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
djbabalo@ncsu.edu

Collin Lynch
North Carolina State University
United States
cflynch@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.

SIGCSE 2024, March 20–23, 2024, Portland, OR, USA

ACM ISBN 979-8-4007-0423-9/24/03…$15.00
https://doi.org/10.1145/3626252.3630826

CCS CONCEPTS

• **Applied computing → Education**.

KEYWORDS

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

ACM Reference Format:
Muntasir Hoq, Yang Shi, Juho Leinonen, Damilola Babalola, Collin Lynch, Thomas Price, and Bita Akram. 2024. Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630826

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-created 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[4] 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

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

[5] https://codeworkout.cs.vt.edu/

[6] https://chat.openai.com/

Table 1: Dataset properties

Dataset	CodeWorkout	ChatGPT
Language	Java	Java
# programs	3162	3000
# problems	10	10
Class	1	0
min code length	4	3
max code length	83	27
mean code length	17	10

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 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 task 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.

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 dataset, while the results were reported on the testing dataset. We set the embedding size to 128, 128, and 256 from a set of {64, 128, 256} for code2vec, ASTNN, and SANN, respectively. The maximum epoch is set to 200 with a patience of 50 to prevent overfitting. To generate the ASTs from the programming code, an open-source tool called javalang[7] is used. javalang provides a lexer and a parser for the Java programming language.

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

[7] https://github.com/c2nes/javalang

Table 2: Performance comparison of different models

Model	Accuracy	Precision	Recall	F1-score
SVM	0.90	0.90	0.90	0.90
XGBoost	0.91	0.91	0.91	0.91
code2vec	0.95	0.95	0.95	0.95
ASTNN	0.92	**0.99**	0.87	0.92
SANN	**0.97**	0.97	**0.97**	**0.97**

it can identify 97% of ChatGTP-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%[8]. 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.

```java
public int caughtspeeding(int speed, boolean isBirthday) {
    if(isBirthday) {
        if(speed <= 65) {
            return 0;
        }
        else if(66 <= speed && speed <= 85) {
            return 1;
        }
        else {
            return 2;
        }
    }
    else {
        if(speed <= 60) {
            return 0;
        }
        else if(61 <= speed && speed <= 80) {
            return 1;
        }
        else {
            return 2;
        }
    }
}
```

Figure 1: Student-written solution for caughtSpeeding

```java
public int caughtSpeeding(int speed, boolean isBirthday) {
    int limit1 = isBirthday ? 65 : 60;
    int limit2 = isBirthday ? 85 : 80;
    if (speed <= limit1) {
        return 0;
    } else if (speed <= limit2) {
        return 1;
    } else {
        return 2;
    }
}
```

Figure 2: ChatGPT-generated solution for caughtSpeeding

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

[8]We used a similar threshold for the student-generated code and found it was sufficient to detect cheating in up to 500 pairs per problem

Table 3: Average edit distance among various sources

Pairs	Edit distance
Student-Student	138.64
Student-ChatGPT	136.20
Student-Impersonate	115.30
Student-Roleplay:novice	114.40
Student-Avoid complications	134.60
Student-Roleplay:introductory	114.30
ChatGPT-ChatGPT	88.30

and conditionals 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 they do not play a meaningful role in the program structure.

Table 3 presents the average edit distance observed among programs generated from variants of prompts and student code. We randomly sampled 20 programs for each problem (from 300 programs per problem for both student and ChatGPT code) and calculated average edit distances over all 10 problems between all pairs of student and ChatGPT programs, including the programs obtained from the prompt variations.

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 programming 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

[1] Bita Akram, Wookhee Min, Eric Wiebe, Bradford Mott, Kristy Elizabeth Boyer, and James Lester. 2018. Improving stealth assessment in game-based learning with LSTM-based analytics. In *EDM*. 208–218.
[2] Bita Akram, Wookhe Min, Eric Wiebe, Anam Navied, Bradford Mott, Kristy Elizabeth Boyer, James Lester, et al. 2020. Automated assessment of computer science competencies from student programs with gaussian process regression. In *EDM*.
[3] Ibrahim Albluwi. 2019. Plagiarism in programming assessments: a systematic review. *TOCE* 20, 1 (2019), 1–28.
[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *POPL* 3 (2019), 1–29.
[5] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard-Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *SIGCSE*. 500–506.
[6] Georgina Cosma and Mike Joy. 2008. Towards a definition of source-code plagiarism. *IEEE Trans. on Ed.* 51, 2 (2008), 195–200.

[7] Seife Dendir and R Stockton Maxwell. 2020. Cheating in online courses: Evidence from online proctoring. *Computers in Human Behavior Reports* 2 (2020), 100033.
[8] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. In *SIGCSE*. 1136–1142.
[9] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. 2023. Computing Education in the Era of Generative AI. *arXiv preprint arXiv:2306.02608* (2023).
[10] Paul Denny, Sami Sarsa, Arto Hellas, and Juho Leinonen. 2022. Robosourcing Educational Resources–Leveraging Large Language Models for Learnersourcing. *arXiv preprint arXiv:2211.04715* (2022).
[11] Martin Dick, Judy Sheard, Cathy Bareiss, Janet Carter, Donald Joyce, Trevor Harding, and Cary Laxer. 2002. Addressing student cheating: definitions and solutions. *SIGCSE* 35, 2 (2002), 172–184.
[12] Steve Engels, Vivek Lakshmanan, and Michelle Craig. 2007. Plagiarism detection using feature-based neural networks. In *SIGCSE*. 34–38.
[13] Akhil Eppa and Anirudh Murali. 2022. Source Code Plagiarism Detection: A Machine Intelligence Approach. In *ICAECC*. 1–7.
[14] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *SIGSOFT*. 516–527.
[15] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *ACE*. 10–19.
[16] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. 2023. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *ACEC*. 97–104.
[17] Manuel A Fokam and Ritesh Ajoodha. 2021. Influence of Contrastive Learning on Source Code Plagiarism Detection through Recursive Neural Networks. In *IMITEC*. 1–6.
[18] Arto Hellas, Juho Leinonen, and Petri Ihantola. 2017. Plagiarism in take-home exams: help-seeking, collaboration, and systematic cheating. In *ITiCSE*. 238–243.
[19] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. In *ICER*.
[20] Kenneth Holstein, Bruce M McLaren, and Vincent Aleven. 2018. Student learning benefits of a mixed-reality teacher awareness tool in AI-enhanced classrooms. In *AIED*. 154–168.
[21] Muntasir Hoq, Peter Brusilovsky, and Bita Akram. 2022. SANN: A Subtree-based Attention Neural Network Model for Student Success Prediction Through Source Code Analysis. In *6th CSEDM Workshop*.
[22] Muntasir Hoq, Peter Brusilovsky, and Bita Akram. 2023. Analysis of an Explainable Student Performance Prediction Model in an Introductory Programming Course. In *EDM*. 79–90.
[23] Muntasir Hoq, Sushanth Reddy Chilla, Melika Ahmadi Ranjbar, Peter Brusilovsky, and Bita Akram. 2023. SANN: Programming Code Representation Using Attention Neural Network with Optimized Subtree Extraction. In *CIKM*. 783–792.
[24] Muntasir Hoq, Yang Shi, Juho Leinonen, Damilola Babalola, Collin Lynch, and Bita Akram. 2023. Detecting ChatGPT-Generated Code in a CS1 Course. In *AIED LLM Workshop*.
[25] Qiubo Huang, Guozheng Fang, and Keyuan Jiang. 2019. An Approach of Suspected Code Plagiarism Detection Based on XGBoost Incremental Learning. In *CNCI*.
[26] Meena Jha, Sander JJ Leemans, Regina Berretta, Ayse Aysin Bilgin, Lakmali Jayarathna, and Judy Sheard. 2022. Online Assessment and COVID: Opportunities and Challenges. In *ACEC*. 27–35.
[27] Mike Joy, Georgina Cosma, Jane Yin-Kim Yau, and Jane Sinclair. 2010. Source code plagiarism—a student perspective. *IEEE Trans. on Ed.* 54, 1 (2010), 125–132.
[28] MS Joy, JE Sinclair, Russell Boyatt, JY-K Yau, and Georgina Cosma. 2013. Student perspectives on source-code plagiarism. *Int. J. for Educational Integrity* (2013).
[29] Oscar Karnalim, Simon, William Chivers, and Billy Susanto Panca. 2022. Educating students about programming plagiarism and collusion via formative feedback. *TOCE* 22, 3 (2022), 1–31.
[30] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *CHI*. 1–23.
[31] Wang Kechao, Wang Tiantian, Zong Mingkui, Wang Zhifei, and Ren Xiangmin. 2012. Detection of plagiarism in students' programs using a data mining algorithm. In *2nd Int. Conf. on Comp. Sc. and Network Tech.* 1318–1321.
[32] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *NeurIPS*.
[33] SeolHwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and HeuiSeok Lim. 2016. Comparing programming language comprehension between novice and expert programmers using eeg analysis. In *BIBE*. 350–355.
[34] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing code explanations created by students and large language models. In *ITiCSE*.
[35] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *SIGCSE*. 563–569.
[36] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *SIGCSE*. 931–937.
[37] Ye Mao, Yang Shi, Samiha Marwan, Thomas W Price, Tiffany Barnes, and Min Chi. 2021. Knowing both when and where: Temporal-ASTNN for Early Prediction of Student Success in Novice Programming Tasks. In *EDM*.
[38] Samiha Marwan, Joseph Jay Williams, and Thomas Price. 2019. An evaluation of the impact of automated programming hints on performance and learning. In *ICER*. 61–70.
[39] Eric Mitchell, Yoonho Lee, Alexander Khazatsky, Christopher D Manning, and Chelsea Finn. 2023. DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature. *arXiv preprint arXiv:2301.11305* (2023).
[40] Sharon Nelson-Le Gall. 1981. Help-seeking: An understudied problem-solving skill in children. *Developmental Review* 1, 3 (1981), 224–246.
[41] Richard S Newman. 2002. How self-regulated learners cope with academic difficulty: The role of adaptive help seeking. *Theory into practice* 41, 2 (2002), 132–138.
[42] Michael Sheinman Orenstrakh, Oscar Karnalim, Carlos Anibal Suarez, and Michael Liut. 2023. Detecting LLM-Generated Text in Computing Education: A Comparative Study for ChatGPT Cases. *arXiv preprint arXiv:2307.07411* (2023).
[43] James Prather, Paul Denny, Juho Leinonen, Brett A Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N Reeves, and Jaromir Savelka. 2023. The Robots Are Here: The Generative AI Revolution in Computing Education. *Working Group Reports on Innovation and Technology in Computer Science Education* (2023).
[44] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *TOCHI* (2023).
[45] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. 2002. Finding plagiarisms among a set of programs with JPlag. *J. Univ. Comput. Sci.* 8, 11 (2002), 1016.
[46] Greg Rosalsky and Emma Peaslee. 2023. This 22-year-old is trying to save us from ChatGPT before it changes writing forever. *NPR* 18 (2023).
[47] Allison M Ryan and Sungok Serena Shim. 2012. Changes in help seeking from peers during early adolescence: Associations with changes in achievement and perceptions of teachers. *J. of Educational Psychology* 104, 4 (2012), 1122.
[48] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *ICER*. 27–43.
[49] Sami Sarsa, Juho Leinonen, and Arto Hellas. 2022. Empirical Evaluation of Deep Learning Models for Knowledge Tracing: Of Hyperparameters and Metrics on Performance and Replicability. *J. of EDM* 14, 2 (2022).
[50] Judy Sheard, Martin Dick, Selby Markham, Ian Macdonald, and Meaghan Walsh. 2002. Cheating and plagiarism: Perceptions and practices of first year IT students. In *ITiCSE*. 183–187.
[51] Judy Sheard, Selby Markham, and Martin Dick. 2003. Investigating differences in cheating behaviours of IT undergraduate and graduate students: The maturity and motivation factors. *Higher Ed. Research & Development* 22 (2003), 91–108.
[52] Yang Shi. 2023. Interpretable Code-Informed Learning Analytics for CS Education. In *LAK*. 180–187.
[53] Yang Shi, Min Chi, Tiffany Barnes, and Thomas Price. 2022. Code-DKT: A Code-based Knowledge Tracing Model for Programming Tasks. In *EDM*. 50–61.
[54] Yang Shi, Ye Mao, Tiffany Barnes, Min Chi, and Thomas W Price. 2021. More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code.. In *EDM*. 446–453.
[55] Yang Shi, Robin Schmucker, Min Chi, Tiffany Barnes, and Thomas Price. 2023. KC-Finder: Automated Knowledge Component Discovery for Programming Problems.. In *EDM*.
[56] Yang Shi, Krupal Shah, Wengran Wang, Samiha Marwan, Poorvaja Penmetsa, and Thomas Price. 2021. Toward semi-automatic misconception discovery using code embeddings. In *LAK*. 606–612.
[57] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *SANER*.
[58] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. 1993. Characteristics of the mental representations of novice and expert programmers: an empirical study. *Int. J. of Man-Machine Studies* 39, 5 (1993), 793–812.
[59] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*. 783–794.