Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models

Muntasir Hoq  
North Carolina State University  
United States  
mhoq@ncsu.edu

Yang Shi  
North Carolina State University  
United States  
yshi26@ncsu.edu

Juho Leinonen  
The University of Auckland  
New Zealand  
juho.leinonen@auckland.ac.nz

Damilola Babalola  
North Carolina State University  
United States  
djabalo@ncsu.edu

Collin Lynch  
North Carolina State University  
United States  
cflyynch@ncsu.edu

Thomas Price  
North Carolina State University  
United States  
twprice@ncsu.edu

Bita Akram  
North Carolina State University  
United States  
bakram@ncsu.edu

ABSTRACT

The emergence of publicly accessible large language models (LLMs) such as ChatGPT poses unprecedented risks of new types of plagiarism and cheating where students use LLMs to solve exercises for them. Detecting this behavior will be a necessary component in introductory computer science (CS1) courses, and educators should be well-equipped with detection tools when the need arises. However, ChatGPT generates code non-deterministically, and thus, traditional similarity detectors might not suffice to detect AI-created code. In this work, we explore the affordances of Machine Learning (ML) models for the detection task. We used an openly available dataset of student programs for CS1 assignments and had ChatGPT generate code for the same assignments, and then evaluated the performance of both traditional machine learning models and Abstract Syntax Tree-based (AST-based) deep learning models in detecting ChatGPT code from student code submissions. Our results suggest that both traditional machine learning models and AST-based deep learning models are effective in identifying ChatGPT-generated code with accuracy above 90%. Since the deployment of such models requires ML knowledge and resources that are not always accessible to instructors, we also explore the patterns detected by deep learning models that indicate possible ChatGPT code signatures, which instructors could possibly use to detect LLM-based cheating manually. We also explore whether explicitly asking ChatGPT to impersonate a novice programmer affects the code produced. We further discuss the potential applications of our proposed models for enhancing introductory computer science instruction.

CCS CONCEPTS

- Applied computing → Education.

KEYWORDS

ChatGPT; large language model; artificial intelligence; introductory programming course; CS1; cheat detection; plagiarism detection

ACM Reference Format:

1 INTRODUCTION

Plagiarism is a common problem in introductory programming courses [3]. Previous work has found, for example, that students might resort to plagiarism due to struggling [18] and might be confused about what constitutes plagiarism in programming [6, 27, 28]. In the context of programming, plagiarism can take various forms, such as copying code from the internet (e.g., StackOverflow), sharing solutions between students, and contract cheating.

Recently, a new possible type of plagiarism has emerged: using powerful, LLM-based AI models and tools such as ChatGPT\(^1\) and GitHub Copilot\(^2\) to create solutions to programming exercises. While these tools might help professional programmers develop code more efficiently\(^3\) and can be used by instructors to create educational resources \([10, 48]\), programming educators have raised concerns around potential student over-reliance on these models \([5, 9]\). Students using such models without attributing the created code to the model might be considered a new type of plagiarism. Prior work has found that most introductory programming problems can be successfully solved by state-of-the-art AI models \([8, 15, 43]\) and

\(^1\)https://openai.com/blog/chatgpt

\(^2\)https://github.com/features/copilot

\(^3\)https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/
that this performance is better than the performance of average students [15, 43]. Similar performance has been observed for more complex data structures and algorithms-level exercises [16].

Code plagiarism or over-reliance on AI models can be difficult to detect. Programming plagiarism detection tools that have been traditionally used in introductory programming courses such as MOSS\(^4\) and JPlag [45] are based on comparing student submissions to each other. However, recent LLM models can generate many different solutions non-deterministically, so, as prior work has argued [42], simply including LLM-generated solutions in MOSS or JPlag may not be sufficient to detect this sort of plagiarism. Some recent efforts aim to detect AI-generated content using AI models, hoping to mitigate this challenge [46].

In this work, we study the automatic detection of ChatGPT-generated programs for introductory programming exercises. We use a publicly available dataset of student-written programs in a CS1 course and use ChatGPT to create programs for the same exercises. This is the continuation of our previous study [24], where we tried detecting ChatGPT and student code. In this study, we evaluate different classification methods for identifying code as AI-generated or student-written. We also compare the structure of the student-created programs to those created by ChatGPT to analyze differences between student and ChatGPT code. Finally, we discuss educational applications for our proposed detection system that go beyond cheating detection and can serve as a tool for providing students with formative feedback and timely intervention during times of struggle. Our research questions (RQs) for this work are:

- **RQ1:** How well can ChatGPT-created programs be distinguished from student-created programs in CS1 courses? What are the differences between student and ChatGPT-generated programs?
- **RQ2:** How do prompts given to ChatGPT impact the similarity of its generated code to student code?

2 RELATED WORK

Cheating and plagiarism are more common issues in introductory-level courses than in higher-level courses, i.e., ones attended by graduate students [51]. Studies have shown that cheating is common among struggling students, who might engage in it despite awareness of university policy [50, 51]. This issue is amplified in online learning where there are more students and less direct instruction and supervision [7, 26]. In addition to fairness and integrity, plagiarism is a concern to educators because it can hinder learning opportunities. From a theoretical perspective, using AI or online resources may be seen as a form of help-seeking, in which students use external resources to overcome challenges in their learning [40, 41]. However, this literature distinguishes between adaptive help-seeking, in which students use these resources to further their understanding and support learning, and expedient help-seeking, where learners use help as a means of avoiding engaging with the problem, which typically inhibits learning [47]. While LLMs have the potential to substantially support students during their programming process through offering programming support such as code explanation, worked examples, and feedback [19, 34–36, 48], over-reliance on such tools can inhibit students’ learning of foundational programming skills. Students who cheat, simply put, never learn to do the work.

Some of the most common plagiarism detection tools among educators are Moss\(^5\) and JPlag [45], both of which rely on token similarity between programs. Kechao et al. [31] have proposed a plagiarism tool that uses the CloSpan data mining algorithm to mine comparable code segments, compute program similarities, and generate a plagiarism report. Experimental results showed improved precision and detection efficiency compared to MOSS, providing more detailed information and visualizing comparable code fragments. A more recent plagiarism detection tool uses XGBoost incremental learning algorithm [25], yielding a high accuracy in plagiarism detection for academic and software industry scenarios. However, these models are not equipped to detect LLM-generated code as LLMs generate code through a stochastic process and cannot be used as pre-determined references for similarity detection.

Several studies have proposed methods to detect LLM-generated text. For instance, Mitchell et al. [39] have developed the DetectGPT tool, which uses probability curvature to detect LLM-generated text. However, few studies have focused on distinguishing between human and LLM-generated programs. In this paper, we train deep learning models to identify LLM-generated code and identify their unique structural differences when compared to student programs.

3 METHOD

3.1 Dataset

We use the student-written code from a publicly available dataset obtained from the CodeWorkout\(^5\) platform. The CodeWorkout dataset contains student code from an introductory programming course in Java. The dataset covers 50 programming problems. We use the first 10 problems from the Spring 2019 semester in our experiment. Uncompilable submissions are removed from the dataset as uncompilable code can not be parsed into Abstract Syntax Trees (ASTs), which is required by some of the approaches we compare. Incorrect submissions are also removed from the dataset as we observed during the ChatGPT code generation phase (Section 3.2) that ChatGPT can correctly solve all 10 programming problems. Thus, we train the models to differentiate correct student vs. correct ChatGPT code. The programming problems cover introductory Java programming concepts, such as methods, variable declaration, data types, conditionals, strings, etc. Students in this course were given the problem statement for each assignment with a Java function prototype. The characteristics of the student-written solutions are provided in Table 1.

3.2 ChatGPT-Generated Code

A dataset comprising programming code generated by ChatGPT is created for the purpose of this study. To create this dataset, we present ChatGPT with the problem statements of the first ten problems. We used the GUI\(^6\) to interact with ChatGPT (March 2023) since introductory programming students will most likely use the online ChatGPT GUI, not the GPT API. The ChatGPT prompt consists of: “Solve this problem: [problem statement]. The function prototype is given: [function prototype]”. We include the prototype

\(^{\text{4}}\) https://theory.stanford.edu/~aiken/moss/

\(^{\text{5}}\) https://codeworkout.cs.vt.edu/

\(^{\text{6}}\) https://chat.openai.com/
We use both traditional ML techniques and recent neural methods. We use different ML models to detect the code sources automatically. We discuss this more in Section 5.

The more recent methods include code2vec [4], ASTNN [59], and SANN [23]. Furthermore, as a baseline, we use MOSS to verify if the code is written by a student or generated by ChatGPT. We performed a manual search to tune the hyperparameters of the code2vec, ASTNN, and SANN models due to the time constraints associated with performing a grid search on deep learning models. The dataset is split in a 3:1:1 ratio for training, validating, and testing. We selected the best hyperparameters using the validating set. As a baseline for comparison, we evaluated more traditional ML models, including SVM and XGBoost. For these models, we used TF-IDF [21] to represent each program in our dataset as a numeric vector, where each index of that vector represents the frequency of a specific token in the given program (e.g., for or double), compared to its relative frequency across all programs. We used 10-fold cross-validation within the training dataset to tune the hyperparameters of the traditional ML models. For SVM, the kernel is set to 'poly' from the set {linear, poly, rbf}, C to 10 from the set {0.1, 1, 10}. For XGBoost, we set the value of max_depth to 10 from the set of {3, 6, 10}, gamma to 1 from the set {1, 5, 9}, and n_estimator to 180.

We use different ML models to detect the code sources automatically. We use both traditional ML techniques and recent neural methods to detect student-written code and ChatGPT-generated code. The traditional methods include SVM [13, 22] and XGBoost [22, 25]. The more recent methods include code2vec [4], ASTNN [59], and SANN [23]. Furthermore, as a baseline, we use MOSS to verify if we can detect ChatGPT-generated code with traditional tools.

We evaluated three state-of-the-art ML code classification models based on their ability to classify programming code as either student-written or generated by ChatGPT. code2vec [4] is an attention-based neural network model designed to learn condensed vector representations for programming code. It has found applications in educational contexts, including bug detection, performance prediction, and skill representation [52–56]. The model processes Abstract Syntax Trees (ASTs) of code snippets and transforms paths between leaf nodes into fixed-length vectors using an attention mechanism that assigns weights to code structures and paths according to their significance for the outcome. This approach highlights significant code structures and paths, aiding in model understanding. ASTNN [59] is an AST-based Neural Network for code classification tasks. It takes an AST of a code snippet and generates a structure-based vector representation. It excels in tasks like code correctness prediction, pattern detection, and clone identification [14, 17, 37, 57, 59]. ASTNN captures code structure and is ideal for categorizing code snippets, including distinguishing between student-written and ChatGPT-generated code. SANN [23] is also an AST-based model utilizing optimized subtree extraction, a two-way embedding approach, and an attention mechanism to represent student code in an effective way. It has shown its effectiveness in student code correctness prediction and detecting code patterns and algorithms from student submissions [23].

## 4 EXPERIMENTS

Our experiments are designed to distinguish between student-written and ChatGPT-generated code. To detect the source of a program, we perform a binary classification task, where we identify if a piece of code has been written by a student (class 1) or generated using ChatGPT (class 0). We trained each model with submissions from all 10 problems in our dataset. We use accuracy, precision, recall, and F1-score as the evaluation metrics. Utilizing a variety of evaluation metrics enables a complete understanding of model strengths and weaknesses [23, 49].

As a baseline for comparison, we evaluated more traditional ML models, including SVM and XGBoost. For these models, we used TF-IDF [21] to represent each program in our dataset as a numeric vector, where each index of that vector represents the frequency of a specific token in the given program (e.g., for or double), compared to its relative frequency across all programs. We used 10-fold cross-validation within the training dataset to tune the hyperparameters of the traditional ML models. For SVM, the kernel is set to 'poly' from the set {linear, poly, rbf}, C to 10 from the set {0.1, 1, 10}. For XGBoost, we set the value of max_depth to 10 from the set of {3, 6, 10}, gamma to 1 from the set {1, 5, 9}, and n_estimator to 180.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CodeWorkout</th>
<th>ChatGPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td># programs</td>
<td>3162</td>
<td>3000</td>
</tr>
<tr>
<td># problems</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Class</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>min code length</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>max code length</td>
<td>83</td>
<td>27</td>
</tr>
<tr>
<td>mean code length</td>
<td>17</td>
<td>10</td>
</tr>
</tbody>
</table>

Since it is given to the students with the problem statement in the CodeWorkout platform, and we want ChatGPT to have the same information as students when constructing the solutions. In order to maintain balance in the dataset, we collect 300 ChatGPT-generated solutions for each problem, which are used for comparison with the student code. To generate each instance of a solution to a specific problem, we regenerate the response of ChatGPT to get different solutions. The correctness of ChatGPT-generated code is manually examined to ensure that we only include the correct programs. For the 10 problems, ChatGPT did not generate any incorrect code. The characteristics of the ChatGPT-generated code are provided in Table 1. In the ChatGPT code generation process, we observe that for small and simple introductory programming problem solutions, ChatGPT solutions have fewer variations than student submissions. We discuss this more in Section 5.

### 3.3 Automatically Distinguishing ChatGPT and Student Code

We used different ML models to detect the code sources automatically. We use both traditional ML techniques and recent neural methods to detect student-written code and ChatGPT-generated code. The traditional methods include SVM [13, 22] and XGBoost [22, 25]. The more recent methods include code2vec [4], ASTNN [59], and SANN [23]. Furthermore, as a baseline, we use MOSS to verify if we can detect ChatGPT-generated code with traditional tools.

We evaluated three state-of-the-art ML code classification models based on their ability to classify programming code as either student-written or generated by ChatGPT. code2vec [4] is an attention-based neural network model designed to learn condensed vector representations for programming code. It has found applications in educational contexts, including bug detection, performance prediction, and skill representation [52–56]. The model processes Abstract Syntax Trees (ASTs) of code snippets and transforms paths between leaf nodes into fixed-length vectors using an attention mechanism that assigns weights to code structures and paths according to their significance for the outcome. This approach highlights significant code structures and paths, aiding in model understanding. ASTNN [59] is an AST-based Neural Network for code classification tasks. It takes an AST of a code snippet and generates a structure-based vector representation. It excels in tasks like code correctness prediction, pattern detection, and clone identification [14, 17, 37, 57, 59]. ASTNN captures code structure and is ideal for categorizing code snippets, including distinguishing between student-written and ChatGPT-generated code. SANN [23] is also an AST-based model utilizing optimized subtree extraction, a two-way embedding approach, and an attention mechanism to represent student code in an effective way. It has shown its effectiveness in student code correctness prediction and detecting code patterns and algorithms from student submissions [23].

## 5 RESULTS

### 5.1 Code Source Identification

To investigate how well student-written and ChatGPT-generated code can be distinguished in an introductory programming course (RQ1), we perform a classification task using the SVM, XGBoost, code2vec, ASTNN, and SANN models. In the experiments, we randomly selected 60% of the dataset for the training set and 20% in each of the validation and test sets. The testing results of the experiment are shown in Table 2.

From Table 2, one can see that all ML models can distinguish between student-written and ChatGPT-generated code well. All accuracies and F1 scores are higher than 90%. Table 2 also shows that deep learning models tend to outperform traditional models, with SANN performing the best in terms of accuracy, recall, and F1-score with values of 0.97, 0.97, and 0.97, respectively. This means

https://github.com/c2nes/javalang

---

Table 1: Dataset properties

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CodeWorkout</th>
<th>ChatGPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td># programs</td>
<td>3162</td>
<td>3000</td>
</tr>
<tr>
<td># problems</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Class</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>min code length</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>max code length</td>
<td>83</td>
<td>27</td>
</tr>
<tr>
<td>mean code length</td>
<td>17</td>
<td>10</td>
</tr>
</tbody>
</table>
Table 2: Performance comparison of different models

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>XGBoost</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>code2vec</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>ASTNN</td>
<td>0.92</td>
<td>0.99</td>
<td>0.87</td>
<td>0.92</td>
</tr>
<tr>
<td>SANN</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
</tr>
</tbody>
</table>

it can identify 97% of ChatGPT-generated code (recall) while only falsely signaling a student of using ChatGPT 3% of the time (1 - precision), suggesting the model is likely viable for classroom use. If higher precision is required, the ASTNN has the highest, with a value of 0.99, while still catching 87% of ChatGPT code.

In general, the AST-based models perform better than the traditional ML models in accuracy, precision, recall, and F1-score. However, the performance of the traditional ML models is competitive compared to the AST-based models, though traditional ML models deal with code as textual data, whereas AST-based models try to encode the syntactic and semantic information of the code. This indicates that there are substantial textual differences between student-written and ChatGPT-generated code.

5.1.1 MOSS’s Detection of ChatGPT Code. The current state of practice for plagiarism detection is to use a similarity detection tool like MOSS [12]. Therefore, we used MOSS as a baseline to detect ChatGPT-generated solutions. MOSS is designed to detect the similarity of solutions (i.e., to detect students copying each others’ code). Therefore, to use MOSS to detect ChatGPT-generated solutions, an instructor would need to create a database of ChatGPT-generated solutions and upload them to MOSS, along with student-submitted code. If a ChatGPT-generated solution matches a student-submitted one, it can indicate likely plagiarism. We simulated this by submitting all student- and all ChatGPT-generated solutions to MOSS for each problem (some of which could represent students submitting ChatGPT-generated code). With 300 ChatGPT-generated solutions, MOSS analyzed 300*299/2 = 44850 unique solution pairs for similarity. We set the language to Java and the MOSS similarity score to 20%). This threshold is chosen based on the authors’ experience in integrating MOSS into their CS classrooms.

Our results show that across the 10 problems, a maximum of 350/44850 (< 1%) of the ChatGPT solution pairs had a similarity ratio above the 20% threshold. This suggests that even if an instructor uses a large database of ChatGPT solutions to detect plagiarism with MOSS, the vast majority (99%) of student-submitted ChatGPT-generated solutions would not be detected as similar to others. This shows that current tools like MOSS are insufficient for detecting ChatGPT-generated code.

5.1.2 Exploring Code Structures and Patterns. To understand why the models are effective at differentiating student and ChatGPT-generated code, we analyzed the structural differences between introductory student code and code generated by ChatGPT using the ten programming problems. We randomly selected one instance of student-written code and one instance of ChatGPT-generated code for each problem. An example solution pair for the caughtSpeeding problem is given in Figures 1 and 2.

The ChatGPT-generated code demonstrates a distinct set of patterns. The generated code is much more concise than student submissions, with an average length of 10 lines compared to 17 lines written by students. These code solutions have high code efficiency and rarely any code duplication compared to student code. We found several structural differences in the ChatGPT-generated solutions compared to the student-written code: 1) ChatGPT-generated code frequently employs ternary operators as an alternative to multiple if-else conditions. This practice allows for more concise and streamlined code, contributing to its brevity. 2) Unlike students’ code which often assigns a value to a boolean variable before returning it, ChatGPT code directly returns the expression. This approach eliminates the need for an additional assignment statement, further enhancing code efficiency. 3) Another common pattern of ChatGPT code is to assign complex expressions to variables before using them as conditions in later if-statements, whereas students directly use the expressions as conditions, often duplicating them across multiple if-statements. ChatGPT uses a direct return statement following an if-return statement, omitting the need for an else-return statement that is commonly observed in student code. Generally, ChatGPT-generated code has simplified the control flow and reduced overall code length. These patterns focus on Boolean logic.
and conditions since that was the primary focus of the 10 problems we analyzed. They likely mirror differences between novice and expert code, mostly on which ChatGPT was trained.

In summary, analysis of even a small sample of student- and ChatGPT-written code shows why our ML models were able to differentiate the two accurately: ChatGPT writes code like an efficient, professional programmer, and novices approach programming fundamentally different than experts [58], even at the level of brain activity [33]. These findings were supported by looking further into the attention weights of the code2vec model and observing that the most influential features in detecting student versus ChatGPT code relied on parts of the program that represented these patterns.

### 5.2 Varying ChatGPT Prompts

Based on our results, ChatGPT-generated code looked very different from student code. Thus, our next question was whether students could complicate the detection process by prompting ChatGPT to impersonate novice students (RQ2). To examine this question, we conducted a pilot study to explore how and to what degree student codes are similar to ChatGPT when prompted to mimic novice programmers. We developed a small dataset that contains 20 programs per four categories of prompts for each problem designed to mimic novice programmer code (as suggested by [32]). The prompts used in this study include i) “Act as a novice programmer” (We will denote this prompt as “impersonate” in the rest of the paper for brevity), ii) “Write the code as a novice programmer” (“roleplay: novice”), iii) “Avoid complications while writing the code” (“avoid complication”), and iv) “Write it as an introductory programming student” (“roleplay: introductory”) along with the same problem descriptions used before. For this pilot study, we analyzed the number of changes in each pair of code from different sources using the edit distance calculated with the Levenshtein algorithm to learn the difference between each type of generated code and use the distance to represent the difference, where a smaller difference means similar output with different prompts [22]. To calculate the edit distance, we strip off all the code comments as

<table>
<thead>
<tr>
<th>Pairs</th>
<th>Edit distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student-Student</td>
<td>138.64</td>
</tr>
<tr>
<td>Student-ChatGPT</td>
<td>136.20</td>
</tr>
<tr>
<td>Student-Impersonate</td>
<td>115.30</td>
</tr>
<tr>
<td>Student-Roleplay: novice</td>
<td>114.40</td>
</tr>
<tr>
<td>Student-Avoid complications</td>
<td>134.60</td>
</tr>
<tr>
<td>Student-Roleplay: introductory</td>
<td>114.30</td>
</tr>
<tr>
<td>ChatGPT-ChatGPT</td>
<td>88.30</td>
</tr>
</tbody>
</table>

An average edit distance of 138.64 in the student-student code shows that novice programmers may follow different unique solution paths during problem-solving. Conversely, the ChatGPT code pairs (using the original prompt without any prompt variation) exhibit less variation in edit distances with a lower average (88.30). This aligns with our earlier observations where, for simple and small introductory programming problems involving basic concepts, ChatGPT demonstrates reduced variations and produces technically sound, optimized, and expert-like programs. Similarly, we observe a higher average edit distance between student code and ChatGPT code using different prompts, including ChatGPT (using the original prompt without any prompt variation), impersonate, roleplay: novice, avoid complications, and roleplay: introductory pairs. Among these, student-ChatGPT and student-avoid complications show the highest variation in programs, meaning higher variation between student code and the ChatGPT-generated code from different prompts, including the original one.

We further investigated the prompts, including “impersonate”, “roleplay: novice”, and “roleplay: introductory” as they show less variation than the previously mentioned ones. In analyzing these three prompts, notable differences emerge in the code based on the complexity of the problems. Larger problems, defined by higher line numbers (>10), exhibit considerable variation in student-written programs and ChatGPT-generated solutions. With prompt variations, ChatGPT produces different solutions compared to the original prompt (10% lower edit distance than student-ChatGPT on average). Nevertheless, these programs remain more optimized and compact compared to student code, especially in the case of return statements, avoiding unnecessary else statements and eliminating unreachable else statements in conditional statements, resulting in a more expert-like programming style. In contrast, novice programmers tend to demonstrate unoptimized programming practices, which experts and ML detectors may identify by observing these patterns in the program structures.

The scenario changes for smaller problems with fewer lines of code (<10). The solution space for these problems is smaller, leading to fewer possibilities of variation and, consequently, a lower average edit distance (20% lower edit distance than student-ChatGPT on average). Detecting differences becomes more challenging in such cases, particularly when the code size is very small and ChatGPT solutions and student programs may share similar patterns. However, novice programs still exhibit distinct novice traits that experts and educators can easily recognize, such as placing values before variable names in expressions (e.g., 60<=temp), using multiple if conditions instead of else-if conditions, and other patterns mentioned earlier. These traits can help differentiate novice-written code from ChatGPT-generated solutions.

In short, larger problems show more variation in student and ChatGPT solutions with or without prompt variation, with ChatGPT producing optimized and compact code. In smaller problems, where solution space and variation in code are limited, distinguishing between ChatGPT and student solutions can be challenging, but novice traits in student code are still likely identifiable by educators.
6 DISCUSSION
The emergence of advanced AI tools brings ample opportunities for CS education. However, it can also cause adverse effects, especially in introductory programming classrooms, as students might use these tools to generate homework solutions without understanding the generated code. In this paper, we conduct an exploratory study to show that there are patterns in ChatGPT-generated code that both machine learning models and computing education practitioners may be able to use for detecting AI-generated codes. Based on our results, ML-based methods can detect ChatGPT-generated code submissions for a set of relatively simple problems with high accuracy (97%). The ChatGPT code submissions share patterns that human instructors could identify, such as using more advanced programming constructs than the typical student would use. Our results demonstrated that code generated by directly prompting ChatGPT for a solution can be effortlessly detected by 1) data-driven code analysis approaches and 2) instructor inspections. However, most of the features that can be used to detect ChatGPT-generated code rely on that code using advanced concepts, so the accuracy in detecting LLM code might be lower in more advanced courses where students are expected to write more optimized code.

We further explored situations where code is generated with different prompts inspired by [32]. We found that even if students attempt to make simple modifications to the prompts for ChatGPT to generate code that mimics novice programming code, the code generated is still distinguishable from students’ own written code. For example, when we tried to add the sentence “Write the code as a novice programmer”, or “Act as a novice programmer while programming”, etc., the generated code is still structurally different from actual students’ programming code (e.g., the ChatGPT-generated code still uses ternary operators). While more systematic experiments are required to validate these findings, the preliminary results suggest that our findings on detecting the AI-generated code remain promising across different prompts.

Teaching and Learning Implications: Identifying code generated by ChatGPT offers various advantages for CS Education. For instance, it enables instructors to intervene when the utilization of generative AI programming tools is deemed to hinder learning. Data-driven plagiarism detection methods have a limitation: the evidence that can be presented to students showing that they committed cheating is not as compelling due to the variations in ChatGPT-generated codes. However, cheating detection does not necessarily need to be the sole end goal for data-driven detection tools. In fact, the alarm systems can serve as formative feedback systems [11, 29]. For example, a highly accurate detection system can be integrated into students’ submission system to prevent students from submitting possibly AI-generated source code until they have made substantial changes that ensure a proper understanding of the code being submitted. Alternatively, the system can prompt the student to reflect on the submitted code to assess their code comprehension and code generation skills. Moreover, since possible cheating students may face challenges in learning [18], the system could also serve as a detector of learning difficulty, and students who trigger the alarms frequently could be targeted by possible support interventions to learn related concepts.

Pedagogical Implications: Generative AI as a tool will expand in the foreseeable future. Schools and teachers should teach students how to use AI ethically and efficiently. AI-based tools are double-edged swords: they can be over-relied on but might help students learn concepts [1, 2, 20]. It remains an open question when and how [38] students should seek help from AI-driven tools.

Limitations: One main limitation of this study is that we used a small subset of the original CodeWorkout dataset for ChatGPT-generated code (10 problems). They mainly focus on the usage of conditionals and are relatively simple and straightforward, and are limited in the variety of possible code structures. Future work should conduct an evaluation of more complex problems, such as ones involving a combination of loops or array structures. In addition, one assumption in our current research is that novice students have access to and know how to use ChatGPT to generate code. While there is little existing research (see e.g. [30, 44] for some preliminary results) systematically investigating how students interact with generative AI tools, such as ChatGPT, students may not know how to manipulate the prompts or do not have the ability to work with such tools. Moreover, what type of use of generative AI constitutes plagiarism is an open discussion. For example, most instructors likely consider the case presented in this paper as plagiarism, where students would directly query ChatGPT to provide an answer to a programming exercise. The situation is more complex if students use ChatGPT to receive intermittent help while conducting problem-solving, however. For example, ChatGPT can be used to debug code and to explain code [34–36], and it is unclear if these use cases should be disallowed. Finally, this work only includes cheat detection models that have not yet been incorporated into live classes, which could be a future research direction.

7 CONCLUSION
In this paper, we introduced an automated ML method to detect code generated by ChatGPT in an introductory programming course. Our results suggest that machine learning methods are able to detect such AI code with a high performance (97% accuracy by SANN) in our dataset, which marks an important step towards exposing ChatGPT-generated code in CS1 courses. We further suggested patterns commonly occurring in ChatGPT-generated code that instructors can identify. We also found that possible variations in prompts will not cause large changes to these patterns. We further discussed possible applications of code source detection tools to improve introductory computer science education.

REFERENCES