool that allows for more specific inspection of code structure, patterns, and behavior.

3.2. Detecting code quality issues using Semgrep

Noticing the inherent uncertainty and occasional inconsistency in the results produced by OpenAI's AI code detector, I decided to incorporate an additional layer of analysis using a complementary tool known as Semgrep. Semgrep is a powerful static analysis tool capable of detecting code patterns, vulnerabilities, and enforcing custom rules across various programming languages. I believed that its rule-based approach could provide a more structured and interpretable assessment of the code samples, especially when evaluating their quality, potential security risks, or stylistic anomalies that might hint at AI involvement.

To begin this phase, I first extracted the raw source code from the JSON file, which contained detailed metadata about each retrieved GitHub commit. This file served as the central repository for all previously gathered information, including code snippets, filenames, timestamps, and commit messages. Using a custom Python script, I parsed the relevant data and systematically stored each code snippet in a separate, isolated file. This setup ensured that each file could be individually processed and analyzed without interference from unrelated data.

With the code files prepared, I moved on to defining the Semgrep analysis pipeline in Python. The script was designed to automatically scan each file using a predefined set of Semgrep rules focused on identifying security vulnerabilities, syntactic patterns, and other relevant code metrics. The output from Semgrep was then collected and organized for further inspection. This process allowed for a deeper understanding of the structural and semantic characteristics of the code samples.

The Semgrep analysis pipeline serves as a crucial component in the broader effort to assess the security posture of software projects by statically analyzing code for potentially dangerous patterns, bad practices, and known vulnerability signatures. By scanning through selected commits, the pipeline provides insights into how often developers, regardless of whether they're human or AI, introduce code patterns that violate common security best practices or align with established software weaknesses cataloged by organizations such as MITRE. These checks are particularly valuable in historical analysis, where the goal is not only to catch vulnerabilities but also to understand their frequency, contexts, and potential evolution over time.

The process is designed to be modular and repeatable, supporting a variety of input formats and programming languages, and integrating seamlessly with pre-processed commit data. The flexibility of Semgrep's rule system further enhances the pipeline's utility, allowing custom patterns based on the project's scope. By observing what types of patterns frequently emerge in real-world codebases, it becomes possible to infer how security-conscious developers are in practice, how language features may lend themselves to misuse.

In summary, the Semgrep analysis pipeline works both as a detection tool and as a way to better understand real-world coding practices. The results from this pipeline are not final conclusions, but instead serve as starting points for deeper discussions about code quality, how developers write software, and how automated tools are becoming more important in helping keep code secure.

3.3. Data Analysis

3.4. Replication Package

https://github.com/Loca0307/Thesis

4. Results discussion

5. Conclusions and Future Work

6. References

References

[1] A Comparative Analysis between AI Generated Code and Human Written Code: A Preliminary Study https://www.computer.org/csdl/proceedings-article/bigdata/2024/10825958/23ykKzCP4RO

[2] FormAI: A Dataset of 112K Verified Compilable C Programs Generated by GPT-3.5 for Security Analysis https://arxiv.org/abs/2307.02192

[3] Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues https://arxiv.org/abs/2307.12596

[4] Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation https://www.mdpi.com/1999-5903/16/6/188

[5] Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers https://arxiv.org/abs/2403.15600