Classifying Code as Human Authored or GPT-4 Generated

by

Oseremen Joy Idialu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some parts of this thesis (Chapters 1-4, 6-8) come from a multi-authored paper of which I am the first author. Pending its publication by Association for Computing Machinery (ACM), this multi-authored paper is publicly available as a preprint, cited below:

> Idialu, O. J., Mathews, N. S., Maipradit, R., Atlee, J. M., & Nagappan, M. (2024). Whodunit: Classifying Code as Human Authored or GPT-4 Generated–A case study on CodeChef problems. arXiv preprint arXiv:2403.04013.

Abstract

Artificial intelligence (AI) assistants such as GitHub Copilot and ChatGPT, built on large language models like GPT-4, are revolutionizing how programming tasks are performed, raising questions about whether generative AI models author code. Such questions are of particular interest to educators, who worry that these tools enable a new form of academic dishonesty, in which students submit AI-generated code as their work. Our research explores the viability of using code stylometry and machine learning to distinguish between GPT-4 generated and human-authored code and attempts to explain the predictions.

Our study comprises two analyses, each based on different datasets, one sourced from CodeChef and the other from an introductory programming course. Both datasets encompass human-authored solutions alongside AI-authored solutions generated by GPT-4. The human-authored solutions selected are from before 2021 to ensure that the solutions were not contaminated with contributions from an AI coding assistant. The initial analysis serves to establish the potential of our approach, while the subsequent analysis extends our approach to actual programming assignments.

In our first analysis, our classifier outperforms the baselines, achieving an F1-score and AUC-ROC score of 0.91. Even a variant of our classifier, which excludes gameable features (features susceptible to manipulation e.g., empty lines, whitespace), maintains a good performance, achieving an F1-score and AUC-ROC score of 0.89. We also conducted an evaluation based on the difficulty level of programming problems, revealing little to no differences across the difficulty levels. Specifically, the F1-score and AUC-ROC remained consistent with scores of 0.89 for easy and medium problems and a slight decrease to 0.87 for harder problems. These results highlight the promise of our approach regardless of the complexity of the programming tasks.

In our second analysis, our classifier, trained and evaluated on programming assignments achieved an F1-score of 0.69 and an AUC-ROC of 0.73. A subsequent evaluation trained and evaluated our classifier on assignments submitted in 2023, a period after the release of Copilot and ChatGPT; we identified 13 out of 54 submissions as GPT-4 generated with an accuracy rate of 73%. We believe educators should recognize and proactively address this emerging trend within academic settings.

Acknowledgements

I would like to thank my supervisors, Meiyappan Nagappan and Joanne Atlee. I would also like to thank members of the SWAG and WATFORM research groups, and others who provided advice and support towards the completion of this work.

As a member of the University of Waterloo, I acknowledge that this work took place on the traditional territory of the Neutral, Anishinaabe and Haudenosaunee peoples.

Dedication

This work is dedicated to my family and friends.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

AI tools like Github Copilot [24], ChatGPT [43], and Code Whisperer [47] are disrupting how educators teach and assess programming. These tools are promoted as "coding assistants" that aim to improve developer productivity by suggesting code snippets, bug fixes, code refactorings, and test cases. The use of AI assistants in introductory programming courses has been shown to increase the productivity of novice programmers in solving introductory programming problems, both in terms of improving the quality of their programs and easing the cognitive load and development effort required [33].

Eventually, however, educators need to assess how well students can program *without* the aid of their coding assistants, and they are worried about academic dishonesty. Programming courses already suffer from high levels of plagiarism [3] and contract cheating [11]. The ease with which these new tools can automatically generate code raises concerns about a new form of academic dishonesty, in which students submit AI-generated programs as their work [5, 22, 37]. Existing approaches to detecting plagiarism among student-submitted programs rely on automated similarity comparison tools, but these tools are unlikely to detect AI-generated solutions because AI-generated code has low similarity to student-authored code [46].

Consider the following real-world example of code-style differences between human and GPT-4 generated code to motivate our work. Figure 1.1 presents two Python programs, both of which compute the sum of all palindromic numbers within an input range of integers[1]: Figure 1.1a represents a user submission obtained from CodeChef, whereas Figure 1.1b was generated by ChatGPT. The two solutions exhibit clear differences in coding styles. The AI-generated code includes empty lines, and helper functions, whereas

[1] https://www.codechef.com/problems/SPALNUM

```python
for _ in range(int(input())):
l, r = map(int, input().split())
result = 0
for i in range(l, r + 1):
    if str(i) == str(i)[::-1]:
        result += i
print(result)
```

(a) Example of human code

```python
def is_palindrome(n):
    return str(n) == str(n)[::-1]

def palindromic_numbers_sum(l, r):
    total = 0
    for n in range(l, r+1):
        if is_palindrome(n):
            total += n
    return total

t = int(input())

for i in range(t):
    l, r = map(int, input().split())
    result = palindromic_numbers_sum(l, r)
    print(result)
```

(b) Example of ChatGPT code

Figure 1.1: Solutions to a palindrome problem from CodeChef

the human-authored code uses shorter identifiers. These differences and other code-style patterns (e.g., frequency of different keywords, complexity of expressions) used in prior studies on author attribution led us to question whether we could use code-stylometry features to build a classifier that distinguishes between human-authored code and GPT-4 generated code.

The goal of our research is to determine the viability of constructing a classifier that can distinguish between GPT-4 generated and human-authored code. We hypothesize that the low similarity between student-authored and GPT-4 generated code suggests that code stylometry and machine-learning classification can be used to distinguish between the two.

1.1 Contributions

In this thesis, we make the following contributions: (1) We make a best-effort attempt at constructing a classifier for detecting GPT-4 generated Python code, using a combination of supervised machine learning (XGBoost [10]), and a collection of 140 code-stylometry features. The classifier is trained and evaluated on a dataset comprising 798 human-authored solutions and 798 GPT-4 generated solutions to 399 Python problems of varying degrees of difficulty from CodeChef.[2] To our knowledge, this is the first attempt to construct a classifier for detecting GPT-4 generated code based on a dataset with over 1000 solutions. (2) Our evaluation focuses on four research questions:

RQ1: How well can code-stylometry features distinguish human-authored code from GPT-4 generated code?

Prior work has shown that a classifier trained on code-stylometry features can distinguish among different human developers. We conjecture that a classifier can be built to differentiate code developed by humans from code generated by AI tools.

RQ2: How influential are non-gameable features in differentiating human-authored vs. GPT-4 generated code?

Coding styles are deemed *gameable* if they can be easily and strategically altered or avoided with minimal effort, particularly to mask the AI-generated nature of the code. Examples of such gameable styles include the use of empty lines and whitespace for readability, which can be quickly adjusted without significantly impacting the overall code structure or functionality. If a classifier relies heavily on gameable features, then it may be relatively easy to disguise a GPT-4 generated solution through simple code edits. We hypothesize that a classifier built using non-gameable code-stylometry features can effectively identify GPT-4 generated code.

RQ3: How well does the classifier perform when trained and evaluated on only correct solutions?

We hypothesize that a dataset containing incorrect GPT-4 generated solutions may exhibit distinctive characteristics that could potentially enhance a classifier's ability to identify GPT-4 generated solutions.

RQ4: How well does the classifier perform when trained and evaluated across varying levels of problem difficulty?

We hypothesize that as the complexity of coding problems increases, the discriminative features between human and GPT-4 generated code may become more pronounced. This could be attributed to the unique problem-solving approaches employed by human developers compared to AI systems when faced with complex programming tasks.

[2] https://www.codechef.com/

(3) Our dataset comprising 399 problems and 1596 human and GPT-4 generated solutions is itself a contribution that we make publicly available to other researchers working on the problem of identifying GPT-4 generated code. We also provide a subset of the dataset comprising 161 problems whose solutions have been checked for correctness. We provide a replication package,[3] which includes raw data, feature lists, and code scripts.
(4) We extend our approach to a dataset of introductory programming assignments spanning three years (2019, 2020, and 2023). In this evaluation, we train a classifier on 80 submissions, 40 human and 40 GPT-4 generated submissions from 2019. Subsequently, we evaluate the classifier on 108 submissions from 2020, predating the release of Copilot and ChatGPT. Lastly, we assess the classifier's performance on 54 submissions from 2023, well after the release of these AI assistants, to gauge their usage in assignments. This evaluation offers insights into how promising our approach is on real-world assignments, contributing to the ongoing discourse surrounding the use of AI assistants in academic settings.

The rest of the thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 describes the approach used in this study (including the data collection, feature extraction, and classification phases) and discusses the baselines. Chapter 4 presents the results of our analyses using CodeChef data. In Chapter 5, we replicate our approach on introductory programming assignments and discuss the results. Chapter 6 explains the classifier's predictions, focusing on correctly and incorrectly predicted code. Chapter 7 highlights the threats to validity. Chapter 8 concludes the paper and discusses future work.

[3] https://zenodo.org/doi/10.5281/zenodo.10152237

Chapter 2

Related Work

In this chapter, we review related work on AI-generated code detection, code stylometry, and machine learning applications.

2.1 Detecting AI-generated Code

The prevalence and potency of AI assistants have led researchers to start investigating the problem of detecting code generated by AI assistants. Puryear and Sprint [46] investigated how well-established plagiarism detection tools, MOSS [2], Codequiry [12], and CopyLeaks [15], could detect Copilot-generated solutions among a set of data science programming assignments. They found that Copilot-generated solutions exhibited little similarity to solutions authored by students. The highest observed similarity, identified by MOSS at 36%, fell well below the thresholds of similarity between student solutions that suggest plagiarism. Moreover, when "similar" Copilot and student solutions were manually inspected, the researchers determined that code similarities often reflected standard, commonly employed coding solutions or expected variable declarations. In work that is closest to ours, Bukhari et al. [8] attempt to use machine learning to distinguish between 28 student-authored and 30 AI-generated solutions for a C-language programming assignment involving singly-linked lists. Their approach leverages lexical and syntactic features in conjunction with multiple machine-learning models, achieving an accuracy rate of 92%.

We are also starting to see commercial tools such as HackerRank [26] and Coderbyte [13] that claim to identify AI-generated code within user-submitted code. Unfortunately, evidence of their performance has not been provided and is not freely available for third-party evaluation.

Our study expands on the body of work in this emerging field, employing a more diverse problem set and more descriptive features for interpretability compared to the study by Bukhari et al. [8].

2.2 Code Stylometry

A related research problem focuses on identifying code authorship, typically by using code stylometry, which analyzes distinct coding styles that reflect patterns in the way programmers write code. There exists a substantial body of work on coding constructs that can serve as distinctive identifiers of individual coding styles. Pioneering work by Oman and Cook in 1989 [42] analyzed the authorship of 18 distinct Pascal programs published in six independently authored computer science textbooks. More recent studies have used code stylometry for authorship attribution [4, 9, 16, 23, 28, 50] and plagiarism detection [19, 44, 51]. Although these terms are sometimes used interchangeably, authorship attribution deals with identifying the author of code, whereas plagiarism detection assumes the author is known and aims to identify instances of unoriginal code [31]. In these studies, different code stylometry features were found to be effective, with layout or typographical features [42] proving to be more accurate than Halstead's metrics [27],[1] a conventional complexity metric. Some studies used syntactic features [4, 16, 23, 28, 50], whereas others combined layout, lexical, and syntactic features [9, 19, 51].

Our work leverages many of the code stylometry features used in the above studies for a new purpose: to distinguish between human-authored and GPT-4 generated code.

2.3 Machine Learning Approaches

More generally, machine learning techniques have been applied in various code analysis tasks such as testing [41], defect defection [1], refactoring [35, 49], vulnerability detection [20, 32, 34], program comprehension [48], code smells detection [45], authorship attribution [4, 9, 16, 23, 28, 50], and plagiarism detection [19, 44, 51]. In our work, we use a machine learning technique to determine whether or not a program is GPT-4 generated.

[1] Halstead's metrics are measurable properties based on the author's hypothesis that the structure of code is based on two independent properties—operators and operands.

Chapter 3

Study Design

In this chapter, we describe the different phases of our approach, including data collection, feature extraction, and classification. These phases are illustrated in Figure 3.1.

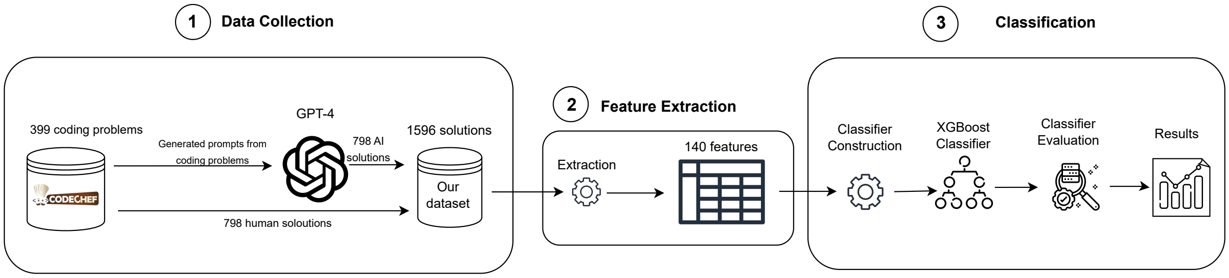


Figure 3.1: An overview of our approach in detecting GPT-4 generated code

3.1 Data Collection

The data collection phase is depicted on the left of Figure 3.1. We collected Python problems and human solutions from a repository of programming problems, and we used an AI assistant to generate AI solutions. We chose Python specifically due to its status as a beginner-friendly language [7] commonly adopted in introductory programming courses [36]. To ensure a wide range of programming problems, we chose CodeChef as our problem repository. CodeChef is a renowned competitive coding platform known for offering problems of varying difficulty levels. We believe that there are no inherent differ-

Table 3.1: Difficulty Levels of Selected CodeChef Problems

Level	Range	Count
Beginner	0 - 999	12
1* Beginner	1000 - 1199	45
1* Advanced	1200 - 1399	71
2* Beginner	1400 - 1499	55
2* Advanced	1500 - 1599	56
3* Beginner	1600 - 1699	60
3* Advanced	1700 - 1799	53
4*	1800 - 1999	30
5*	2000 - 2199	14
6*	2200 - 2499	2
7*	2500 - 5000	1
		399

ences across other competitive programming platforms, so our approach's performance on CodeChef should be similar to its performance on these other platforms.

Due to the absence of public APIs, we extracted data by scraping CodeChef's website, obtaining both problem sets and user submissions. The data collection process is divided into two steps, as highlighted in Figure 3.1. We briefly describe each step below.

Problem Set and Human Solutions Extraction. In this step, we curated a problem set of coding problems from CodeChef and their corresponding human solutions.

CodeChef assigns each problem on its platform a difficulty score and classifies ranges of difficulties into 11 buckets, as depicted in Table 3.1. To ensure that our study's problem set has a good distribution with respect to difficulty, we fetched the 100 most popular problems from each difficulty level. Here popularity refers to the number of accepted solution attempts that existed when the data was scraped (November 2023). In the case that a user submits multiple solutions to the same problem statement, we collect only the latest correct solution submitted by the user. Thus, we began our filtering process with 1100 problem statements.

To further refine our problem set, we selected those problems with at least two correct solutions submitted in 2020. This year was specifically chosen as it postdates the end-of-life for Python 2 in January 2020 and predates the release of AI assistants like Copilot in October 2021 and ChatGPT in November 2022. We purposefully excluded solutions submitted after the release of these AI assistants to ensure that the human-authored solutions

Table 3.2: Final Problem Set Binned into 3 Classes of Difficulty

Difficulty	**Difficulty Scores - Range**	**Average**	**Count**
Easy	828 - 1417	1224.95	133
Medium	1419 - 1646	1529.51	133
Hard	1647 - 3420	1827.50	133

in our dataset are not polluted with AI-generated code. We also purposefully excluded incorrect solutions to avoid including incomplete submissions that do not reflect a typical attempt at a successful solution. In the context of our study, a solution is deemed *correct* if it passes CodeChef's public tests.[1] Such a solution represents successfully interpreted and executed code, effectively solving the specified problem as far as the public tests are concerned.

After this filtering, the number of problems in our dataset was reduced to 419. Upon closer inspection, we found 20 problems that were tagged as "Python3" problems but had only Python2 solutions. These problems were therefore excluded from our study, resulting in a problem set comprising 399 problems, each with at least two correct human-authored Python3 solutions submitted in 2020. In addition, each problem includes comprehensive details from the platform such as the problem statement, unique problem code ID, input and output formats, assigned difficulty score, subtasks, constraints, problem names, user-assigned tags, computed tags, and sample test cases containing input, output, and explanations.

After filtering we noted that we were left with very few problems in the highest levels of difficulty as shown in Table 3.1. This could be attributed to the fact that competitive programmers often choose C/C++ over Python for various reasons, particularly in contests involving problems of higher difficulty. To mitigate issues with inference due to disparities in the number of problems per category, we re-binned the 399 problems into three classes of difficulty (easy, medium, and hard) of equal size. The classes contain 133 problems each, with average difficulty scores of 1224.95, 1529.51, and 1827.50 respectively as shown in Table 3.2. This reclassification of difficulty attempts to ensure a balanced representation across categories and enables a fair evaluation of how well our classifiers fare with respect to problems of different difficulty.

Not including comments. Although comment-based features like commentsDensity [9], inlineCommentsDensity [42], and blockedCommentsDensity [42] have been used in past

[1] In the context of competitive programming, a submission is deemed *correct* if it passes all public and private test cases; however, we had access only to CodeChef's public tests for each problem.

works on authorship attribution, we have chosen not to include comments for two reasons:

1. Comments may make it too easy for a classifier to determine if the code is human-authored or GPT-4 generated. This is because when you specifically ask for comments from a model like GPT-4, the number of comments is far more than any human would normally write. By excluding comments we handicap our approach and thus provide a lower bound for our classifier. Besides this, comment-based features are easily gameable.

2. The amount of comments present in GPT-4 generated code varies by the prompt that we give. If we simply query the API of the model to write a program for the problem at hand, the model generates code only and no comments. Alternatively, if we prompt the model to explain the code, then almost every line of code is commented on. Given that the focus of this work is to explore whether we can differentiate between human-authored and GPT-4 generated code, we did not want the choice of prompt to be a variable of the experiment.

Hence we explicitly asked the GPT-4 model not to explain the code and then we stripped all comments that may have been included even by mistake. We apply the same comment-stripping technique to remove any comments from the human code as well. Therefore in this study, we explicitly avoid using comment-related features for classification.

AI Solutions Generation. To generate AI solutions to the problems in our problem set we used GPT-4 (Version 0613 from OpenAI [30]), which is one of the most powerful and easily accessible generative models available to consumers as of November 2023. We set the temperature to 0 so that our results are reproducible (at the time of this writing, setting a seed for consistent generation was not available through the API). We used the prompt shown in Figure 3.2 to obtain two GPT-4 solutions for each of the 399 problems in our problem set, resulting in 798 unique solutions. In constructing our prompt, we employed strategies recommended by OpenAI for effective prompt engineering.[2] The specific strategies we followed are outlined below:

- **Include Details in Your Query to Get More Relevant Answers:** The prompt specifies the details of the task by defining the format of the input (problem statement, input format, output format, constraints) and the required output (two Python solutions). This helps in getting relevant and specific answers tailored to the given programming problem.

[2] https://platform.openai.com/docs/guides/prompt-engineering/strategy-provide-reference-text

You are an expert Python Programmer. Your job is to look at a programming puzzle provided by the user and output 2 different ways to solve the solution in python.

The Input is provided with the following contents:

{The problem statement}

{How the input would be formatted},

{Format to be followed in the output generated},

{Constraints on the variables specified in the problem}

Make sure to take the input from the user considering the input format Output should be printed as defined in the output format

Do not attempt to explain the solution only output the code in the following format:

[PYTHON1]

{Solution to given puzzle in Python}

[\PYTHON1]

[PYTHON2]

{Alternate solution to given puzzle in Python}

[\PYTHON2]

Figure 3.2: Prompt used for generating 2 GPT-4 code solutions for CodeChef problems

- **Ask the Model to Adopt a Persona:** The prompt begins with "You are an expert Python Programmer." This tactic of persona adoption sets a context for the responses expected and guides the AI to frame its responses within the expertise of a Python programmer.

- **Use Delimiters to Clearly Indicate Distinct Parts of the Input:** The prompt uses a structured format with clear delimiters, such as [PYTHON1] and [PYTHON2], to separate the two different solutions. This helps the AI understand that two distinct solutions are required and organizes the output in a clear, readable manner. It also enables us to programmatically process the outputs generated.

- **Specify the Steps Required to Complete a Task:** While the prompt implicitly suggests the steps (understand the problem, code the solution), it does not explicitly break down the task into smaller steps. In tasks like programming, outlining steps such as analyzing the problem, considering algorithms, and then coding can enhance the quality of the response.

We opted for a zero-shot inference approach. We intentionally did not constrain the output length within the prompt or provide a detailed step-by-step breakdown, among the other suggested strategies to accommodate the diverse nature of problems in our dataset. While this prompt could be further refined, our goal was to develop a pragmatic, ‘best effort’ prompt reflective of what a typical user might employ.

All generated GPT-4 solutions were syntactically valid and could be successfully parsed, which was important for AST-based features. In the case of duplicate solutions to a problem, we reran the prompt to obtain a new solution to swap in.

Private tests for the problems could not be scraped from the platform, thus we evaluated the GPT-4 generated solutions on the available public tests to check whether they were correct to some degree. We found that only 137 problems had two GPT-4 solutions that satisfy the available public tests, and another 24 problems had only one of the solutions passing the test cases. We used this information to create a sanitized set of 161 problems that includes one to two correct GPT-4 solutions for each problem and an equal number of unique and correct human solutions to those problems for RQ3. This dataset is a contribution of the thesis provided in the replication package.[3]

3.2 Feature Extraction

Our study leverages a combination of layout, syntactic, and lexical features that have been effectively used in previous studies for authorship attribution and plagiarism detection among human programmers [9, 18, 21, 25]. Layout features refer to the visual organization of code, such as indentation and spacing. Lexical features, on the other hand, are derived from analyzing the tokens within the code, capturing elements such as keywords and literals whereas syntactic features are extracted based on the code’s structural patterns, involving the arrangement and relationships between various code elements. Our study also incorporates Halstead’s metrics [27], which have been used in previous studies for authorship attribution [6, 42]. We also included additional complexity metrics such as maintainability index [14] and cyclomatic complexity [40] to enrich our approach.

In the feature extraction phase (shown in the middle of Figure 3.1), we iterate through the Python solution files, systematically generating these code stylometry and complexity features essential for training and evaluating our classifier.

We extracted 27 base features excluding Halstead metrics (all shown in Table 3.3) plus variants, leading to 140 features. Most base features have no variants. Feature keywordsDensity has 28 variants, out of 35 Python keywords; these are listed in Table 3.4. Features ASTNodeTypesTF and ASTNodeTypeAvgDep each have 42 variants, out of 130 possible

[3] https://zenodo.org/doi/10.5281/zenodo.10152237

Table 3.3: Code Stylometry and Code Complexity Features

Feature	**Description**
ASTNodeTypesTF [9]	Term frequency of 130 possible AST node types excluding leaves
ASTNodeTypeAvgDep [9]	Average depth of 130 possible AST node types excluding leaves
avgFunctionLength [21]	The average length of lines in a function
avgIdentifierLength [21]	The average length of identifier names
avgLineLength [9]	The average length of characters in each line
avgParams [9]	The average number of parameters across all functions
branchingFactor [9]	Average branching factor of the code's AST
cyclomaticComplexity [40]	The number of decisions within a block of code
emptyLinesDensity [9]	The number of empty lines divided by source code lines
keywordsDensity [9]	Frequency of Python keywords divided by source lines of code
maintainabilityIndex [14]	A metric that gauges the ease of supporting and modifying the source code
maxDecisionTokens	The maximum number of tokens in decision conditions excluding ternary conditions
maxDepthASTNode [9]	Maximum depth of an AST node
nestingDepth [9]	Deepest level to which conditional statements, loops, and functions are nested within each other
numAssignmentStmtDensity [18]	The total number of assignment statements divided by source code lines
numClassesDensity	The total number of classes divided by source code lines
numFunctionCallsDensity [18]	The total number of function calls divided by source code lines
numFunctionsDensity [18]	The number of functions divided by source code lines
numInputStmtsDensity [18]	The total number of input statements divided by source code lines
numKeywordsDensity [9]	The total number of unique Python keywords divided by source code lines
numLiteralsDensity [9]	The number of literals divided by sloc
numStatementsDensity [18]	The total number of statements divided by source code lines
numVariablesDensity [18]	The total number of assignment variables divided by source code lines
numberOfDistinctOperands [27]	The number of distinct operands
numberOfDistinctOperators [27]	The number of distinct operators
sloc [25]	The total number of source code lines
stdDevLineLength [9]	The standard deviation of character lengths of each line
stdDevNumParams [9]	The standard deviation of the number of parameters across all functions
totalNumberOfOperands [27]	The total number of operands
totalNumberOfOperators [27]	The total number of operators
whiteSpaceRatio [9]	The ratio of whitespace characters to non-whitespace characters

AST node types; these are listed in Table 3.5. The prefixes "nttf_" for ASTNodeTypesTF, "ntad_" for ASTNodeTypeAvgDep, and the suffix "_Density" for KeywordsDensity were adopted to correlate variants with their respective base features. This accounts for why there are less features extracted from this dataset than CodeChef's.

Our approach to feature extraction varies somewhat from prior studies [9, 18, 42]. We normalized our features by source lines of code rather than by character count [9] or by omitting normalization altogether [18, 42]. Additionally, we refrained from logarithmic transformations of some features, as practiced by Caliskan et al. [9], to facilitate the ease of interpretability of our feature set, particularly for visual analysis. For the nestingDepth feature, we considered node types rather than actual tokens. The MaintainabilityIndex feature measures code maintainability by evaluating complexity and modularity and is calculated using complexity metrics such as Cyclomatic Complexity, and SLOC. Our version was computed with the Radon Python library,[4] which uses a modified formula different from that used in the study by Coleman et al. [14]. Although our dataset contains only single-file solutions, the MaintainabilityIndex feature is included in our study as it may yield insights into the relative maintainability of code produced by AI assistants compared to code authored by humans, potentially impacting the performance of our classifier.

Of the 140 features extracted, four are Halstead metrics, selected to explore their viability in this context. Their ineffectiveness in the context of authorship attribution in human code has been pointed out by Berghel and Sallach [6] and Oman and Cook [42], but we included them for completeness. Our research is directed at assessing how well these metrics can identify GPT-4 generated code.

Table 3.4: Python Keywords

and	as	break	class	continue	def	del
elif	else	except	False	for	from	global
if	import	in	is	lambda	None	not
or	pass	return	True	try	while	yield

Table 3.5: Python AST Node Types

arg	arguments	Assign	Attribute	AugAssign	BinOp
BoolOp	Call	ClassDef	Compare	comprehension	Delete
Dict	DictComp	ExceptHandler	Expr	For	FormattedValue
FunctionDef	GeneratorExp	If	IfExp	Import	ImportFrom
JoinedStr	keyword	Lambda	List	ListComp	Module
Name	Return	Set	SetComp	Slice	Starred
Subscript	Try	Tuple	UnaryOp	While	Yield

3.3 Classification

The righthand side of Figure 3.1 provides an overview of the classification phase, where we construct a model to distinguish between GPT-4 generated and human-authored code and subsequently evaluate its performance. We describe each step of this phase below.

Classifier Construction. To construct our classifier, we chose XGBoost [10], because

[4] https://pypi.org/project/radon/

it is an effective and scalable machine learning algorithm. Additionally, in the study by Bukhari et al [8], XGBoost with syntactic and lexical features had the best performance when considering accuracy and F1 score. This allows us to compare our approach with the best of the earlier study's approaches. XGBoost constructs decision trees iteratively, refining the model by correcting misclassifications at each step. During training, the algorithm optimizes an objective function to strike a balance between prediction accuracy and model simplicity. At each tree, the algorithm assigns scores to examples, and each example's final prediction is calculated by summing the scores [10]. Through this aggregation process, the resulting model classifies code as either human-authored or GPT-4 generated.

Classifier Evaluation. To evaluate the classifier's performance on unseen data, we employ ten-fold cross-validation, which divides the dataset into ten subsets. This approach provides robust assessments by training on nine subsets and testing on the remaining subset. This process is iterated ten times, guaranteeing that each subset serves as the test set exactly once.

To enhance the evaluation process and avoid data poisoning during training, we grouped solutions based on the specific coding problem they addressed, resulting in 399 distinct groups corresponding to the 399 coding problems in our dataset. We employed GroupKFold to ensure that each group, representing solutions to a particular problem, appeared only once in the test set across all folds. This grouping strategy maintains the integrity of the evaluation by preserving the context of solutions within each problem, preventing the model from training and testing on the same problem sets since each problem set contains multiple solutions.

The classifier's predictions yield four possible outcomes, described below.

- **True Positive (TP):** A true positive occurs when the classifier correctly labels a GPT-4 generated code.
- **True Negative (TN):** A true negative occurs when the model classifier correctly labels a human-authored code.
- **False Positive (FP):** A false positive occurs when the classifier incorrectly labels human-authored code as GPT-4 generated.
- **False Negative (TN):** A false negative occurs when the model incorrectly labels code generated by GPT-4 as human-authored.

Based on these possible outcomes, the model's performance is measured in terms of accuracy, recall, precision, F1-score, and AUC-ROC. We describe each metric below.

- **Accuracy:** Accuracy gauges the model's overall correctness in classifying both GPT-4 generated and human-authored code. It is calculated as

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{3.1}$$

- **Precision:** Precision measures the proportion of code classified as GPT-4 generated that were truly GPT-4 generated. It is calculated as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.2}$$

- **Recall**: Recall is the measure of sensitivity that quantifies the model's ability to identify all GPT-4 generated code correctly. It is calculated as

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.3}$$

- **F1-score:** F1-score is the harmonic mean of precision and recall ensures a balanced evaluation when precision and recall need equal consideration. It is calculated as

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

- **Area Under the Curve of the Receiver Operating Characteristic (AUC-ROC):** AUC-ROC evaluates the model's ability to differentiate between GPT-4 generated code and human-authored code. It uses the true positive rate ($\frac{TP}{TP+FN}$) versus false positive rate ($\frac{FP}{FP+TN}$) at various thresholds settings. AUC ranges from 0 to 1, higher values indicate better performance.

Baselines Comparison. In our model comparison, we evaluate our approach alongside two baselines: (1) a naïve baseline approach, based on the assumption that GPT-4 generated code can be detected through random guessing, and (2) the approach presented by Bukhari et al. [8] that identified AI-generated solutions for C programming assignments. To benchmark our classifier, we replicated the methodology of Bukhari et al. [8] using Python and an XGBoost classifier, selecting XGBoost because it performed the best in their study on syntactic and lexical features. We compare the performances of our classifier and these baseline models using metrics such as accuracy, recall, precision, F1-score, and AUC-ROC.

Given that our case study focuses solely on utilizing the CodeChef dataset, we will not employ statistical significance tests like the Mann-Whitney U or T-test between our approach and the second baseline. These tests typically require multiple datasets, a requirement our single-dataset study does not meet. Additionally, such tests cannot be employed on the classifier's raw prediction scores due to the presence of high probability misclassifications. Furthermore, applying statistical significance tests to performance metrics derived from ten-fold cross-validation is problematic due to the overlap of the training dataset across folds, with about 80% of data shared between each pair of training sets [17]. This overlap leads to interdependence among the folds, thereby violating the independence assumption of these tests. As a result, we risk incorrectly rejecting the null hypothesis even in instances where there may be no actual difference.

Chapter 4

Results

In this chapter, we present the results of our evaluations based on the four research questions introduced in Chapter 1. We first outline our approach to addressing each research question, followed by the observed results.

4.1 *RQ1. How well can code-stylometry features distinguish human-authored code from GPT-4 generated code?*

Approach. To address our research question, we build a classifier trained on code stylometry features. After training, we evaluate its performance using ten-fold cross-validation, focusing on precision, recall, F1-score, and AUC-ROC metrics. High scores in these metrics will support our research hypothesis, demonstrating our approach's ability to distinguish human-authored code from AI-generated code. Additionally, we compare this classifier with an alternative classifier trained on the same feature set but augmented with Halstead's metrics. The objective is to evaluate the impact of Halstead's metrics in detecting GPT-4 generated code. This will involve a comparative analysis of the classifiers' performances with and without Halstead's metrics. Moreover, we contrast the performance of our classifier with two baselines.

The Naïve Baseline is a random guess and its performance metrics can be computed by applying statistics to our dataset. The precision is calculated by dividing the number of GPT-4 generated code by the total number of solutions:

Table 4.1: Classifier Performance Comparison among Different Approaches for Distinguishing between AI-generated and Human-authored Code

	Our Approach		**Baseline**		
	All	**Non-Gameable**	**Naive**	**n-grams + L**	
				n = 2	**n = 3**
Accuracy	0.91	0.89	-	0.86	0.88
Precision	0.91	0.89	0.5	0.86	0.87
Recall	0.91	0.89	0.5	0.88	0.88
F1-score	0.91	0.89	0.5	0.87	0.88
AUC-ROC	0.91	0.89	-	0.86	0.88

$$\text{Precision} = \frac{\text{number of GPT-4 generated code}}{\text{total number of solutions}} = 0.5 \tag{4.1}$$

The recall is 0.5, reflecting the classifier's two possible outcomes—identifying code as either GPT-4 generated or human-authored. This results in a probability of 0.5 for classifying solutions as GPT-4 generated. Based on precision and recall values, the F1-score of the naïve baseline is 0.5 and is computed using Equation 3.4.

Results. As shown in Table 4.1,[1] our approach achieved a high average precision and recall of 0.91, ensuring accurate and comprehensive identification of GPT-4 generated code. This highlights the potential of code stylometry in differentiating between GPT-4 generated and human-authored code. Comparing the classifiers, one with and the other without Halstead metrics, we found a striking similarity in their performance metrics. Between both classifiers, all the metrics considered were the same, except recall which was higher for the classifier with Halstead metrics by 0.01. This observation suggests that the presence of Halstead metrics does not considerably enhance the classifier's ability to distinguish between human-authored and GPT-4 generated code, supporting past work [25, 38].

When compared to the baselines, our classifier shows a considerable improvement. It notably outperformed the Naïve Baseline, which has a precision and recall of 0.5, demonstrating that our classifier considerably exceeds what would essentially be random guessing. We also compare with the work presented in Bukhari et al. [8] that incorporated lexical features of 2-4 n-grams. However, in our replication, we could only process 2-3 n-grams

[1] It is just a coincidence that the performance metrics for our models all have the same value when considering all features (0.91) and all non-gameable features (0.89)

due to the memory-intensive nature of the task. The data for the 4-gram model was at least 212.33 GBs, resulting in out-of-memory errors on our machine (Macbook Air 2020, with 16GBs of RAM). Also, it took more than 8 hours to extract the data for the 4-gram model. However, our classifier took less than 3 minutes with no additional space. We were also able to achieve higher precision and recall by 4% and 3% respectively.

Moreover, unlike the baseline models, which may obscure the reasoning behind predictions, our model uses a feature set that clarifies the decision-making process. To understand the influence of specific features on our model's predictions, we used the SHAP framework [39], a method renowned for its interpretability of machine learning models. SHAP offers tools for both local (individual) and global (overall) explanations of model predictions.

The global interpretive power of our classifier is demonstrated in the SHAP summary plot depicted in Figure 4.1. This plot visualizes key features in our classifier, arranging them on the y-axis by their aggregate SHAP values, with the highest predictive feature at the top. The x-axis displays these SHAP values, showing how each feature shifts the prediction from a neutral base value, indicated by the vertical line at 0 on the x-axis. Deviations to the left or right increase the likelihood of the prediction being a human or GPT-4 class, respectively. The plot uses a blue-to-red color gradient to signify feature magnitudes, and the data points represent feature values across instances.

For example, in Figure 4.1, the *avgLineLength* is the most important feature, distinctly separating human and GPT-4 classes where lower values are typically associated with the human class and higher values are typically associated with the GPT-4 class. This implies that a line of code from a human is shorter on average compared to a line of code from GPT-4. In contrast, the *ntad_Name* is the tenth most important feature and quantifies the average depth at which the *Name* nodes occur within an AST. *Name* nodes in the AST of a Python program represent identifiers, which are the names of variables, functions, classes, modules, or other objects in the code. The distribution of the *ntad_Name* feature across all instances shows a relatively narrow range of SHAP values, suggesting that its impact on the model is less compared to the features ranked above it. It is important to note that SHAP functions as an explanation model that provides an interpretable approximation of our classifier. Therefore, while SHAP values offer a simplified and interpretable view, the feature importance rankings derived directly from our classifier are based on the classifier's internal mechanisms. Consequently, the feature importance rankings from our classifier, do not entirely align with SHAP's. Despite these differences, there is considerable overlap in the primary features identified by both methods. This overlap highlights the value of SHAP in interpreting the model's predictions, offering insights into how features influence outcomes rather than detailing the classifier's internal mechanisms.

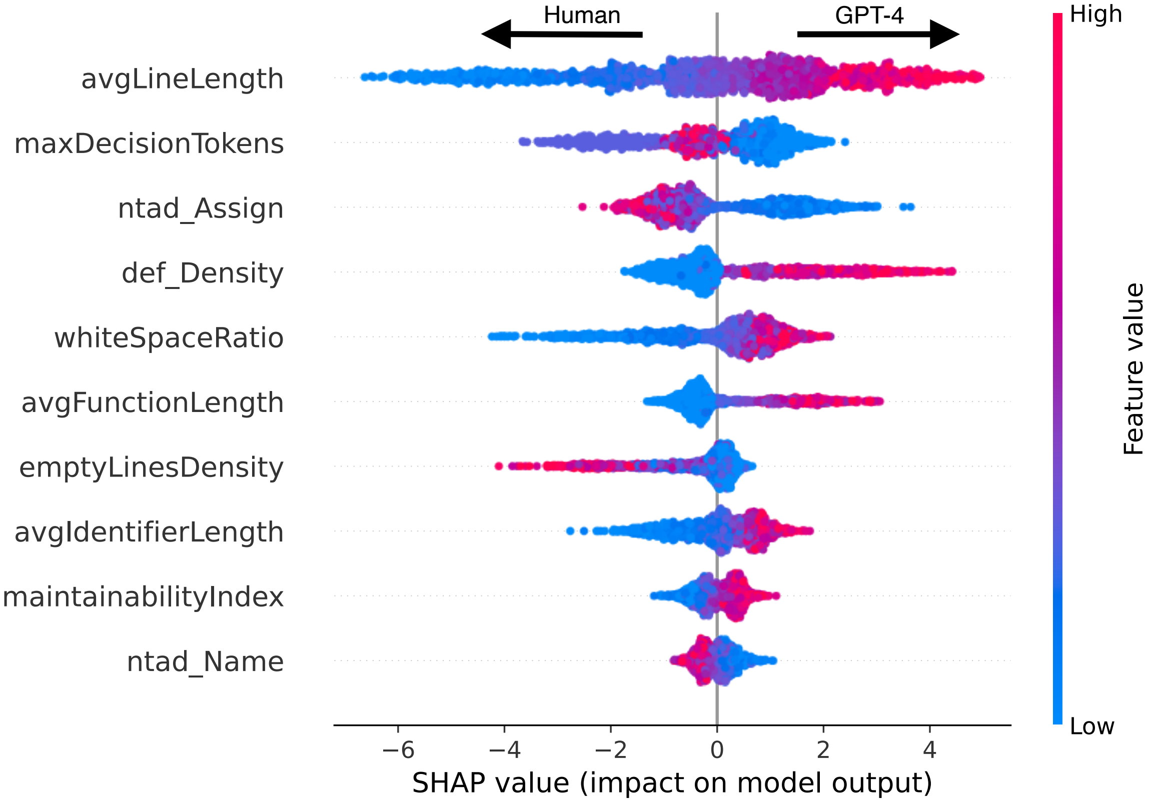


Figure 4.1: SHAP feature importance of our approach

4.2 *RQ2. How influential are non-gameable features in differentiating human-authored vs. GPT-4 generated code?*

Approach. To assess the impact of non-gameable features on the detection of GPT-4 generated code, we build a classifier that excludes gameable features.

In our analysis, we consider the non-code layout features *emptyLinesDensity* and *whiteSpaceRatio* as gameable features. After training on the non-gameable features, we evaluate the performance of this classifier in contrast to our classifier with both gameable and non-gameable features. We also evaluate this classifier with the same n-grams baseline compared in RQ1, as that baseline does not include any feature we consider gameable in its feature set. Through these comparisons, we aim to evaluate the relative importance of

Table 4.2: Classifier Performance Comparison on Correct and Randomly Sampled Solutions

	Our Approach		**Baseline (n-grams + L)**			
			n = 2		**n = 3**	
	C	**R**	**C**	**R**	**C**	**R**
Accuracy	0.86	0.87	0.83	0.84	0.87	0.86
Precision	0.87	0.87	0.83	0.84	0.87	0.86
Recall	0.86	0.88	0.81	0.85	0.87	0.85
F1-score	0.86	0.87	0.82	0.84	0.87	0.86
AUC-ROC	0.86	0.87	0.83	0.84	0.87	0.86

C = Correct Solutions, R = Random Solutions, L = Lexical Features

using only non-gameable features in the classification of code as human-authored or GPT-4 generated.

Results. As shown in Table 4.1, there is a noticeable but not severe drop in performance for the non-gameable classifier compared to the classifier built on gameable and non-gameable features. This suggests that although gameable features contribute to the classifier's accuracy, non-gameable features alone still provide a high predictive power. In comparison to the n-grams baseline, the non-gameable classifier still performs better and is interpretable as evident in the SHAP summary plot of Figure 4.2. The plot reveals that, aside from the two gameable features among the ten most important features of our origin classifier trained on both gameable and non-gameable features shown in Figure 4.1, the other features remain consistent with a slight reordering. The absence of these two gameable features accounts for the performance dip in the non-gameable classifier. Consequently, *stdDevLineLength* and *nttf_Name*, the latter representing the term frequency of *Name* nodes, now appear in the top ten. The *stdDevLineLength* feature influences predictions towards the GPT-4 class at lower values and towards the human class at intermediate values. Conversely, *nttf_Name* influences predictions towards the human class at higher values and towards the GPT-4 class when lower.

4.3 *RQ3. How well does the classifier perform when trained and evaluated on only correct solutions?*

Approach. To address this research question, we refine our dataset to include only correct solutions, resulting in a balanced dataset of 596 correct solutions from both humans and

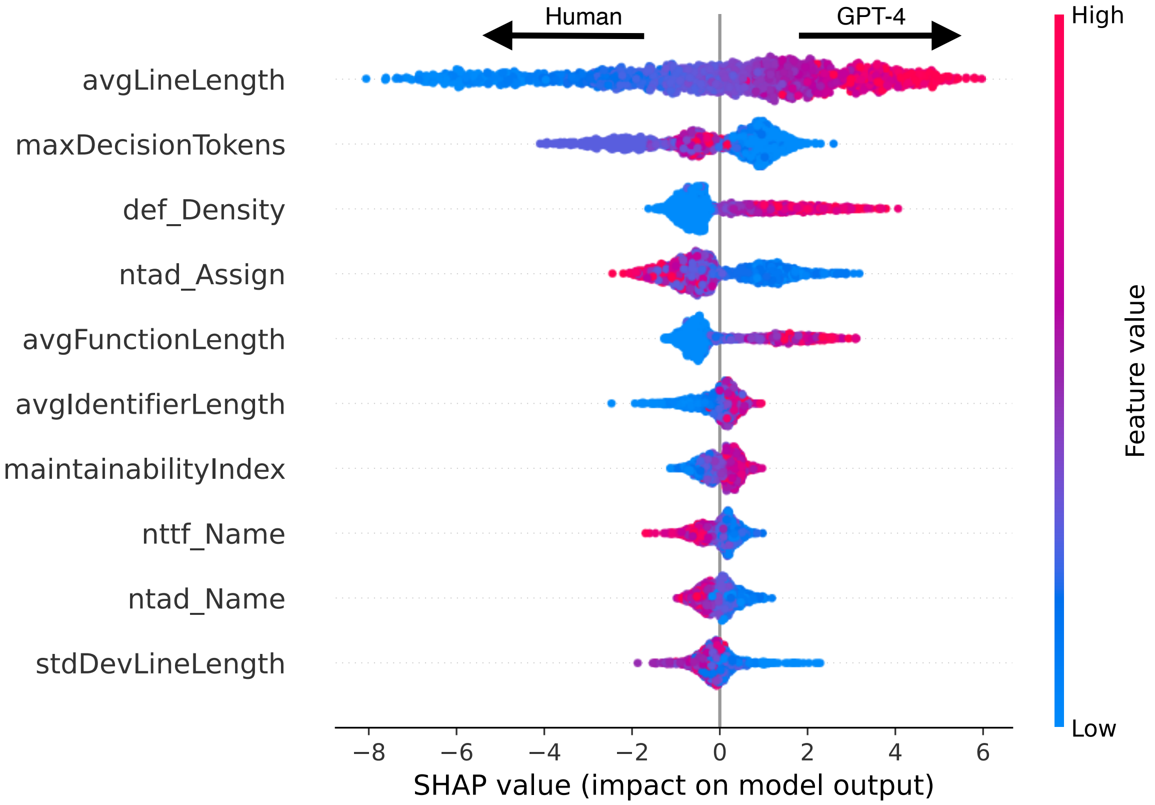


Figure 4.2: SHAP feature importance of non-gameable features

GPT-4. We aim to evaluate our classifier's predictive power by eliminating the potential noise that incorrect solutions might introduce. This ensures that the detection of GPT-4 generated solutions is based on inherent characteristics of code, not the errors they might contain.

Results. Table 4.2 shows a slight decline in performance when compared to the classifier in Table 4.1. The minor decline could stem from the correct solutions being a smaller subset. Hence, we randomly sample the solutions comprising both correct and incorrect solutions with the same distribution of difficulty levels. When compared with the random solutions classifier, both perform almost the same. This implies that genuine differences in coding styles between human and GPT-4 generated code are being detected, rather than errors introduced by incorrect solutions. When compared to the baseline [8], for correct solutions, our classifier outperformed the 2-gram baseline but was marginally less effective than the 3-gram. The classifier with randomly sampled solutions showed

Table 4.3: Classifier Performance Comparison Across Levels of Problem Difficulty

	Our Approach			**Baseline (n-grams + Lexical Features)**					
	Easy	**Medium**	**Hard**	**n = 2**			**n = 3**		
				Easy	**Medium**	**Hard**	**Easy**	**Medium**	**Hard**
Accuracy	0.89	0.89	0.87	0.87	0.79	0.80	0.89	0.86	0.80
Precision	0.87	0.88	0.89	0.85	0.80	0.79	0.89	0.87	0.80
Recall	0.91	0.90	0.86	0.89	0.77	0.82	0.88	0.85	0.81
F1-score	0.89	0.89	0.87	0.87	0.79	0.80	0.89	0.86	0.80
AUC-ROC	0.89	0.89	0.87	0.87	0.79	0.80	0.89	0.86	0.80

noticeable improvement over the 2-gram baseline and was marginally better than the 3-gram.

4.4 *RQ4. How well does the classifier perform when trained and evaluated across varying levels of problem difficulty?*

Approach. To address this research question, we construct separate classifiers for the three problem-difficulty levels outlined in Table 3.2. By training and evaluating these classifiers independently, we evaluate their performance and the potential impact of problem complexity on the classifier's ability to correctly identify GPT-4 generated code. This stratified analysis allows us to understand the nuances of classifier performance across problem difficulty levels.

Results. Table 4.3 shows a correlation between the classifier performance and the difficulty of the problems. The performance of classifiers of easy and medium problem difficulty are close as they have the same F1-score of 0.89. The performance of the classifier with hard problems had a minor drop in performance with an F1-score of 0.87. When compared to the baseline, there is no considerable difference in the baseline for easy questions. For both medium and hard questions, our classifiers perform better than both 2-gram and 3-gram classifiers, showing improvements of 3% and 9%, respectively. This result highlights the effectiveness of our approach.

Chapter 5

Evaluation on Introductory Programming Assignments

In this chapter, we replicate our approach on introductory programming assignments, then we investigate the viability of assignment submissions being GPT-4 generated in the era of AI assistants such as Copilot and ChatGPT.

5.1 Replicating Our Approach Using Introductory Programming Assignments

To address our research questions within an academic setting, we replicated our approach with introductory programming assignments, specifically targeting RQ1 and RQ2. Due to certain limitations, this replication could not be extended to address RQ3 and RQ4. For RQ3, we were not provided with test suites, preventing the assessment of assignment correctness. Whereas for RQ4, the limitation was due to assignments not being categorized into difficulty levels, preventing comparisons across these levels. We collected assignments (comprising questions, test case requirements, starter code, style guides that students were expected to follow and student submissions) from 2019 and 2020 as shown in Table 5.1. Considering the timeline of the release of AI assistants such as Copilot and ChatGPT, we categorized student submissions from these years as human-authored since they predate the release of these AI assistants. We collected 27 questions from both 2019 and 2020. However, from the 27 questions in 2019, we excluded 7 questions because they either depended on image descriptions for their solutions or they referenced the solutions of previous questions and thus could not be solved independently. For each question in 2019 and 2020, we randomly selected 2 student submissions and generated 2 corresponding GPT-4 solutions using a different prompt shown in Figure 5.1 that accommodated the style guide and test case requirements.

Table 5.1: Introductory Programming Course Assignments: Pre (2019, 2020) and Post (2023) Copilot and ChatGPT release

Year	**Questions**	**Submissions**		
		Human	AI	Total
2019	20	40	40	80
2020	27	54	54	108
2023	27	-	-	54

2023 submissions are unlabeled

You are a student in a Python course. Your task is to look at the problem statement enclosed within <question></question> tags provided by the user, and provide two distinct Python solutions. Enclose the first solution in <python1></python1> tags and the second in <python2></python2> tags. In addition to the problem statement, the user also provides you with one or more examples enclosed within <example></example> tags illustrating the expected output of the resulting solution, starter code enclosed within <codeblock></codeblock> tags, restrictions enclosed within <restrictions></restrictions>tags that you must follow when writing your code, an interface script enclosed within <interface></interface> tags to assist with the implementation of your code, and a style guide enclosed within <style></style> tags that you must adhere to when writing your code. You are also provided with a check module enclosed within <check></check> tags, containing methods for creating test cases for your code. Use this module if asked to in the problem statement.
The problem statement: <question>{problem_statement}</question>
Example(s): <example>{example}</example>
Code block: <codeblock>{code_block}</codeblock>
Style guide: <style>{style_guide}</style>

Figure 5.1: Prompt used for generating 2 GPT-4 code solutions for assignment questions

To answer RQ1, we extracted 119 code stylometry features from 2019 and 2020 submissions. These features consist of the 31 base features in Table 3.3. However, the base

Table 5.2: Python Keywords

and	as	break	continue	def	elif
else	False	for	from	if	import
in	is	lambda	None	not	or
pass	return	True	while	with	

Table 5.3: Python AST Node Types

arguments	Assign	Attribute	AugAssign	BinOp	BoolOp
Call	Compare	comprehension	Dict	DictComp	Expr
For	FormattedValue	FunctionDef	GeneratorExp	If	IfExp
Import	ImportFrom	JoinedStr	keyword	Lambda	List
ListComp	Module	Name	Return	Slice	Starred
Subscript	Tuple	UnaryOp	While	With	withitem

feature keywordsDensity has 23 variants out of 35 Python keywords; these are listed in Table 5.2. The base features ASTNodeTypesTF and ASTNodeTypeAvgDep each have 36 variants, out of 130 possible AST node types; these are listed in Table 5.3. All three base features have fewer variants than those of the corresponding base features in the CodeChef dataset. This accounts for why fewer features are extracted from this dataset than from CodeChef's. Subsequently, we constructed a classifier using the code stylometry features extracted from 2019 submissions and evaluated its performance on 2020 submissions.

As shown in Table 5.4, the classifier built with all features achieved precision, recall, and F1-Score of 0.82, 0.59, and 0.69 respectively. These results indicate that although the classifier prioritizes correctly identifying GPT-4 submissions, it tends to overlook many actual GPT-4 submissions. In academia, this is pardonable as erring on the side of caution is preferable because falsely accusing a student of cheating can have serious implications.

To better understand the classifier's predictions, we look at the important features in the SHAP summary plot shown in Figure 5.2. The most important feature is the *avgLineLength* which indicates that submissions from students have less code on each line on average in contrast to GPT-4 solutions, a finding that is consistent with our analysis of CodeChef solutions. The next important feature is the *avgIdentifierLength* which shows that identifier names in student submissions are typically shorter or of medium length, whereas GPT-4 tends to generate identifiers that are longer on average. This observation suggests that GPT-4 solutions are characterized by more meaningful identifier names.

To answer RQ2, we built a classifier using only non-gameable features from the 2019 submissions and evaluated its performance using non-gameable features from the 2020

Table 5.4: Classifier Performance on Introductory Programming Assignments Trained on 2019 Assignments and Evaluated on 2020 Assignments

	All	**Non-Gameable**
Accuracy	0.73	0.72
Precision	0.82	0.82
Recall	0.59	0.57
F1-Score	0.69	0.67
AUC-ROC	0.73	0.72

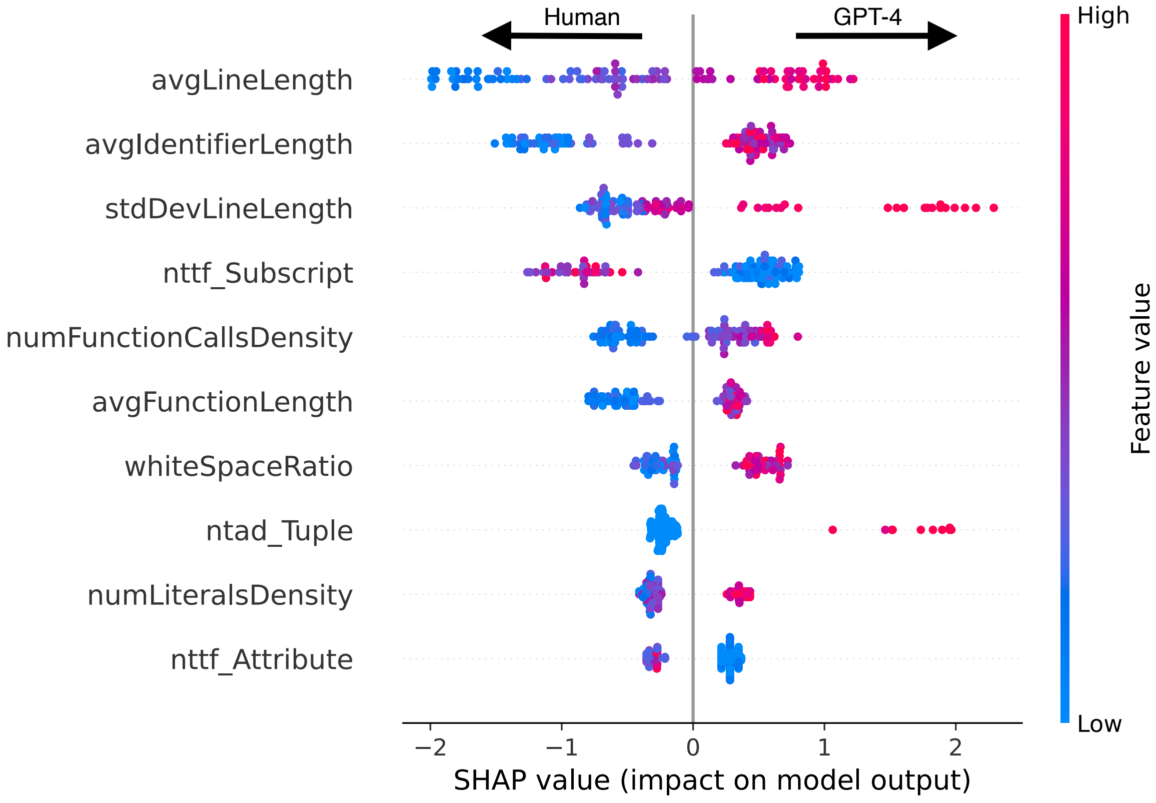


Figure 5.2: SHAP feature importance among all assignment features

submissions.

As shown in Table 5.4, there is a modest decrease in performance when the classifier relies solely on non-gameable features, rather than a combination of gameable and non-gameable features. This pattern mirrors the trend observed with classifiers trained on CodeChef solutions. Notably, while the inclusion of gameable features enhances the overall accuracy, the non-gameable features by themselves still yield a comparable level of predictive performance. The SHAP summary plot in Figure 5.3 provides insight into the important features driving the non-gameable features classifier's predictions. It reveals that the top two most important features remain consistent with those from the all-features classifier and are, notably, non-gameable. However, the *whiteSpaceRatio*—a gameable feature—also stands out as significant, ranking among the top ten in the all-features classifier, as shown in Figure 5.2. This particular feature's prominence may account for the slight decline in the non-gameable features classifier's performance when it is excluded.

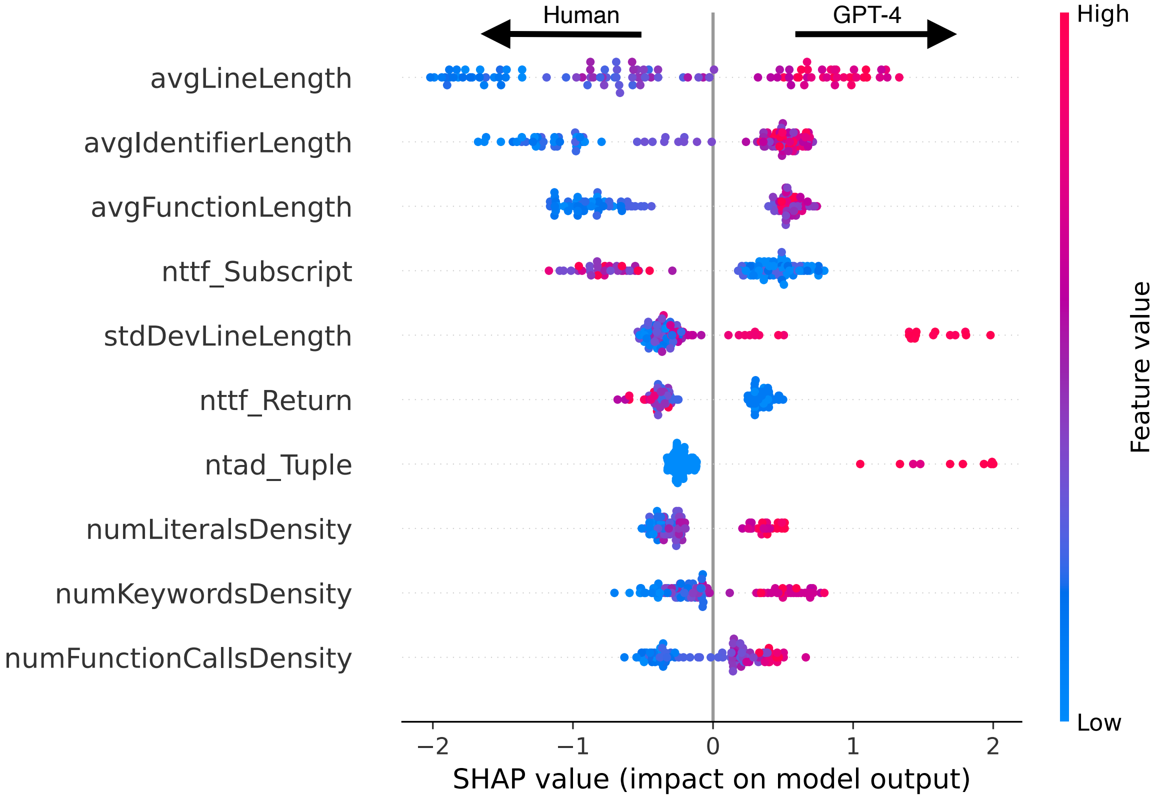


Figure 5.3: SHAP feature importance of classifier built on non-gameable assignment features

5.2 Investigating the Authorship of Submissions in the Era of AI Assistants

To investigate the authorship of submissions made well after the release of Copilot and ChatGPT, we collected assignments, comprising questions and student submissions, from the year 2023, as shown in Table 5.1. Given the prevalent use of these AI tools at that time, the origins of these submissions were considered unknown. Accordingly, we treated the 2023 submissions as unlabeled to reflect this uncertainty. We collected 27 questions and we randomly selected 2 submissions for each question. Using the all-features classifier developed from the 2019 submissions, we assessed the authorship of the 2023 submissions.

Although we cannot be entirely certain, we suspect students may be using AI assistants for their programming assignments. In the classification of the 2023 submissions, 13 out of 54 were identified as AI-generated. The classifier trained on 2019 submissions achieved a 73% accuracy rate when applied to the 2020 submissions, revealing a false positive rate of 13% and a false negative rate of 41%. Thus, it can be estimated that the number of AI-generated submissions in the 2023 dataset falls within the range of 11 to 18 submissions, representing approximately 20% to 33% of the total (54) submissions considered.

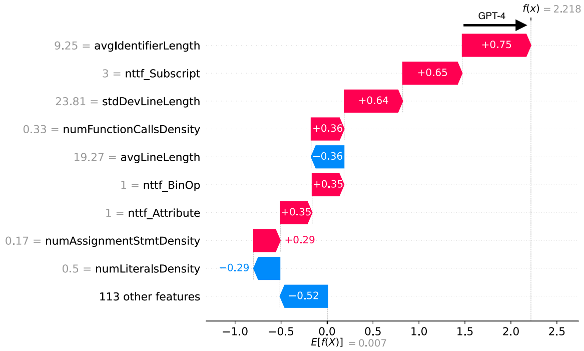
To better understand the predictions of the classifier built on 2019 submissions and evaluated on 2023 submissions, we investigate individual predictions, analyzing the waterfall plots and code samples for high, low, and average probability predictions and relating these predictions to the important features in the SHAP summary plot in Figure 5.2. Consider the code presented in Figure 5.4, classified as GPT-4 generated with a high probability of 0.9, and its corresponding SHAP waterfall plot. The most important feature contributing to this classification is *avgIdentifierLength*, which is the second most important feature in the summary plot, indicating that GPT-4 generated code is characterized by having longer identifier names, possibly reflecting more meaningful naming conventions, as shown with most of the identifiers in the code in Figure 5.4a. Despite *avgLineLength* and *numLiteralsDensity* having lower values that push the classifier's prediction towards the human class, the overall prediction of GPT-4 is strongly influenced by the cumulative effect of other features. Among these features are five of the most important features. These include *avgIdentifierLength*, *nttf_Subscript*,[1] *stdDevLineLength*, *numFunctionCallsDensity*, *nttf_Attribute*.[2] Code with a combination of the values for these features suggests that the code is likely GPT-4 generated. Conversely, Figure 5.5 presents code classified as human-authored with a high probability of 0.98 and its corresponding waterfall plot.

[1] The term frequency for node types that access sequence structures like lists and dictionaries.

[2] The term frequency for accessing an object's attributes.

```
def calculate_points(all_players, all_goals):

  if all_players==[]:
    return []
  else:
    T=[[all_players[0], all_goals.count([all_players[0]])]]
    return T + calculate_points(all_players[1:], all_goals)
```

(a) GPT-4 generated code

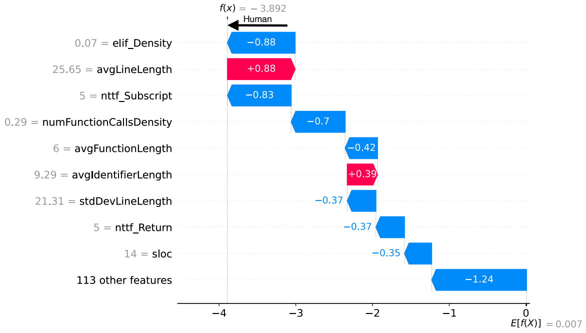


(b) SHAP waterfall plot showing how features impact the model's decision for this code

Figure 5.4: Code classified as GPT-4 generated with a high probability of 0.9

```
def player_points(player, all_goals):
  if all_goals == []:
    return 0
  elif player in all_goals[0]:
    return 1 + player_points(player, all_goals[1:])
  else:
    return player_points(player, all_goals[1:])

def calculate_points(all_players, all_goals):
  if all_players == []:
    return []
  else:
    name = all_players[0]
    points = player_points(name, all_goals)
    return [[name, points]] + calculate_points(all_players[1:], all_goals)
```

(a) Human code



(b) SHAP waterfall plot showing how features impact the model's decision for this code

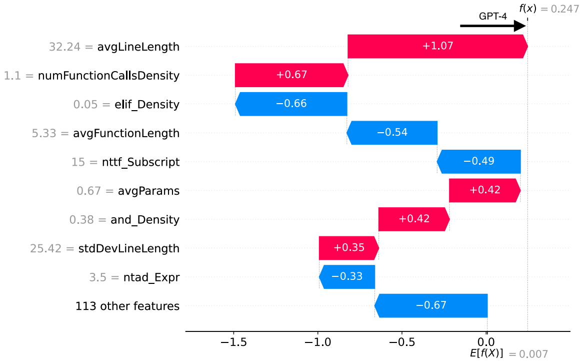
Figure 5.5: Code classified as human-authored with a high probability of 0.98

As shown in Figure 5.5b, although two of the classifier's most important features influence the prediction towards the GPT-4 class, all other features influence the classifier's prediction towards the human class.

To better understand the classifier's lower-confidence predictions, we analyzed instances of both GPT-4 and human-authored code, each with a low probability score of 0.56, and their corresponding waterfall plots as shown in Figures 5.6 and 5.7, respectively. In Figure 5.6b, values for *stdDevLineLength*, *and_Density*, *avgParams*, *numFunctionCallsDensity*, and *avgLineLength* push the prediction towards the GPT-4 class despite having a large number of features indicating human authorship. *numLiteralsDensity* influence the prediction towards the GPT-4 class.

```

input_prompt = "Please enter a valid Canadian postal code: "
error_msg = "Invalid postal code."

def isvalid(p):
  return (len(p) == 7) and  p.isupper() and (p[0]+p[2]+p[5]).isalpha() and \
  (p[1]+p[4]+p[6]).isdigit() and (p[3] == " ") and (p.find("D") == p.find("F") \
  == p.find("I") == p.find("O") == p.find("Q") == p.find("U") == -1) and \
  (not p[0] == "W") and (not p[0] == "Z")

def postal_code_w_counter(t):
  p = input(input_prompt)
  if (t <= 1) and (not isvalid(p)):
    print(error_msg)
  elif isvalid(p):
    abc = "0ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    return abc.index(p[0])*1000**6 + int(p[1])*1000**5 + \
    abc.index(p[2])*1000**4 + int(p[4])*1000**2 + abc.index(p[5])*1000 + \
    int(p[6])
  else:
    print(error_msg)
    return postal_code_w_counter(t-1)

def postal_code():
  return postal_code_w_counter(4)
```

(a) GPT-4 generated code



(b) SHAP waterfall plot showing how features impact the model's decision for this code

Figure 5.6: Code classified as GPT-4 generated with a low probability of 0.56

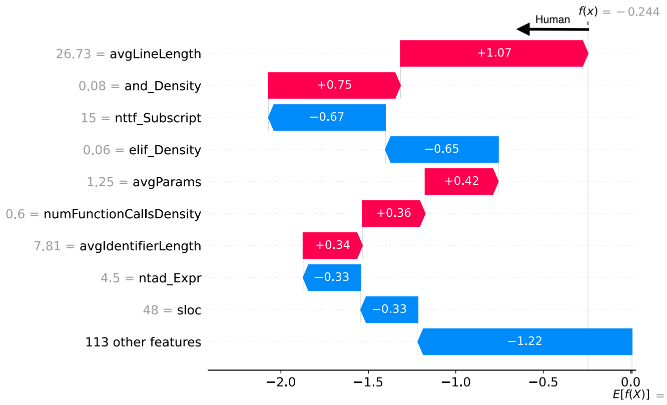
This suggests that although the code shown in Figure 5.6a has feature values that are typically associated with human-authored code, it also has enough feature values to cause the final prediction to be GPT-4 generated code. Conversely, Figure 5.7b shows that 113 features, and the values for *sloc*, *ntad_Expr*,[3] *elif_Density*, *nttf_Subscript* strongly influence the classifier's final prediction to be the human class; even though the values for *avgIdentifierLength*, *numFunctionCallsDensity*, *avgParams*, *and_Density*, and *avgLineLength* influence the prediction towards the GPT-4 class.

Finally, we look at code examples with average confidence levels of prediction. Consider Figure 5.8, which shows code predicted to be GPT-4 generated with a probability of 0.77, along with its SHAP waterfall plot. Figure 5.9 shows code predicted to be human-authored with a probability of 0.78 along with its corresponding waterfall plot. Figure 5.8b shows how the values of some of the ten important features such as *stdDevLineLength*, *avgLineLength*, *avgIdentifierLength*, *avgFunctionLength*, *numLiteralsDensity* and another feature, *numAssignmentStmtDensity* influence the classifier's prediction towards the GPT-4 class. Conversely, in Figure 5.9b, we observe that the values of some of the important features such as *avgIdentifierLength*, and *whiteSpaceRatio* among other features influence the classifier's decision towards the human class. Notably, the values of three important features, *avgLineLength*, *avgFunctionLength*, and

[3] The average depth for Expression nodes in the AST

```


input_prompt = "Please enter a valid Canadian postal code: "
error_msg = "Invalid postal code."

def contains_DFIQOU(s):
    if s == "":
      return False
    elif s[0] in "DFIOQU":
      return True
    else:
      return contains_DFIQOU(s[1:])

def four_errors(s, x):
    starts_with = s.startswith("W") or s.startswith("Z")
    if x <= 0:
        return None
    elif (starts_with or (len(s) < 6) or not(s[3] == " ")
      or contains_DFIQOU(s)):
        if x <= 1:
          print(error_msg)
          return four_errors(s, x - 1)
        else:
          print(error_msg)
          s = input(input_prompt)
          return four_errors(s, x - 1)
    alpha_pos = (s[0].isalpha() and s[5].isalpha())
    number_pos = (s[1].isnumeric() and s[4].isnumeric() and
                  s[6].isnumeric())
    if not(alpha_pos and number_pos):
      if x <= 1:
        print(error_msg)
        return four_errors(s, x - 1)
      else:
        print(error_msg)
        s = input(input_prompt)
        return four_errors(s, x - 1)
    else:
      return sum_of_postals(s, 0)

def sum_of_postals(s, pos):
    if pos == 6:
      return 1
    if s[0] == " ":
      return sum_of_postals(s[1:], pos+1)
    elif s[0].isnumeric():
      return int(s[0])*((1000)**(6-pos)) + sum_of_postals(s[1:], pos+1)
    else:
      alph_position = ’ABCDEFGHIJKLMNOPQRSTUVWXYZ’.index(s[0]) + 1
      return alph_position*((1000)**(6-pos)) + sum_of_postals(s[1:], pos+1)


def postal_code():
    s = input(input_prompt)
    return four_errors(s, 4)
```

(a) Human code

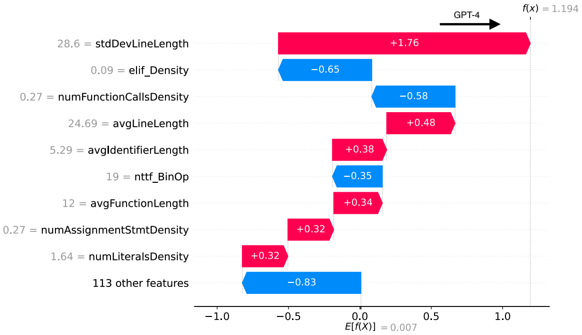


(b) SHAP waterfall plot showing how features impact the model's decision for this code

Figure 5.7: Code classified as human-authored with a low probability of 0.56

```

import math

def feels_like(temp, wind, hum):
  vapour = 6.112 * math.pow(10, (7.5*temp) / (237.7 + temp)) * (hum/100)
  windchill = 13.12 + 0.6125*temp - 11.37 * math.pow(wind, 0.16) + 0.3965*temp*math.pow(wind, 0.16)
  humidex = temp + (5/9)*(vapour - 10)

  if temp >= 15 and humidex == temp + 1:
    return humidex

  elif temp < 15 and windchill == temp - 1:
    return windchill

  else:
    return temp
```

(a) GPT-4 generated code

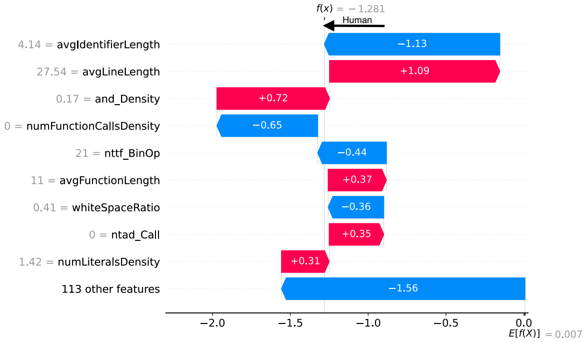


(b) SHAP waterfall plot showing how features impact the model's decision for this code

Figure 5.8: Code classified as GPT-4 generated with an average probability of 0.77

```

def feels_like(temp,wind,hum):
  W = wind ** 0.16
  V = 6.112 * 10 ** (7.5 * temp / (237.7 + temp)) * hum / 100
  windchill = 13.12 + 0.6125 * temp - 11.37 * W + 0.3965 * temp * W
  humidex = temp + 5 / 9 * (V - 10)
  if temp >= 15 and humidex >= temp + 1:
    return humidex
  if temp < 15 and windchill <= temp - 1:
    return windchill
  else:
    return temp
  pass
```

(a) Human code



(b) SHAP waterfall plot showing how features impact the model's decision for this code

Figure 5.9: Code classified as human-authored with an average probability of 0.78

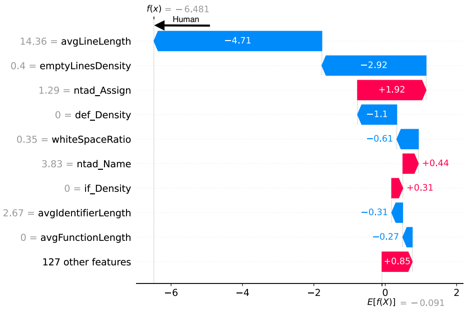
Chapter 6

Discussion

In this chapter, based on our findings, we make observations about our classifier, examining why it correctly and incorrectly predicts solutions in specific programs. Understanding the reason behind its performance on a specific program is crucial for humans to make a final decision about whether it is human-authored or GPT-4 generated. We also investigate the reasons behind the differences in results on CodeChef and programming assignments datasets.

```python
n, m = map(int, input().split())
mi = 2
ma = n + m
ans = [1 for i in range(ma + 1)]

for i in range(2, int(ma**0.5) + 1):
    for j in range(i + i, ma + 1, i):
        ans[j] = 0
ans[0] = 0
ans[1] = 0
print(ans.count(1))



```

(a) Human code

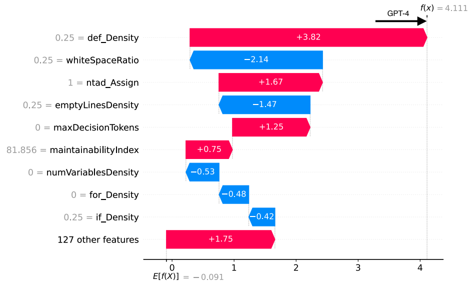


(b) SHAP waterfall plot showing how features impact the model's decision for this code

Figure 6.1: Correctly predicted human code

```python
def solve(n, m):
    return 1 if min(n, m) > 1 else 2

n, m = map(int, input().strip().split())
print(solve(n, m))
```

(a) GPT-4 generated code



(b) SHAP waterfall plot showing how features impact the model's decision for this code

Figure 6.2: Correctly predicted GPT-4 generated code.

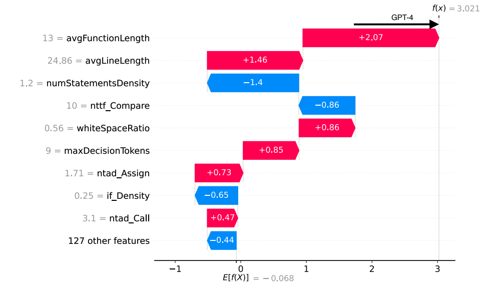
6.1 Correctly Predicted Solution

Figures 6.1 and 6.2 present examples where our model correctly distinguishes between human-authored and GPT-4 generated code. By investigating the SHAP waterfall plot[1] for each example, we gain insights into the model's decision-making for individual predictions, in contrast to the important features in Figure 4.1 which suggests insights into the model's overall decision-making process. The waterfall plot presents the model's expected value, with each row representing how each feature contributes positively (red) towards a prediction of GPT-4 generated code or negatively (blue) towards a prediction of human-authored code. For example, Figure 6.1 shows a correctly predicted human-authored code fragment and its corresponding SHAP waterfall plot. This plot provides a local explanation of the classifier, visually depicting the key features that influence individual predictions. At the bottom of the plot in Figure 6.1b, the model's prediction begins with a base value depicting the cumulative effect of 127 features whose contributions are relatively minor and thus aggregated in the plot. The features' contributions are ranked in ascending order of SHAP values. For example, the *ntad_Assign* feature, representing the AST nodes for assignment operations, has the greatest impact towards the GPT-4 class, with its value of 1.29. The value of this feature, in addition to the value of other features with red arrows in the plot, is indicative of the GPT-4 generated class. However, the most impactful feature is *avgLineLength*, with a value of 14.36 and SHAP value of -4.71; this feature in addition to the value of other features with blue arrows in the plot influences the model's decision towards its final prediction, the human class. Thus, code having an *avgLineLength* (the

[1] https://shap.readthedocs.io/en/latest/example_notebooks/api_examples/plots/waterfall.html

```
def getCount(h,m,i) :
    h=int(h)
    m=int(m)
    h1=0
    lst=[11,22,33,44,55,66,77,88,99]
    while(h1<h) :
        for m1 in range(0,m) :
            if h1<10 :
                if(m1<10 and h1==m1) : count[i]+=1
                if (m1 in lst and m1%10==h1) : count[i]+=1
            else :
                if(m1 in lst and h1==m1) : count[i]+=1
                if(h1 in lst and h1%10==m1) : count[i]+=1
        h1+=1
t=int(input())
count=[0]*t
for i in range(0,t) :
    h,m=input().split()
    getCount(h,m,i)

for i in count : print(i)
```

(a) Human code



(b) SHAP waterfall plot showing how features impact the model's decision for this code

Figure 6.3: Human code predicted incorrectly as GPT-4 generated code

average length of characters in each line) of 14.36, *emptyLinesDensity*, (i.e. the ratio of empty lines to sloc) of 0.4, *def_Density* (i.e. term frequency of the def keyword normalized by sloc) of 0, *whiteSpaceRatio* (i.e. the ratio of whitespace to non-whitespace characters) of 0.35, *avgIdentifierLength* (i.e. the average length of identifier names) of 2.67 and no function is likely human-authored, taking into consideration the other features.

Figure 6.2 shows a correctly predicted GPT-4 generated code fragment and its corresponding SHAP waterfall plot. In Figure 6.2b, the most important feature with a value of 0.25 is the *def_Density*. The value of this feature alongside the value of other features with red arrows influences the model's decision towards the final prediction, the GPT-4 class. This suggests that code having *def_Density* of 0.25, *ntad_Assign* of 1, *maxDecisionTokens* of 0, *maintainabilityIndex* of 81.856 is likely GPT-4 generated, taking into consideration other features. *MaintainabilityIndex* is a feature that isn't observed by looking at the code, so it is an interesting find that this complexity metric could potentially influence the classifier's decision.

These findings underscore the model's ability to correctly predict solutions based on distinctive features, shedding light on the significance of specific code characteristics in the classification process.

6.2 Incorrectly Predicted Solution

Figure 6.3 presents an incorrectly predicted GPT-4 generated code and its corresponding SHAP waterfall plot. This case is concerning, as the model incorrectly guesses that a human-authored code is GPT-4 generated. Although a false negative (predicting GPT-4 generated code as human-authored) is bad, we especially want to avoid false positives because we do not want to unjustly accuse someone of presenting GPT-4 generated code when they have authored their code. However, all classification techniques are bound to have some false positives. We hope that our choice of an explainable model helps educators look at a prediction's SHAP waterfall plot and understand the reasons behind the prediction before making a final decision as to whether to accuse someone of presenting AI-generated code.

In Figure 6.3b, the two top features that influence the model's prediction towards the GPT-4 generated class are related to the length of the code (i.e., *avgFunctionLength*, *avgLineLength*) and have values of 13 and 24.86, respectively. In the case of both *avgLineLength* and *avgFunctionLength*, their values are high compared to the feature values of other observations within the dataset, and features with higher *avgLineLength* and *avgFunctionLength* tend to drive the model's prediction towards the GPT-4 generated class.

These observations explain why the model predicted this code as GPT-4 generated, highlighting the challenges in accurately distinguishing certain code characteristics and the potential consequences of false positives.

6.3 Differences in Results on CodeChef vs. Programming Assignments Datasets

The performance of the classifier built with 140 features extracted from CodeChef solutions, as shown in Table 4.1 shows an F1-Score of 0.91. In contrast, the classifier constructed using 119 features from 2019 submissions, shown in Table 5.4, achieved a lower F1-Score of 0.69. Comparing these results, we see a notable decline in performance when the classifier is adapted to programming assignments.

We identify two potential factors contributing to this performance gap. First, the dataset from the 2019 submissions is considerably smaller, containing only 188 samples compared to CodeChef's 1596. Second, and perhaps the more important factor, is the differences in styling requirements between the CodeChef solutions and programming assignments. Unlike CodeChef where adherence to standard style guides is not specified, programming assignments often demand strict compliance with specific style guides. Additionally, these assignments include certain constraints which may influence students' coding styles. For example, some questions prohibit recursion. We included these style guides and additional restrictions in the prompt, which we believe could lead to similar coding styles between students and GPT-4, thereby impacting the classifier's performance.

Chapter 7

Threat to Validity

We break down the threats into two parts, external and construct.

7.1 External Threat to Validity

These threats relate to the ability to generalize based on our results. In this study, we conducted an empirical investigation on the competitive programming platform CodeChef for human-authored code and utilized GPT-4 for AI-generated code. We then extended our approach to introductory programming assignments. A concern is since we only generate code using GPT-4, the generated code may not be representative of code by other AI assistants. However, compared to other tools, GPT-4 is the most popular AI assistant and should represent real-world usage. In future work, we aim to expand to other programming languages and adapt to a broader range of AI coding assistants.

7.2 Construct Validity

These threats relate to the degree to which our measurements are captured. Regarding the correctness of AI-generated code, we extract public test cases from CodeChef, which do not include private tests for the problems. We decided not to pursue direct submissions of AI-generated code to CodeChef, as this would violate ethical guidelines to submit AI-generated code as a human solution. This is not applicable to the programming assignments as we were not provided with public tests.

Chapter 8

Conclusion

The advent of AI assistants has introduced a new form of academic dishonesty, where students submit AI-generated code as their work. In this thesis, we investigated the impact of using code stylometry features to differentiate between human-authored and GPT-4 generated code, focusing on submissions from CodeChef and GPT-4 generated solutions in Python. The findings demonstrate our approach's promise. Our classifier achieved an F1-score and AUC-ROC score of 0.91, highlighting its potential as a preliminary tool for identifying AI-generated code. Moreover, we identified several key distinguishing features, with the average line length as the most important feature. By providing a means to identify GPT-4 generated code, our study contributes to the ongoing discourse on the use and regulation of AI assistance in coding tasks.

Extending our analysis to assignments from an introductory programming course, our classifier achieved an F1-score of 0.69 and an AUC-ROC score of 0.73. Subsequently, we evaluated this classifier on assignments submitted post-release of Copilot and ChatGPT in 2023 revealing that 13 out of 54 submissions were GPT-4 generated based on a relatively small dataset (trained on 80 submissions and evaluated on 54). With a false positive rate of 13% and a false negative rate of 41% observed during the classifier's evaluation on 2020 submissions, it can be estimated that 20% to 33% of these 2023 submissions may have used AI assistants. This highlights the growing influence of AI assistance in academic settings, necessitating awareness and proactive measures from educators.

Future Work. In the future, we could evaluate our hypothesis that our approach can perform as well on Python programs from other sources such as other competitive programming platforms and programming assignments from different courses. We could also extend our study to include other programming languages to enhance the generalizability of our findings. Additionally, future research could evaluate the effectiveness of our approach in identifying AI-generated code that has been intentionally modified post-generation or through prompt engineering. Such investigation would provide valuable insights into the robustness and shortcomings of our detection methods when AI-generated code is deliberately altered to evade identification. These avenues for further exploration would contribute to a deeper understanding of the capabilities and limitations of code stylometry in distinguishing between AI-generated and human-authored code.

References

[1] Simran Aggarwal. Software code analysis using ensemble learning techniques. In *Proceedings of the 1st International Conference on Advanced Information Science and System*, AISS '19, pages 1–7, New York, NY, USA, January 2020. Association for Computing Machinery.

[2] Alex Aiken. Moss - a system for detecting software similarity. https://theory.stanford.edu/~aiken/moss/.

[3] Ibrahim Albluwi. Plagiarism in Programming Assessments: A Systematic Review. *ACM Transactions on Computing Education*, 20(1), 2019.

[4] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. Source Code Authorship Attribution Using Long Short-Term Memory Based Networks. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, volume 10492, pages 65–82. Springer International Publishing, Cham, 2017. Series Title: Lecture Notes in Computer Science.

[5] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard - or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 500–506, New York, NY, USA, 2023. Association for Computing Machinery.

[6] H. L. Berghel and D. L. Sallach. Measurements of program similarity in identical task environments. *SIGPLAN Not.*, 19(8):65–76, aug 1984.

[7] Jason R Briggs. *Python for kids: A playful introduction to programming*. no starch press, 2012.

[8] Sufiyan Bukhari, Benjamin Tan, and Lorenzo De Carli. Distinguishing ai- and human-generated code: A case study. In *Proceedings of the 2023 Workshop on Software Supply*

Chain Offensive Research and Ecosystem Defenses, SCORED '23, page 17–25, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270, Washington, D.C., August 2015. USENIX Association.

[10] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[11] Robert Clarke and Thomas Lancaster. Commercial aspects of contract cheating. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'13)*, page 219–224, 2013.

[12] Codequiry. Codequiry. https://codequiry.com/.

[13] Coderbyte. Detect candidates that cheat with ai / chatgpt, 2021.

[14] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, Aug 1994.

[15] Copyleaks. Copyleaks. https://copyleaks.com/.

[16] Edwin Dauber, Aylin Caliskan, Richard Harang, and Rachel Greenstadt. Git blame who? stylistic authorship attribution of small, incomplete source code fragments. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 356–357, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Thomas G Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, 10(7):1895–1923, 1998.

[18] John L. Donaldson, Ann-Marie Lancaster, and Paula H. Sposato. A plagiarism detection system. In *Proceedings of the twelfth SIGCSE technical symposium on Computer science education - SIGCSE '81*, pages 21–25, St. Louis, Missouri, United States, 1981. ACM Press.

[19] Wenyuan Dong, Zhiyong Feng, Hua Wei, and Hong Luo. A Novel Code Stylometry-based Code Clone Detection Strategy. In *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pages 1516–1521, June 2020. ISSN: 2376-6506.

[20] Mojtaba Eshghie, Cyrille Artho, and Dilian Gurov. Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, EASE '21, pages 305–312, New York, NY, USA, June 2021. Association for Computing Machinery.

[21] J.A.W. Faidhi and S.K. Robinson. An empirical approach for detecting program similarity and plagiarism within a university programming environment. *Computers & Education*, 11(1):11–19, 1987.

[22] James Finnie-Ansley, Paul Denny, Brett Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, ACE '22, pages 10–19, 02 2022.

[23] Sophia F. Frankel and Krishnendu Ghosh. Machine learning approaches for authorship attribution using source code stylometry. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 3298–3304, 2021.

[24] Github. Copilot: Your AI Pair Programmer. https://github.com/features/copilot, Oct 2021. [Online; accessed 9-October-2023].

[25] Sam Grier. A tool that detects plagiarism in Pascal programs. *ACM SIGCSE Bulletin*, 13(1):15–20, February 1981. Number: 1.

[26] HackerRank. Hackerrank launches ai-powered plagiarism detection, 2021.

[27] M. H. Halstead. Natural laws controlling algorithm structure? *ACM SIGPLAN Notices*, 7(2):19–26, February 1972. Number: 2.

[28] Pengnan Hao, Zhen Li, Cui Liu, Yu Wen, and Fanming Liu. Towards Improving Multiple Authorship Attribution of Source Code. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 516–526, December 2022. ISSN: 2693-9177.

[29] Oseremen Joy Idialu, Noble Saji Mathews, Rungroj Maipradit, Joanne M Atlee, and Mei Nagappan. Whodunit: Classifying code as human authored or gpt-4 generated–a case study on codechef problems. *arXiv preprint arXiv:2403.04013*, 2024.

[30] Cheng Jiao, Neel R Edupuganti, Parth A Patel, Tommy Bui, Veeral Sheth, and Neel Edupuganti. Evaluating the artificial intelligence performance growth in ophthalmic knowledge. *Cureus*, 15(9), 2023.

[31] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. Code Authorship Attribution: Methods and Challenges. *ACM Computing Surveys*, 52(1):3:1–3:36, February 2019.

[32] Gurpreet Kaur, Yasir Malik, Hamman Samuel, and Fehmi Jaafar. Detecting Blind Cross-Site Scripting Attacks Using Machine Learning. In *Proceedings of the 2018 International Conference on Signal Processing and Machine Learning*, pages 22–25, Shanghai China, November 2018. ACM.

[33] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. Studying the Effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'23)*, 2023.

[34] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. Discovering software vulnerabilities using data-flow analysis and machine learning. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES '18, pages 1–10, New York, NY, USA, August 2018. Association for Computing Machinery.

[35] Lov Kumar, Shashank Mouli Satapathy, and Lalita Bhanu Murthy. Method Level Refactoring Prediction on Five Open Source Java Projects using Machine Learning Techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference)*, ISEC'19, pages 1–10, New York, NY, USA, February 2019. Association for Computing Machinery.

[36] Wanda M. Kunkle and Robert B. Allen. The impact of different teaching approaches and languages on student learning of introductory programming concepts. *ACM Trans. Comput. Educ.*, 16(1), jan 2016.

[37] Sam Lau and Philip Guo. From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools Such as ChatGPT and GitHub Copilot. In *Proceedings of the ACM Conference on International Computing Education Research (ICER'23) - Volume 1*, page 106–121, 2023.

[38] Ronald J. Leach. Using metrics to evaluate student programs. *SIGCSE Bull.*, 27(2):41–43, jun 1995.

[39] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *CoRR*, abs/1705.07874, 2017.

[40] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. Number: 4 Conference Name: IEEE Transactions on Software Engineering.

[41] Aravind Nair, Karl Meinke, and Sigrid Eldh. Leveraging mutants for automatic prediction of metamorphic relations using machine learning. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2019, pages 1–6, New York, NY, USA, August 2019. Association for Computing Machinery.

[42] P. W. Oman and C. R. Cook. Programming style authorship analysis. In *Proceedings of the seventeenth annual ACM conference on Computer science : Computing trends in the 1990's Computing trends in the 1990's - CSC '89*, pages 320–326, Louisville, Kentucky, United States, 1989. ACM Press.

[43] OpenAI. Introducing ChatGPT. https://openai.com/blog/chatgpt, November 2022. [Online; accessed 9-October-2023].

[44] Julia Opgen-Rhein, Bastian Küppers, and Ulrik Schroeder. Requirements for Author Verification in Electronic Computer Science Exams:. In *Proceedings of the 11th International Conference on Computer Supported Education*, pages 432–439, Heraklion, Crete, Greece, 2019. SCITEPRESS - Science and Technology Publications.

[45] Manjula Peiris and James H. Hill. Towards detecting software performance antipatterns using classification techniques. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–4, February 2014.

[46] Ben Puryear and Gina Sprint. Github copilot in the classroom: learning to code with ai assistance. *Journal of Computing Sciences in Colleges*, 38(1):37–47, 2022.

[47] Amazon Web Services. What is CodeWhisperer? https://docs.aws.amazon.com/codewhisperer/latest/userguide/what-is-cwspr.html, 2023. [Online; accessed 9-October-2023].

[48] Zhiyu Sun, Fang Peng, Junrui Guan, and Yanchun Sun. An approach to helping developers learn open source projects based on machine learning. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, Internetware '19, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Irene Tollin, Francesca Arcelli Fontana, Marco Zanoni, and Riccardo Roveda. Change Prediction through Coding Rules Violations. In *Proceedings of the 21st International*

Conference on Evaluation and Assessment in Software Engineering, EASE '17, pages 61–64, New York, NY, USA, June 2017. Association for Computing Machinery.

[50] Farhan Ullah, Sohail Jabbar, and Fadi Al-Turjman. Programmers' de-anonymization using a hybrid approach of abstract syntax tree and deep learning. *Technological Forecasting and Social Change*, 159:120186, October 2020.

[51] Nickolay Viuginov, Petr Grachev, and Andrey Filchenkov. A Machine Learning Based Plagiarism Detection in Source Code. In *Proceedings of the 2020 3rd International Conference on Algorithms, Computing and Artificial Intelligence*, ACAI '20, pages 1–6, New York, NY, USA, March 2021. Association for Computing Machinery.