AI or Human: Detection Techniques for Uncovering the Source of Code - Technical Report

Oedingen Marc
Computer Science and Engineering
TH Köln
Gummersbach, Germany
Email: Marc.Oedingen@th-koeln.de

Hammer Maximilian
Computer Science and Engineering
TH Köln
Gummersbach, Germany
Email: Maximilian.Hammer@smail.th-koeln.de

***Abstract*—More recently, Large Language Models have achieved remarkable results in code generation. With their ever-increasing performance, the distinction between human- and AI-generated code poses a complex and opaque problem. In this era of rapid technological advancement, it is becoming increasingly essential to understand the extent and nuances of the impact that AI technologies, particularly Large Language Models, have on code generation. The potential for fraud in the context of higher education through the use of AI technologies is highly likely to occur. This paper aims to investigate this problem using unsupervised and supervised classification models and meticulous analysis of their performance and explainability. Our research introduces a novel methodology combining embeddings with Deep Neural Networks and Gaussian Mixture Models to distinguish between AI and human-generated code. Achieving accuracies over 98%, our black-box models provide deep insights by analyzing code snippets both holistically and token-wise. Additionally, we present white-box models to elucidate key differences between the code sources, enhancing the explainability and transparency of our approach. This study is crucial in understanding and mitigating the potential risks and nuances associated with the use of AI in code generation, particularly in the context of higher education, where the likelihood of fraud is present.**

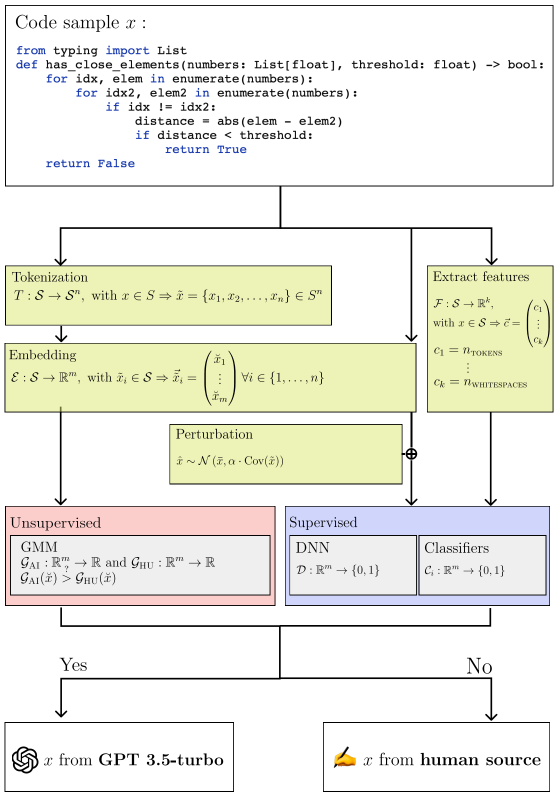
***Index Terms*—AI, Machine Learning, Code Detection, ChatGPT**

I. Introduction

A. *Motivation*

Presently, the world is looking ambivalently at the development and opportunities of powerful Large Language Models (**LLMs**). On the one hand, such models can execute complex tasks and augment human productivity due to their enhanced performance in various areas [1, 2]. On the other hand, these models can be misused for malicious purposes, such as generating deceptive articles or cheating in educational institutions and other competitive environments [3, 4, 5].

The inherently opaque nature of these black-box models, combined with the difficulty of distinguishing between human- and AI-generated content, poses a problem that can make it challenging to trust these models [6]. Earlier research has extensively focused on detecting natural language text content generated by LLMs [7, 8, 9]. However, detecting AI-generated code is an equally important and relatively unexplored area of research. As LLMs are utilized more often in the field of software development [2], the ability to distinguish between human- and AI-generated code becomes increasingly paramount. Thereby, it is not exclusively about the distinction but also the implications, such as the code's trustworthiness, efficiency, and security. Moreover, the rapid development and improvement of AI models may lead to an arms race where detection techniques must continuously evolve to stay ahead of the curve [4, 5, 10].


Code sample x :
Tokenization
$T : \mathcal{S} \rightarrow \mathcal{S}^n$, with $x \in S \Rightarrow \tilde{x} = \{x_1, x_2, \dots, x_n\} \in S^n$
Embedding
$\mathcal{E} : \mathcal{S} \rightarrow \mathbb{R}^m$, with $\tilde{x}_i \in \mathcal{S} \Rightarrow \vec{\tilde{x}}_i = \begin{pmatrix} \breve{x}_1 \\ \vdots \\ \breve{x}_m \end{pmatrix} \forall i \in \{1, \dots, n\}$
Extract features
$\mathcal{F} : \mathcal{S} \rightarrow \mathbb{R}^k$, with $x \in \mathcal{S} \Rightarrow \vec{c} = \begin{pmatrix} c_1 \\ \vdots \\ c_k \end{pmatrix}$
$c_1 = n_{\text{TOKENS}}$
$c_k = n_{\text{WHITESPACES}}$
Perturbation
$\hat{x} \sim \mathcal{N}(\bar{x}, \alpha \cdot \text{Cov}(\tilde{x}))$
Unsupervised
GMM
$\mathcal{G}_{\text{AI}} : \mathbb{R}^m_? \rightarrow \mathbb{R}$ and $\mathcal{G}_{\text{HU}} : \mathbb{R}^m \rightarrow \mathbb{R}$
$\mathcal{G}_{\text{AI}}(\breve{x}) > \mathcal{G}_{\text{HU}}(\breve{x})$
Supervised
DNN
$\mathcal{D} : \mathbb{R}^m \rightarrow \{0, 1\}$
Classifiers
$\mathcal{C}_i : \mathbb{R}^m \rightarrow \{0, 1\}$
Yes
No
x from **GPT 3.5-turbo**
x from **human source**

```
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True
    return False
```

Fig. 1. Flow chart of methodology for classifying code samples

Earlier research showed that using LLMs trained on code can lead to security vulnerabilities and 40% of the generated code failing to solve the given task [11]. Contrastingly, the usage of LLMs can also lead to a significant increase in productivity and efficiency, as reported in [2]. This dual-edged nature of LLMs necessitates a balanced approach. Harnessing the potential benefits of LLMs while mitigating risks is the key. However, the ability to distinguish between human- and AI-generated code is a prerequisite for the acceptance and trustworthiness of LLMs in the field of software development, setting the foundation for a more secure and efficient future in tech.

In essence, the ability to differentiate between human- and AI-generated code is not just a technical challenge; it is fundamental for the future trajectory of software development. Ensuring the authenticity of code is especially crucial in academic environments, where the integrity of research and educational outcomes is paramount. Fraudulent or AI-generated submissions can undermine the foundation of academic pursuits, leading to a loss of trust in research findings and educational qualifications [6]. Moreover, in the context of examinations, robust fraud detection is essential to prevent cheating, ensuring the assessments accurately reflect the student's capabilities and do not check the output of a luck-based prompt due to the stochastic decision-making of Transformer Model (**TM**)-based [12] LLMs during inference [13]. As the boundaries of what AI can achieve expand [14], our approach to understanding, managing, and integrating these capabilities into our societal fabric, including academic settings, will determine our success in the AI-augmented era [1].

B. Problem Introduction

In this rapidly evolving landscape for AI-supported software development, distinguishing between human-generated and AI-generated code has become increasingly important. This paper delves deep into this challenge, offering a comprehensive overview of state-of-the-art methods and proposing novel strategies to tackle this problem.

Central to our methodology is a reduction of complexity: the intricate task of discerning AI-generated code is distilled into a fundamental binary classification dilemma. Specifically, given a code snippet $x \in \mathcal{C}$ as input, we aim for a function $f : \mathcal{C} \rightarrow \mathcal{Y} = \{0, 1\}$, which indicates whether the codes' origin is human $\{0\}$ or artificial $\{1\}$. Before delving into the classification, however, an immediate challenge arises: establishing a common representation for these code snippets. This uniformity is crucial to ensure that the input is effectively interpreted and processed by the models during inference.

The simplification of binary classification offers several benefits. Firstly, the binary classification domain is well-charted, with many models that can potentially be leveraged for this task [15]. Additionally, many of these models possess a degree of interpretability, facilitating a clearer understanding of their decision-making or the underlying differences between human- and AI-generated code [16].

Nevertheless, there are many obstacles to face before classification is applicable. Foremost among these is the trade-off between the volume and quality of data. For a model to truly generalize in the huge field of software development across many coding snippets, it requires training on vast amounts of data. However, volume alone is insufficient for a model. The data's quality is paramount, ensuring meaningful and discriminating features are present within the code snippets. An ideal dataset, then, would consist of code snippets that satisfy a variety of test cases to guarantee their syntactical and semantical correctness for a given task. Moreover, to overcome the pitfalls of biased data, our emphasis is squarely on code snippets produced prior to the proliferation of AI in code generation.

However, a significant blocker emerges from a stark paucity of publicly available AI-generated solutions to match our criteria mentioned before. This scarcity underscores the necessity to find and generate solutions that can serve as fitting data sources for our classifier.

C. Research Questions

We formulate the following research questions based on the problem introduction and the need to obtain detection techniques for AI-generated code. In addition, we provide our hypotheses regarding the questions we want to answer with this work.

RQ1*: Can we distinguish between human- and AI-generated code?*

RQ2*: To what extent can we explain the difference between human- and AI-generated code?*

- $\boldsymbol{H_{1.0}}$: Programmers give their code a unique style, and these styles are different from AI-generated code.

- $\boldsymbol{H_{2.0}}$: There are differences between human- and AI-generated code
- $\boldsymbol{H_{2.1}}$: There are syntactical differences between human- and AI-generated code that can be used to differentiate between them.

Upon rigorous scrutiny of the posed research questions, an apparent paradox emerges. LLMs have been trained using human-generated code. Consequently, the question arises: Does human-generated code diverge from its human counterpart? As previously postulated, see hypothesis $\boldsymbol{H_{1.0}}$, we hypothesize that LLMs demonstrate learning trajectories similar to individual humans. Throughout the learning process, humans and machines are exposed to many code snippets, each encompassing distinct stylistic elements. They subsequently develop and refine their unique coding style, i.e., by using consistent variable naming conventions, commenting patterns, code formatting, or selecting specific algorithms for given scenarios [17]. However, given the vast amount of code the machine has seen during training, it is anticipated to adopt a more generalized coding style. Thus, identifiable discrepancies between machine-generated code snippets and individual human-authored code are to be expected.

D. Structure

In the first section I, we state our motivation grounded on the development of LLMs in software development. We also introduce the central problem under investigation in this paper and the research questions it engenders. The second section II provides a literature review, highlighting methods successfully applied in previous works to tackle a problem similar to the one we are addressing, which relates to the detection of AI in natural language. In the third section III, we present our methodology by offering a comprehensive overview of used algorithms with their mathematical foundations essential to distinguish between human- and AI-generated code. Section four IV describes the experimental setup, including the collection and preprocessing of data for algorithm application. Further, section five V is dedicated to presenting our findings. We showcase our results and analyze them thoroughly to address the research questions, offering deep insights into the problem. Section six VI discusses the main advantages and shows the limitations of our experiment, both technically and financially. Lastly, we conclude our main findings and provide insights for future work in section seven VII.

II. Related Work

In the domain of AI, fraud detection is a well-established research area. Numerous methodologies addressing this challenge have been documented in the literature [18, 19]. However, most of these methodologies focus on AI-generated textual content rather than code [7, 8, 20]. Nevertheless, findings from these studies remain relevant, as techniques and essential properties can be transferred to fraud detection in code. Models designed for detecting machine-generated text can be classified into two categories: (**1**) Conventional statistical methodologies and (**2**) Classification executed by an LLM in either an unsupervised or supervised manner.

A. Statistical Methods

One of the most successful approaches to distinguishing between human-generated and AI-generated textual content is given by the DetectGPT model [7]. Designed for Zero-Shot text detection, it exploits the inherent probability curvature of the log-likelihood function of an LLM, denoted p_θ, and asserts that p_θ lies in regions of negative curvature of the p_θ log-likelihood function, as opposed to human text. Given some text samples from the reference model, represented by $x \sim p_\theta$, the authors introduce a perturbation function $q(\cdot|x)$. This function generates a perturbed version, $\tilde{x}$, of the original sample x while preserving its semantic meaning. Colloquially, this perturbation can be regarded as a human rephrasing the content of the sample. The interference discrepancy serves as a key metric in assessing the extent of perturbation:

$$d(x, p_\theta, q) \triangleq \log\left(p_\theta\left(x\right)\right) - \mathbb{E}_{\tilde{x}\sim q(\cdot|x)} \log\left(p_\theta\left(\tilde{x}\right)\right).$$

The core hypothesis posits that $d(x, p_\theta, q)$ approaches zero for human-generated samples, while for AI-generated samples $x \sim p_\theta$, $d(x, p_\theta, q)$ exhibits a high positive probability across all instances of x. Thus, exceeding some predefined threshold $\epsilon \in \mathbb{R}$ as hyperparameter suggests that the sample probably belongs to the reference model and vice versa. This becomes particularly interesting when transferring to our posed problem: Pertubing code is more challenging than text due to syntactic and semantic constraints, where small changes can break the code entirely. However, code perturbations are feasible by focusing on creating functionally equivalent versions or adjusting aspects like styling or variable naming. Particular attention must be paid to ensure that the intended functionality is provided while sufficient variation for robustness testing exists. In the paper, the T5 model [21], another LLM, was used to generate the perturbations for a scalable and automated procedure.

Another machine-generated content detector model, called Giant Language Model Test Room (**GLTR**), was proposed in 2019 [8]. An empirical study showed that the human detection rate of fake text was improved from 54% to 72%. Their study uses BERT [22] and GPT-2 [23] as backend language models. The authors apply three different statistical tests to determine the probability of the occurrence of a word in a sequence by color encoding it concerning the tok-k metric. Given a sequence of words, the top-k metrics determine the probability of the next token, with $p\left(x_i|x_{1,\ldots,i-1}\right)$, selecting k words with the highest values from the probability distribution of the model. Moreover, these tests with a deterministic setting are defined as: (**1**) The probability of the word $p\left(x_i = \tilde{x}_i|x_{1,\ldots,i-1}\right)$, (**2**) the absolute rank $p_{det}\left(x_i|x_{1,\ldots,x_{i-1}}\right)$ and (**3**) the entropy of the predicted distribution $-\sum_w p_{det}\left(x_i = w|x_{1,\ldots,i-1}\right)\log\left(p_{det}\left(x_i = w|x_{1,\ldots,i-1}\right)\right)$. This approach works well in a white-box scenario where the probability distribution p_θ is known but lacks confidence in black-box scenarios.

Currently, many common Supervised Learning (**SL**) models like Logistic Regression, Decision Trees, Support Vector Machines, Random Forests, and Neural Networks for text content explicitly generated by ChatGPT were investigated [9]. The training and evaluation data set consists of student-written and ChatGPT-written in an academic setting. This study shows the potential of using simple classifier models to detect AI content [9]. To extract features, they apply the Term Frequency-Inverse Document Frequency (**TF-IDF**) technique and vectorize the resulting values. In the experiment, the Random Forest classifier performs best according to accuracy, but it must be mentioned that they only evaluated ChatGPT-generated content. This is one of the few research papers that also targets the detection of programming code, even though it is not explicitly stated what programming language they evaluated.

B. Classification with Language Model

The second category of classification methodologies pertains to classification conducted by a language model. Solaiman et al. [20] proposed a baseline technique wherein a language model is utilized to assess its log probability and determine a threshold for classification decisions in a zero-shot approach. Nonetheless, this approach exhibits inferior performance compared to conventional statistical methods. Similar to this zero-shot classification, a language model can also be used as a foundation and fine-tuned with a separate classifier model attached. Chen et al. trained a frozen RoBERTa model with a Multilayer Perceptron (**MLP**) neural network, which they call *GPT-Sentinel* to classify ChatGPT-generated text [24]. The RoBERTa model was trained to extract relevant features from the input text, followed by an MLP to distinguish human and AI content. They achieve an accuracy of over 97% on the test data set. However, their approach relies heavily on the RoBERTa model, which is mainly trained on an English language corpus only [24].

III. Algorithms and Methodology

In this section, we will present fundamental algorithms and describe the approaches we use for the identification of AI-generated code samples. We start by detailing the general prerequisites and preprocessing needed for algorithm application. Subsequently, we explicate our strategies for code detection and briefly delineate the models used for code sample classification. We, respectively, conclude by presenting the evaluation metrics for each model.

A. General Prerequisites

Our primary goal is to develop a model differentiating AI and human-generated code. Therefore, we need a suitable and representative dataset for the application in academic, educational, or other coding-assessment scenarios to train such a model. Code snippets similar to code answers of university exams and assignments are of interest. To the best of our knowledge, there are two main methods in which an LLM such as ChatGPT can be used to create fraudulent solutions: (**1**) Represent the requirement text of a coding problem as a prompt and use the model's pure output, or (**2**) use an existing solution and modify it using ChatGPT. In addition, the solutions generated are usually slightly modified to overcome plagiarism detection. However, in this paper, we only investigate the first method of cheating.

To have comparable code snippets, we decided to sample tasks from multiple online coding task websites that offer task descriptions, human-written solutions, and test cases. These human-written solutions are used as a baseline to compare to the AI-generated code. We use the `gpt-3.5-turbo` model [25] to generate code. This model powers the commonly known application ChatGPT [25, 26] and represents the most-used AI tool by students.

Fundamentally, ChatGPT is a Sequence-to-sequence learning [27] model with an Encoder-Decoder structure based on the Transformer architecture [12]. Due to its positional encoding and Self-Attention mechanism, it can process data in parallel rather than sequentially, unlike previously used models such as Recurrent Neural Networks [28] or Long-Short-Term Memory models [29]. Just limited in the maximum capacity of input tokens, it is capable of capturing long-term dependencies. During inference, the decoder is detached from the encoder and is used solely for the output of further tokens. Hence, the initial prompt is inputted into the decoder, which generates the output sequence token-by-token, one at a time. After generating one token, the input of the previous forward pass is updated with the new token, and the model generates the next token until some termination criteria are met. If terminated, the model awaits the user's subsequent input and appends it to their previous conversation, thus mimicking the user's chat with a Generative Pre-trained Transformer (**GPT**) model. This technique allows us to give individual feedback to the model when it does not meet all test cases, which can increase the probability of a correct answer [25].

Having test cases for the tasks is important to ensure that the generated code is syntactically correct, can be executed, and successfully solves the given task. By only comparing code snippets from humans and AI that pass all test cases, we can ensure that classification is not based on the code's functionality but rather on the code's style. It is also a prophylactic method to prevent bias in the data and, as will be shown later, to exclude outliers in advance. Further, it allows us to show that not only random outputs of the model are compared to functioning human-generated codes but that significant differences exist in a large sample set.

Machine learning models require a uniform data representation to ensure comparability and interpretability. Without embeddings, the inconsistent lengths and discrete nature of textual code samples make it challenging to measure and compare underlying patterns consistently. A shared vector space, provided by the embeddings [30], is essential for models to capture and quantify these differences effectively. However, it is noteworthy that embeddings are vital for constructing black-box models. For white-box models, where interpretability is paramount, one can directly extract numerical features from the data, obviating the need for such embeddings. We can break down code snippets into manageable units by tokenization, allowing for more precise embeddings. Conversely, tokenization choices can significantly impact the quality of the embeddings; overly granular tokenization might result in loss of contextual information, whereas too coarse tokenization could overlook subtle nuances. Therefore, selecting a suitable tokenization strategy is essential to ensure that embeddings capture the richness and complexity of the data.

Subsequently, the refined dataset serves as a foundation for contrasting three distinct embedding methodologies in combination with various classification algorithms. These embeddings then serve as input for the classification algorithms, which aim to categorize these snippets as either human- or AI-generated. Upon evaluating the experimental outcomes, an optimal combination of tokenization, embeddings, and classification techniques can be discerned based on the performance metrics.

B. *Embeddings*

Embeddings are numerical vector representations that capture the essence of words, sentences, or code in a continuous space. By transforming discrete data into continuous vectors, embeddings allow ML models to perform mathematical operations and understand patterns or similarities in the data. In our context, we use the following three models to embed all code snippets:

TF-IDF [31]: Term Frequency-Inverse Document Frequency refers to a statistical method used to quantify the significance of a word in a document relative to a collection of documents. As its essence, TF-IDF captures two primary components: (**1**) Term frequency (TF), which is the number of times a word appears in a document, and (**2**) the inverse document frequency, which reduces the weight of words that are common across multiple documents. Formally, TF-IDF is defined as:

$$\begin{aligned}&\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t),\\ &\qquad\text{with } \text{TF}(t,d) = \frac{N_{t,d}}{N_d} \text{ and } \text{IDF}(t) = \log\left(\frac{N}{N_t}\right),\end{aligned}$$

where N is the number of documents, $N_{t,d}$ the number of times term t appears in document d, N_d the number of terms in document d and N_t the number of documents that include term t. The score is high for frequent words in a specific document but not across all, highlighting their importance and vice versa. To employ this embedding, tokenizing the given text or code snippet to ascertain frequencies is essential. Consequently, the embedding captures the relative significance of words in each document as a sparse matrix.

Word2Vec [32]: Word2Vec is a neural network-based technique used to generate dense vector representations of words in a continuous vector space. It fundamentally operates on two architectures: (**1**) Skip-Gram (**SG**), where the model predicts the surrounding context given a word, and (**2**) Continuous Bag of Words (**CBOW**), where the model aims to predict a target word from its surrounding context. Given a sequence of words $w_1, \ldots, w_T$, their objective is a minimization of:

$$\mathcal{L}(\theta) = \frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-c\leq j\leq c\\ j\neq 0}}\begin{cases}\log\left(p\left(w_t|w_{t+j}\right)\right) & \text{for SG}\\ \log\left(p\left(w_{t+j}|w_t\right)\right) & \text{for CBOW}\end{cases}$$

$$\text{with } p\left(w_O|w_I\right) = \frac{\exp\left({v'}_{w_O}^{\intercal} v_{w_I}\right)}{\sum_{w=1}^{W}\exp\left({v'}_{w}^{\intercal} v_{w_I}\right)},$$

where v_w and v'_w denote the input and output vector representation of word $w_i \in \mathcal{V}$ in the sequence of all words in the vocabulary, $W \in \mathbb{N}$ the number of words in that vocabulary $\mathcal{V}$ and $\theta \in \mathbb{R}^n$ the networks parameters. The probability of a word given its context is calculated by the softmax function with $p(w_O|w_I)$. Training the model efficiently involves strategies like Hierarchical Softmax and Negative Sampling to avoid the computational challenges of the softmax over large vocabularies [33].

OpenAI Ada [23]: The OpenAI Ada model is a text embedding model, which is based on a Transformer's pre-trained encoder E [34], initialized with GPT models [26, 35]. Given a set of positive example pairs $\{(x_i, y_i)\}_{i=1}^{N}$, where x_i and y_i are semantically similar, the encoder E maps the inputs x and y to embeddings v_x and v_y, respectively. Their similarity is computed by the cosine similarity, yielding:

$$\begin{aligned}v_x &= E\left([\texttt{SOS}]_x \oplus x \oplus [\texttt{EOS}]_x\right)\\ v_y &= E\left([\texttt{SOS}]_y \oplus y \oplus [\texttt{EOS}]_y\right)\\ \text{and } \text{sim}(x,y) &= \frac{v_x \cdot v_y}{\|v_x\| \cdot \|v_y\|},\end{aligned}$$

where $\oplus$ denotes the operation of string concatenation and $\texttt{EOS}, \texttt{SOS}$ some special tokens, delimiting the sequences. Considering a mini-batch of M examples, in which only one positive example for the inputs exists, $(x_k, x_j) \Rightarrow k = j$, the other $(M-1)$ samples are used as negative samples in training. Thus, the logits for one batch is a $M \times M$ matrix, where each logit is defined as $\hat{y} = \text{sim}(x_i, y_i) \cdot \exp(\tau)$, and τ is a trainable temperature parameter. In essence, only entries on the diagonal of the matrix are considered positive examples. Finally, the loss is calculated as the sum of the cross entropy losses for each row $\mathcal{L}_r$ and column $\mathcal{L}_c$ in M:

$$\begin{aligned}\mathcal{L}(y,\hat{y}) &= \frac{1}{2}\mathcal{L}_r(y,\hat{y}) + \mathcal{L}_c(y,\hat{y})\\ &= \frac{1}{2}\left(-\sum_{i=1}^{M}\sum_{j=1}^{M} y_{i,j}\log\left(\hat{y}_{i,j}\right) + y_{j,i}\log\left(\hat{y}_{j,i}\right)\right).\end{aligned}$$

While this embedding model seems promising, it currently, in contrast to the other introduced embeddings, is subject to the use of an API provided by OpenAI, namely `text-embedding-ada-002` [25], which returns a non-variable dimension of $v_x, v_y \in \mathbb{R}^{1536}$.

Each of these embeddings has its own set of advantages, and the best choice depends on the specific requirements of a task, such as the level of semantic understanding needed, computational resources available, and the nature of the data we are working with. In terms of explainability, TF-IDF is the most transparent model, offering clear insight into how the importance of terms is calculated based on their frequency in documents. On the other hand, Word2Vec and OpenAI Ada, leveraging neural network architectures, operate more as "black boxes" with complex, multi-layered operations that abstract away the underlying mechanics, making them less interpretable compared to TF-IDF, albeit potentially more powerful in capturing subtle semantic relationships.

To avoid further use of computationally intensive LLMs or costly APIs but mimicking a real-world scenario in which humans may modify the generated code, we utilize a method rooted in the statistical properties of the dataset. Specifically, we calculate the covariance matrix $\text{Cov}(X) \in \mathbb{R}^{m \times m}$, where $X \in \mathbb{R}^{n \times m}$ denotes the matrix of all concatenated embedded vectors, n the number of code samples and m the dimension of the embedding. This process involves computing

$$\text{Cov}(X) = \frac{1}{n-1} \left(X - \bar{X}\right)^{\intercal} \left(X - \bar{X}\right),$$

with $\bar{X} \in \mathbb{R}^m$ the vector of mean values for each dimension. We proceed to generate a perturbated dataset by sampling new vectors X_{PER} from a multivariate normal distribution centered at $\bar{X}$ with the covariance matrix scaled by a factor of $\alpha \in \mathbb{R}$:

$$X_{\text{PER}} = \mathcal{N}\left(\bar{X}, \alpha \cdot \text{Cov}(X)\right).$$

Thereby, α serves as a hyperparameter, granting control over the degree of perturbation applied to the original vectors. Optimally, α should be as large as possible without disabling the model to differentiate between human- and AI-generated samples. This approach maintains the core statistical structure of the original dataset and crafts a modified dataset that not only embodies controlled variations but also potentially mirrors the kind of alterations a human might introduce. Further, it enhances the authenticity and applicability of the generated code samples in real-world scenarios while aiming for improved generalization.

C. Supervised Learning methods

Feature extraction and embedding derivation constitute integral components in distinguishing between AI-generated and human-generated code, serving as inputs for classification models. Subsequently, we delineate the SL models employed in this analysis.

Logistic Regression [36]: The naive approach of applying a linear regression model with $\hat{Y}_i = \beta_0 + \beta_1 \cdot X_{i1} + \ldots + \beta_p \cdot X_{np}$, $\beta_i \in \mathbb{R}^p$ and $X_i \in \mathbb{R}^n$, with $i = 1, \ldots, n$, for classification fails, since $0 \leq \hat{Y} \leq 1$ does not hold, which is necessary to represent probabilities. Therefore, a logistic function $\sigma : \mathbb{R} \to [0, 1]$ with $\sigma(z) = \frac{1}{1+\exp(-z)}$ is introduced, mapping any real number to the desired interval. Consequently, applying the logistic function to the right side of the equation yields:

$$p\left(Y_i = 1\right) = \frac{1}{1 + \exp(-\hat{Y}_i)}.$$

If $p\left(Y_i = 1\right) \geq 0.5$, then the sample belongs to class 1, otherwise to class 0. Thus, logistic regression alleviates the shortcomings of linear regression in classification tasks, effectively modeling the probability of a binary outcome and ensuring the predicted values lie in the $[0, 1]$ range, thereby providing a sound basis for probabilistic classification. However, applying the logistic function makes the result no longer interpretable.

Classification and Regression Trees [37]: In graph theory, a Decision Tree is considered an acyclic-directed graph, where decisions are represented as nodes, decision paths as edges, and decision classes as leaves. Considering a candidate split $\theta = (j, t_m)$ consisting of a feature j and threshold t_m, partitioning of data is represented as:

$$Q_m^{left}(\theta) = \{(x, y) | x_j \leq t_m\}; Q_m^{left}(\theta) = Q_m \setminus Q_m^{left}(\theta),$$

where Q_m denotes the data at node m with n_m samples. Objectively, the quality of the split is determined by the minimization of θ by recursively calling $Q_m^{left}(\theta)$ and $Q_m^{right}(\theta)$ until the maximum or desired depth is reached with

$$G(Q_m, \theta) = \frac{n_m^{left}}{n_m} H(Q_m^{left}(\theta)) + \frac{n_m^{right}}{n_m} H(Q_m^{right}(\theta)).$$

In our context, respectively classification, the loss function $\mathcal{L}$ is defined as

$$\mathcal{L}(Q_m) = \sum_k p_{mk}(1 - p_{mk}),$$
$$\text{with } p_{mk} = \frac{1}{n_m} \sum_{y \in Q_m} I(y = k).$$

By splitting features individually in each node, dividing the feature space along one axis, and building a decision rule for this node, the tree offers transparent decision-making as it consists of simple rules. Lastly, decision trees can isolate the most important features through the feature selection process at each node, making them an effective tool for feature selection and understanding the key drivers behind the classifications.

Random Forest [38]: A Random Forest is an ensemble method, i.e. the application of several Decision Trees, and is subject to the idea of bagging. In essence, bagging is the generalization of the bootstrap to supervised learning algorithms, in which n new datasets through resampling with replacement are generated. Further, each of the Decision Trees is fitted, as described above, to one resampled dataset. Randomness is integrated into the method by providing only a random selection of features of a tree, lowering the variance but slightly increasing the bias. Finally, the prediction of a value is determined by having each tree make a prediction and choosing the one with the most votes:

$$\hat{Y} = \underset{k}{\operatorname{argmax}} \sum_{i=1}^{T} I(\hat{Y}_i = k),$$

where T denotes the number of trees in the forest. Random forests tend to be much more accurate than individual decision trees due to their ensemble nature. However, this comes at the cost of increased computational complexity and memory usage and a loss in interpretability, as random forests, consisting of many deep trees, are much harder to visualize and interpret than a single decision tree.

Boosting: Boosting is an ensemble technique that aims to create a strong classifier from several weak classifiers. In a Random Forest, all weak learners are independent of each other in training. In contrast, in Boosting, the weak learners are trained sequentially, with each new learner attempting to correct the errors of their predecessors. Considering T Decision Trees, the final predictive model is formulated as the weighted sum of the Decision Trees:

$$\hat{Y} = \operatorname{sign}\left(\sum_{i=1}^{T} \alpha_i f_i(x)\right),$$

where α_i denotes the weight of the i-th iteration and $f_i(x)$ the i-th Decision Trees predictions. Since we are in a classification task and the output of the weighted sum is a real number, the sign function is applied to obtain a binary classification. This sequential learning process allows boosting to focus more on the instances that are hard to classify, essentially giving it a corrective iterative nature. Furthermore, by optimally choosing the weights α_i through the training process, the boosting algorithm ensures a harmonized collaboration among weak learners to construct a strong and potentially more accurate predictive model.

Oblique Predictive Clustering Tree [39]: In an Oblique Predictive Clustering Tree (**OPCT**), the splitting of nodes is not restricted to a single feature, but rather a linear combination of features, cutting the feature space along a hyperplane, defined as:

$$w^\intercal x + b \leq 0,$$

where $w \in \mathbb{R}^p$ and $b \in \mathbb{R}$ are the weight vector and bias, respectively. At each node m, the model aims to find the best-splitting hyperplane by minimizing a certain loss function $\mathcal{L}$, which is defined as:

$$\mathcal{L}(Q_m, w, b) = \sum_{z \in Q_m^{left}} l(y, f(x)) + \sum_{z \in Q_m^{right}} l(y, f(x)),$$

where $z = (x, y)$, $l(\cdot)$ some specific classification loss function, i.e. cross-entropy, and $f(x)$ the prediction for datapoint x. Similarly to splitting nodes in basic Decision Trees, Q_m^{left} and Q_m^{right} are determined based on the the hyperplane:

$$Q_m^{left} = \{(x, y) | w^\intercal x + b \leq 0\}$$
$$Q_m^{right} = \{(x, y) | w^\intercal x + b > 0\}.$$

The tree is built recursively, with the optimal parameters w and b being determined at each node to minimize the loss function, effectively determining the best hyperplane to split the data at that node. This process continues, partitioning the data at each node, starting from the root node and progressing until a stopping criterion, such as a minimum node size or a maximum tree depth, is met. This method facilitates the creation of more complex decision boundaries compared to traditional axis-aligned decision trees, potentially leading to more accurate models.

Deep Neural Network [40, 41]: A Deep Neural Network (**DNN**) is a neural network with multiple layers $\vec{h}_k$, with $k = 0, 1, \ldots, L$ between the input ($k = 0$) and output ($k = L$) layers and be described as directed acyclic graph, where neurons are represented as nodes and their connections as edges. Their knowledge is stored in the edges as randomly initialized weight matrices $W_0, W_1, \ldots, W_L$, with $W_k \in \mathbb{R}^{m_{k+1} \times m_k}$, where m_i is the number of neurons in the i-th layer. For prediction, the input is propagated forward through the network, such that the output of the k-th layer is:

$$\vec{h}_k = \sigma\left(W_k \times \vec{h}_{k-1} + \vec{b}_k\right),$$

with $\vec{b}_k$ the bias vector of the k-th layer and $\sigma : \mathbb{R} \to \mathbb{R}$ some activation function, e.g. the sigmoid function. In a classification task, the cross entropy loss function is usually chosen as a function of the parameters of the network and stepwise optimized using gradient descent:

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{n} y_{i,j} \log\left(\hat{y}_{i,j}\right)$$
$$\text{updated via } \theta_{t+1} = \theta_t - \alpha \nabla \mathcal{L}(\theta_t),$$

where θ denotes the parameters of the network, N the number of samples, n the number of classes and $\alpha \in \mathbb{R}$ the step size. The gradient $\nabla \mathcal{L}(\theta_t)$ is calculated via backpropagation, which is the application of the chain rule to the network, yielding:

$$\frac{\partial \mathcal{L}(\theta)}{\partial W_k} = \frac{\partial \mathcal{L}(\theta)}{\partial h_L} \times \frac{\partial h_L}{\partial h_{L-1}} \times \cdots \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_1}{\partial W_k}.$$

DNNs can learn highly complex patterns and hierarchical representations, making them extremely powerful for various tasks. However, they require large amounts of data and computational resources for training, and their highly non-linear nature makes them, in contrast to other methods, somewhat of a "black box," making it difficult to interpret their predictions.

D. Evaluation Metrics

With the methods presented, it is possible to decide whether the code was generated by humans or by an AI if enough qualitative features exist. However, metrics are needed to reflect the actual performance of the models on the respective dataset. In general, there are four fundamental outcomes of a model in classification tasks: (**1**) True Positive (TP), (**2**) False Positive (FP), (**3**) True Negative and (**4**) False Negative (FN). These outcomes are then used to calculate the following metrics [42]:

Accuracy: Accuracy represents the overall rate of correct predictions in proportion to the total number of predictions, with:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN}.$$

It is straightforward but misleading in imbalanced datasets because it does not differentiate between the different types of errors.

Precision: Precision is the ratio of correctly predicted positive instances to the total number of predicted positive instances, with:

$$Prec = \frac{TP}{TP + FP}.$$

It is a useful metric when the costs of False Positives are high, e.g., in medical diagnosis, and is often used in combination with Recall.

Recall: Represented by the equation:

$$Rec = \frac{TP}{TP + FN},$$

it indicates the fraction of all positive instances that the model correctly identified. It is vital when the goal is to capture all positive instances, even if it results in more false positives.

E. Unsupervised Learning method

In addition to SL methods, we also employ an Unsupervised Learning (**UL**) method, namely the Gaussian Mixture Model (**GMM**) [43]. Our motivation for using a GMM stems from the assumption that distinguishable clusters exist between human and AI-generated code snippets, manifesting different stylistic or structural preferences ingrained in their development processes. Initially developed for density estimation, GMMs can dissect the intricate multimodal distributions usually observed in large code datasets, revealing distinctive "peaks" or clusters. By tuning into these subtle nuances, we aspire to map the probability distribution of ChatGPT within a less complex yet robust framework, facilitating the categorization of these snippets into distinct groups based on their genesis. Interestingly, recognizing these components does not initially require human intervention; the number suffices, as features are algorithmically generated rather than extracted. This transition from a density estimation tool to a classification asset is grounded in strong theoretical foundations. It is a promising pathway to a refined classification system capable of unveiling the underlying clustered structures in datasets. Hence, we envisage creating a discerning mechanism for code detection that leverages a simpler model to estimate the underlying patterns, offering a sophisticated approach rooted in statistical theory.

Generally, a GMM is parametrized by a set of K Gaussian distributions, each with a mean $\vec{\mu}_k$ and covariance matrix Σ_k, with $k = 1, \ldots, K$, and a set of mixing coefficients ψ_k, with $\sum_{k=1}^{K} \psi_k = 1$. Thus, the probability is normalized to 1. Each of the K Gaussian distributions is associated with a component C_k, predefined either randomly or with the k-means algorithm, modeling that cluster with the corresponding Gaussian distribution. The probability density function of a GMM is defined as:

$$p(\vec{x}) = \sum_{k=1}^{K} \psi_k \mathcal{N}\left(\vec{x} | \vec{\mu}_k, \Sigma_k\right),$$

where $\mathcal{N}(x|\vec{\mu}_k, \Sigma_k)$ denotes the probability density function of a Gaussian distribution with mean $\vec{\mu}_k$ and covariance matrix Σ_k. The means $\vec{\mu}_k$ of the clusters are initialized to the centroids obtained from the k-means algorithm. The covariance matrices σ_k are initialized using the sample covariances of the data points assigned to the respective clusters, and the mixing coefficients ψ_k are determined based on the proportions of data points in each cluster. The clusters are a starting point for optimizing the GMM with the Expectation-Maximization (**EM**) [44] algorithm in this setup. The EM algorithm is an iterative method that alternates between the expectation and maximization steps. In the expectation step, the posterior probabilities of the data points are calculated, i.e., the probability that a data point belongs to the k-th component, respectively Gaussian distribution, with:

$$\hat{\gamma}_{ik} = \frac{\hat{\psi}_k \mathcal{N}(\vec{x}_i | \hat{\vec{\mu}}_k, \hat{\Sigma}_k)}{\sum_{j=1}^{K} \hat{\psi}_j \mathcal{N}(\vec{x}_i | \hat{\vec{\mu}}_j, \hat{\Sigma}_j)},$$

where $\hat{\psi}_k$, $\hat{\vec{\mu}}_k$ and $\hat{\Sigma}_k$ denote the current estimates of the mixing coefficients, means and covariance matrices, respectively, and, thus, $\hat{\gamma}_{ik} = p(C_k | \vec{x}_i, \hat{\psi}, \hat{\vec{\mu}}, \hat{\Sigma})$. In the maximization step, the parameters of the GMM are updated to maximize the likelihood of the data given the current estimates of the parameters, with:

$$\hat{\psi}_k = \frac{1}{N} \sum_{i=1}^{N} \hat{\gamma}_{ik}, \quad \hat{\vec{\mu}}_k = \frac{1}{N\hat{\psi}_k} \sum_{i=1}^{N} \hat{\gamma}_{ik} \vec{x}_i$$

$$\hat{\Sigma}_k = \frac{1}{(N-1)\hat{\psi}_k} \sum_{i=1}^{N} \hat{\gamma}_{ik} (\vec{x}_i - \hat{\vec{\mu}}_k)(\vec{x}_i - \hat{\vec{\mu}}_k)^{\intercal}.$$

Iteratively repeating this process guarantees to always at least converge to a local optimum, i.e., a slight change in the log-likelihood, and potentially to a global optimum. Regarding the number of components, determining the optimal number of components (or clusters) K is a critical step in the modeling process. We can employ techniques such as the Bayesian Information Criterion (**BIC**) [45] or the Akaike Information Criterion (**AIC**) [46] to find the most suitable number of components, helping to avoid overfitting while capturing the essential structures in the data. It balances model complexity and goodness of fit, aiming for a model that explains the data with the fewest parameters necessary.

IV. Experimental Setup

In this section, we outline the requirements to carry out our experiments. We cover basic hard- and software components, as well as the collection and pre-processing of data to apply the methodology and models presented in the previous section.

A. Soft- and Hardware

Our experiments were implemented in Python, ranging from version 3.9.16 to 3.10. We used various packages and libraries, which can be found in this GitHub repository:https://github.com/MarcOedingen/GP_ChatGPT_DetectionModel. All experiments were performed on MacOS, specifically with Apple Silicon, potentially resulting in a non-functional version on other operation systems and processor architectures. Furthermore, due to the large amount of data, a minimum of 20 GB of storage and 24 GB of RAM is recommended for working with the data correctly.

B. Data Collection

One of the most prominent challenges was collecting and gathering a suitable dataset to apply the presented algorithms. An ideal dataset for our use case consists of code snippets that meet the following criteria: (**1**) Problem description, (**2**) canonical solution(s), (**3**) test cases and (**4**) an entry point of the solutions.

Firstly, the problem description provides a fundamental guide, shaping the trajectory for the solutions generated by ChatGPT. This component ensures that close attention is paid to essential parts of the problem, fostering a deep understanding reflected in the solutions generated. Conversely, an unclear defined problem description may lead ChatGPT to create solutions that hide the core problem. As a result, this could lead to a deterioration in the quality of the solutions, reflecting a misalignment with the central issue of the problem and, thus, eliminating a potentially helpful solution from the samples, which are limited already. While mentioning canonical solutions, it is noteworthy that they are not incorporated within ChatGPT's input prompt. However, they hold a significant role in the posterior analysis, serving as a referential benchmark against the output of ChatGPT, without which our research would not be feasible. Further, test cases are needed to compare the canonical solutions with semantically correct code that solves precisely the task set by the problem description. Consequently, we focus strongly on syntactic and executable code but ignore a possibly typical behavior of ChatGPT in case of uncertainties or wrong answers. However, comparing arbitrary code with the canonical solutions would cause a strong bias in the data, which could heavily contaminate the pattern recognition. Furthermore, a stringent set of test cases facilitates the elimination of solutions that, while syntactically correct, fail to meet the functional requirements stipulated in the problem description, thereby ensuring that the generated solutions are technically accurate and practically feasible. Moreover, it fosters a controlled environment where the functionality of the code is rigorously tested, thus averting the inclusion of code snippets that operate based on incorrect logic or those that might potentially produce erroneous outputs, thereby maintaining the integrity of the dataset and ensuring the reliability of the subsequent analyses. Finally, delineating a clear entry point for the solutions augments the evaluation process. This element is quintessential in discerning the precise juncture at which a function invocation should occur, especially for solutions necessitating such calls to initiate the computational process.

Suitable data meeting all the above criteria are given by programming tasks from various programming competitions. Given the abundance and quality of tasks available across multiple programming languages, Python was selected as the preferred language for this initiative. It should be noted that the data originates from a period preceding the launch of ChatGPT, thereby nullifying any concerns of bias associated with including ChatGPT-influenced material. Yet, the sparse availability of pure AI-generated code, especially ChatGPT, is an obstacle. To address this, it was imperative to create at least one potential solution generated by ChatGPT for every problem at hand. For generating the AI code examples, we used OpenAI's `gpt3.5-turbo` API [25] with the standard parameters.

In reference to study [14], a claim was made suggesting a probability of $\approx 48.1\%$ for obtaining a correct solution from the HumanEval dataset of `gpt-3.5-turbo` upon a single generation attempt. Our experimental verification yielded a notably lower average probability of $\approx 21\%$ after a single generation. In this endeavor, the role of prompt engineering cannot be overstated. Prompt engineering is a separate research area that seeks to leverage the intrinsic capabilities of an LLM while mitigating potential pitfalls associated with unclear problem descriptions or inherent biases. During the project, we tried different prompts to increase the yield of successful generations. Our most successful one is given by: "*Question:* <Coding_Task_Description> *Please provide your answer as Python code. Answer:*". Other prompts, i.e., "*Question:* <Coding_Task_Description> *You are a developer writing Python code. Put all python code $PYTHON in between [[[$PYTHON]]]. Answer:*", led to a detailed explanation of the problem and an associated solution strategy of the model. However, only a part of the prompt as a solution in code was returned: *[[[$PYTHON]]]*.

We conducted five distinct API calls for each problem sourced from our dataset to enhance the accuracy rate further. This strategy improved considerably, with the accuracy rate surging to $\approx 45\%$. This outcome aligns with the assertion by [14, 47], substantiating the theory that an increment in generation attempts correlates positively with heightened accuracy levels. Furthermore, it is noteworthy that the existence of multiple AI-generated solutions for a single problem does not pose an issue, given that the majority of problems possess various canonical solutions, see table I for a detailed view of the sources used and the number of canonical solutions.

Dataset	n_{PROBLEMS}	$\bar{n}_{\text{SOLUTIONS}}$	n_{SAMPLES}
APPS [48]	1.00×10^4	21	2.10×10^5
CCF [49]	1.56×10^3	18	2.81×10^4
CC [50]	8.26×10^3	15	1.23×10^5
HAEA [51]	1.49×10^3	16	2.38×10^4
HED [47]	1.64×10^2	1	1.64×10^2
MBPPD [52]	9.74×10^2	1	9.74×10^2
MTrajK [53]	1.44×10^2	1	1.44×10^2
Sum	$\mathbf{2.26 \times 10^4}$	-	$\mathbf{3.86 \times 10^5}$

TABLE I
DATASETS AND CALCULATION OF TOTAL HUMAN SAMPLES n_{PROBLEMS} - NUMBER OF DISTINCT PROBLEMS, $\bar{n}_{\text{SOLUTIONS}}$ - AVERAGE NUMBER OF CANONICAL SOLUTIONS PER PROBLEM AND n_{SAMPLES} - TOTAL SAMPLES PER DATA SOURCE.

C. Data Preprocessing

When dealing with Big Data, which pertains to our dataset with a total sample size of 3.86×10^5, see table I, it is inevitable and of great importance to clean the data for the later application of algorithms. In general, there are four different types of data quality problems: (**1**) Missing values, (**2**) duplicate data, (**3**) outliers, and (**4**) noise [54].

Missing values: In data preprocessing, adequately addressing missing values is of paramount importance to prevent the induction of bias and to facilitate precise analytical conclusions. These absent values not only harbor the potential to skew analytical outcomes but can also hinder the proper functioning of algorithms, resulting in an incomplete and inconclusive data analysis process. Thus, a meticulous approach to managing missing values is fundamental to laying a robust groundwork for effectively distinguishing between AI and human-generated code. During the creation of AI samples, a prevalent issue was missing data, with empty responses from the API often stemming from faulty communication or excessive server load. Similarly, ChatGPT sometimes generated responses that did not meet the criteria of desired output, notably the absence of Python code. In addressing this, we adopted a stringent approach by eliminating the samples characterized as missing data, a strategy grounded in the impossibility of substituting these samples with meaningful and accurate placeholders. Conversely, the datasets comprising human samples required no such scrutiny, as they were pre-prepared and devoid of missing values, thereby eliminating concerns over incomplete or incorrect data entries in this subset.

Duplicate data: Eliminating duplicate entries is essential in safeguarding the efficacy of subsequent algorithm applications; it prevents the overpopulation of particular solutions, thereby fostering a balanced and representative dataset that can facilitate more nuanced and objective analyses. Due to the versatile choice of different sources, there is a high risk of duplicated data for individual problems and their human- and AI-generated solutions. While humans may have inherently different solution strategies, the diversity of ChatGPT is controlled by the model parameter during inference. The most prominent parameter in that context is the temperature $T \in \mathbb{R}$, with $0 \leq T \leq 2$, which controls the randomness in the output. Values closer to zero indicate a fully deterministic behavior, whereas values closer to two represent some non-deterministic behavior, with a very low probability of having an output twice in thousands of generations. Thus, the probability of the same output for five different API calls at the specified size of $T = 0.75$ is close to zero. However, duplicates are removed from other problems and human- and AI-generated samples.

Outliers: Outliers can significantly impact the training process of models by introducing errors that distort underlying patterns and relationships within the data. In this context, biased decision-making and the resulting explanations for a generated output are particularly critical. Let $f : \mathcal{S} \to \mathbb{R}$ be a function that signifies the number of characters in a code snippet. Our objective is to systematically exclude those sample pairings (X_{AI}, X_{HU}) wherein there is a substantial discrepancy in the lengths of the code. A standard method for this is the empirical rule [55]. For normally distributed data, which we have in our case, this rule states that:

$$\Pr\left(\mu - 3\sigma \leq |f(X_{AI}) - f(X_{HU})| \leq \mu + 3\sigma\right) \approx 0.9973,$$
$$\text{where } \sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}}.$$

Thus, $\approx 99.73\%$ of the data lie in this range, forming a criterion for filtering outliers based on the difference in code lengths. It implies that data points falling outside this range can be considered abnormal and removed from the analysis. This technique ensures a more balanced dataset, fostering accurate and unbiased model training.

Noise: Noise elimination is often reserved for time-dependent processes, typically characterized by recurrent behaviors that noticeably deviate from the average behavior or trend. In many instances, this is undertaken to hone in on central tendencies and discard anomalies that could cloud the understanding of the underlying structure or principle guiding a phenomenon. However, in the context of our investigation, where the intrinsic goal is to identify and analyze precisely such divergent structures, eliminating noise would make the search exponentially more

Algorithm 1 Data Preparation and Modelling

DATA COLLECTION

```
Require: Data sources $\mathcal{D}$
 1: procedure CollectProblems($\mathcal{D}$)
 2:     $\mathcal{P} \leftarrow \emptyset$                                   ▷ Set of Coding Problems $\mathcal{P}$
 3:     for each $\mathcal{D}_i \in \mathcal{D}$ do
 4:         for each $\mathcal{P}_j \in \mathcal{D}_i$ do
 5:             $P_{\text{DCR}}, P_{\text{HSol}}, P_{\text{Test}}, P_{\text{Entr}} \leftarrow \texttt{extract}(\mathcal{P}_j)$
 6:             $\mathcal{P} \leftarrow \mathcal{P} \cup (P_{\text{DCR}}, P_{\text{HSol}}, P_{\text{Test}}, P_{\text{Entr}})$
 7:         end for
 8:     end for
 9: end procedure
```

```
Require: API $\mathcal{A}$, Coding Problems $\mathcal{P}$ and $n \in \mathbb{N}$ calls
 1: procedure GenerateSolutions($\mathcal{A}, \mathcal{P}, n$)
 2:     $\mathcal{P}_{\text{GPT}} \leftarrow \emptyset$           ▷ GPT solutions per Coding Problem
 3:     for each $\mathcal{P}_j \in \mathcal{P}$ do
 4:         $\text{PROMPT} \leftarrow \texttt{createPrompt}(\mathcal{P}_j)$
 5:         $\mathcal{A}_{\text{Out}} \leftarrow \mathcal{A}(\text{PROMPT}, n)$
 6:         $\mathcal{P}_{\text{GPT}} \leftarrow \mathcal{P}_{\text{GPT}} \cup \mathcal{A}_{\text{Out}}$
 7:     end for
 8:     $\mathcal{P} \leftarrow \mathcal{P} \oplus \mathcal{P}_{\text{GPT}}$                     ▷ Concat $\mathcal{P}_{\text{GPT}}$ to $\mathcal{P}$
 9: end procedure
```

DATA PRE-PROCESSING

```
Require: Coding Problems $\mathcal{P}$, Code Length characters function $f$
 1: procedure CleanData($\mathcal{P}, f$)
 2:     for each $\mathcal{P}_j \in \mathcal{P}$ do
 3:         $\mathcal{P}_{\text{Null}} \leftarrow \texttt{isNull}(\mathcal{P}_j)$
 4:         $\mathcal{P}_{\text{NP}} \leftarrow \texttt{noPython}(\mathcal{P}_{j,GPT})$
 5:         $\mathcal{P}_{\text{Dupl}}, \mathcal{P}_{\text{Out}}, \mathcal{P}_{\text{Tests}} \leftarrow \emptyset, \emptyset, \emptyset$
 6:         $\mu, \sigma \leftarrow \frac{1}{n}\sum_{i=0}^{n} \mathcal{P}_{i,\text{Code}}, \sqrt{\frac{\sum_{i=1}^{n}(\mathcal{P}_{i,\text{Code}}-\mu)^2}{n}}$
 7:         for each $\mathcal{P}_k \in \mathcal{P}$ do
 8:             if $j \neq k$ then
 9:                 $\mathcal{P}_{\text{Dupl}} \leftarrow \mathcal{P}_{\text{Dupl}} \cup \texttt{equals}(\mathcal{P}_j, \mathcal{P}_k)$
10:             end if
11:         end for
12:         $d \leftarrow |f(\mathcal{P}_{j,\text{HSol}}) - f(\mathcal{P}_{j,\text{GPT}})|$
13:         $\mathcal{P}_{\text{Out}} \leftarrow (\mu - 3\sigma > d$ or $d > \mu + 3\sigma)$
14:         $\mathcal{P}_{\text{Tests}} \leftarrow \mathcal{P}_{j,\text{Test}}\left(\mathcal{P}_{j,\text{HSol}}, \mathcal{P}_{j,\text{GPT}}, \text{FALSE}\right)$
15:         $\mathcal{P} \leftarrow \mathcal{P} \setminus \{\mathcal{P}_{\text{Null}}, \mathcal{P}_{\text{NP}}, \mathcal{P}_{\text{Dupl}}, \mathcal{P}_{\text{Out}}, \mathcal{P}_{\text{Tests}}\}$
16:         $\mathcal{T}_{\text{Human}}, \mathcal{T}_{\text{AI}} \leftarrow \emptyset, \emptyset$
17:         $l_i, k_i \leftarrow |\mathcal{P}_{j,\text{HSol}}|, |\mathcal{P}_{j,\text{GPT}}|$
18:         for $q = 1, 2, \ldots, \min(l_j, k_j)$ do
19:             $\mathcal{T}_{\text{Human}} \leftarrow \mathcal{T}_{\text{Human}} \cup (\mathcal{P}_j, l_{i,q})$
20:             $\mathcal{T}_{\text{AI}} \leftarrow \mathcal{T}_{\text{Human}} \cup (\mathcal{P}_j, k_{i,q})$
21:         end for
22:         $\mathcal{P}_{j,\text{HSol}}, \mathcal{P}_{j,\text{GPT}} \leftarrow \mathcal{T}_{\text{Human}}, \mathcal{T}_{\text{AI}}$
23:     end for
24: end procedure
```

MODELLING

```
Require: Coding Problems $\mathcal{P}$, Tokenization function $\mathcal{T}$, Embedding function $\mathcal{E}$, Feature function $\mathcal{F}$
 1: procedure AlgorithmApplication($\mathcal{P}, \mathcal{T}, \mathcal{E}, \mathcal{F}$)
 2:     $X \leftarrow \{\mathcal{P}_{\text{HSol}} \cup \mathcal{P}_{\text{GPT}}\}$
 3:     $X_{\text{Emb}}, X_{\text{TEmb}}, X_{\text{Feat}} \leftarrow \emptyset, \emptyset, \emptyset$
 4:     for each $x \in X$ do
 5:         $X_{\text{Emb}} \leftarrow X_{\text{Emb}} \cup \mathcal{E}(x)$
 6:         $X_{\text{TEmb}} \leftarrow X_{\text{TEmb}} \cup \mathcal{E}\left(\mathcal{T}(x)\right)$
 7:         $X_{\text{Feat}} \leftarrow X_{\text{Feat}} \cup \mathcal{F}(x)$
 8:     end for
 9:     $X_{\text{GPT}} \leftarrow \texttt{concat}(\mathcal{P}_{\text{GPT}})$           ▷ Concatenated matrix
10:     $X_{\text{Per}} \leftarrow X_{\text{Emb}}^{\text{GPT}} + \mathcal{N}\left(\bar{X}_{\text{Emb}}^{\text{GPT}}, \alpha \cdot \text{Cov}\left(X_{\text{Emb}}^{\text{GPT}}\right)\right)$
11:     $Y \leftarrow \texttt{label}(\{X_{\text{Emb}}, X_{\text{Per}}, X_{\text{TEmb}}, X_{\text{Feat}}\}, \{0, 1\})$
12:     for each Classifier $\mathcal{C}_i \in \mathcal{C}$ do
13:         $\mathcal{C}_i \leftarrow \texttt{fit}\left(\mathcal{C}_i, \left(X_{\text{Feat}}^{\text{Train}}, \mathcal{Y}_{\text{Train}}\right)\right)$
14:         $Y_{\mathcal{C}}^{i} \leftarrow \texttt{predict}\left(X_{\text{Feat}}^{\text{Test}}, \mathcal{Y}_{\text{Test}}\right)$
15:     end for
16:     $\mathcal{D}_{\text{Emb}} \leftarrow \texttt{fit}\left(\mathcal{D}_{\text{Emb}}, X_{\text{Emb}}^{\text{Train}}, \mathcal{Y}_{\text{Train}}\right)$          ▷ DNN $\mathcal{D}$
17:     $Y_{\mathcal{D}}^{\text{Emb}} \leftarrow \texttt{predict}\left(X_{\text{Emb}}^{\text{Test}}, Y_{\text{Test}}\right)$
18:     $\mathcal{D}_{\text{Per}} \leftarrow \texttt{fit}\left(\mathcal{D}_{\text{Per}}, X_{\text{Per}}^{\text{Train}}, \mathcal{Y}_{\text{Train}}\right)$
19:     $Y_{\mathcal{D}}^{\text{Per}} \leftarrow \texttt{predict}\left(X_{\text{Per}}^{\text{Test}}, Y_{\text{Test}}\right)$
20:     $\mathcal{G}_{\text{AI}} \leftarrow \texttt{fit}\left(\mathcal{G}_{\text{AI}}, X_{\text{Emb}}^{\text{GPT}} \veebar X_{\text{TEmb}}^{\text{GPT}}\right)$                 ▷ GMM $\mathcal{G}$
21:     $\mathcal{G}_{\text{HU}} \leftarrow \texttt{fit}\left(\mathcal{G}_{\text{HU}}, X_{\text{Emb}}^{\text{HU}} \veebar X_{\text{TEmb}}^{\text{HU}}\right)$
22:     $Y_{\mathcal{G}} \leftarrow \mathcal{G}_{\text{HU}}\left(X_{\text{Emb}}^{\text{Test}} \veebar X_{\text{TEmb}}^{\text{Test}}\right) \overset{?}{>} \mathcal{G}_{\text{AI}}\left(X_{\text{Emb}}^{\text{Test}} \veebar X_{\text{TEmb}}^{\text{Test}}\right)$
23:     $\text{RESULTS} \leftarrow \texttt{eval}\left(Y_{\text{Test}}, Y_{\mathcal{C}}^{i}, Y_{\mathcal{D}}^{\text{Emb}}, Y_{\mathcal{D}}^{\text{Per}}, Y_{\mathcal{G}}\right)$
24: end procedure
```

challenging and render it nearly impossible. Moreover, the intricacies brought about by the noise could offer deeper insights, revealing hidden patterns and nuances that would otherwise remain undiscovered. Given this, we have consciously opted against eliminating noise in our analytical approach. This decision is grounded in the desire to preserve the rich fabric of information embedded in the data, allowing for a more exhaustive and potentially rewarding exploration of the dataset. The approach ensures we do not lose sight of valuable details, fostering a comprehensive understanding rather than a generalized view. Consequently, we anticipate that this methodology will pave the way for discoveries that are both nuanced and grounded in a fuller representation of the data landscape.

In addition to the established data preprocessing techniques previously delineated, we further refined the sample pool by the employment of two strategies: (**1**) Utilizing the task-affiliated assessments to check each sample for correctness, and (**2**) paired human- and AI-generated samples, guaranteeing an equivalent number of solutions from both human and AI sources for identical tasks, and, thus, a balanced dataset. In the general prerequisites, we have already stated the importance of having reliable and correct samples since we want to compare syntactically and semantically meaningful code snippets. Therefore, human- and AI-generated samples underwent

Dataset	n_{HUMAN}	n_{AI}	n_{TOTAL}
APPS	1.93×10^3	1.93×10^3	3.86×10^3
CCF	1.57×10^3	1.57×10^3	3.14×10^3
CC	1.02×10^4	1.02×10^4	2.04×10^4
HAEA	1.27×10^2	1.27×10^2	2.54×10^2
HED	2.12×10^3	2.12×10^3	4.24×10^3
MBPPD	6.07×10^2	6.07×10^2	1.21×10^3
MTrajK	5.90×10^1	5.90×10^1	1.18×10^2
Sum	$\mathbf{1.66 \times 10^4}$	$\mathbf{1.66 \times 10^4}$	$\mathbf{3.32 \times 10^4}$

TABLE II
ALIGNED EXPERIMENT DATA SET WITH THEIR CORRESPONDING SOURCES n_{HUMAN} - NUMBER OF HUMAN SAMPLES, n_{AI} - NUMBER OF AI SAMPLES AND n_{TOTAL} - NUMBER OF TOTAL SAMPLES

a systematic process through a validation pipeline to assert their intended functionality. Some data points did not exhibit any test cases; hence, all dependent samples were removed. By running the test pipeline, many samples were discarded, both ChatGPT-generated and canonical solutions that were no longer executable due to different Python versions. However, there still is an imbalance between the number of possible human solutions and the number of generated AI samples. Suppose we have n coding problems $\mathcal{P}$ with l_i human- and k_i AI-solutions for each coding problem $P_i \in \mathcal{P}$, then we create pairs $\mathcal{T}$ with:

$$\mathcal{T}_{\text{HUMAN}} = \{(P_1, l_{i,1}), (P_2, l_{i,2}), \ldots, (P_p, l_{i,p})\} \text{ and}$$
$$\mathcal{T}_{\text{AI}} = \{(P_1, k_{i,1}), (P_2, k_{i,2}), \ldots, (P_p, k_{i,p})\}, \text{ where}$$
$$p = \min(l_i, k_i).$$

Thus, p samples for each coding problem C_i for human and AI samples, respectively. Leveraging the refined preprocessing strategies delineated, we have successfully created a balanced human and AI contributions dataset. As detailed in table II, each category now hosts an equal number of human and AI samples, with the total number of cleaned and tested samples amounting to 3.32×10^4. The dataset is a testament to the rigorous preprocessing steps, setting a firm ground for the explorative journey in distinguishing between AI- and human-generated code.

V. RESULTS

In this section, we discuss the primary outcomes from deploying the introduced models in section III, operating on the preprocessed dataset presented in section IV. The resulting performance measured by the introduced metrics in section III of each model is displayed in table IV. Therefore, we aim to offer a balanced view that could serve as a cornerstone for future explorative studies in this domain.

A. *Indications*

One of the most critical indicators of a possible distinction between AI and human-generated code is the visualization of the embeddings. Given the ADA-Embeddings of a random subset from our final dataset, we can use the t-distributed stochastic neighbor embeddings (**t-SNE**) [56] to map the high dimensionality of ADA to lower dimensions with advantageous representation, making them interpretable by humans, see figure 2. In the two-dimensional space, there is hardly any difference between the samples generated by ChatGPT and those generated by humans, as the points in space are very dense. Finding a separating non-linear function is almost impossible, as points partially lie on each other, in contrast to the three-dimensional representation, where points in space become sparser, and the distance between individual samples increases. Therefore, it is hypothesized that within the high-dimensional space spanned by the embedding, a hyperplane can distinctly segregate the samples.

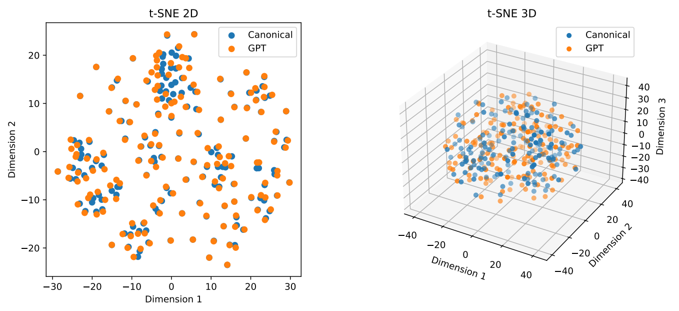


Fig. 2. 2D AND 3D t-SNE REPRESENTATION OF ADA-EMBEDDING OF RANDOM PAIRED SAMPLES FROM THE FINAL DATASET

Pursuing this hypothesis, we conducted a Principal Component Analysis (**PCA**) [57], discovering that a substantial portion of the dataset's variance could be retained by reducing the dimensionality to 570 principal components, encompassing about 98% of explained variance. Retaining a high percentage of explained variance is pivotal in preserving the underlying structure and information of the data, thereby ensuring that the reduced-dimension data retains most of the original data's variability. However, despite its large variance coverage, this representation did not encourage a significantly improved human interpretability compared to the full 1536-diemnsional space. Conversely, lower dimensions, although enabling clearer visualizations, encapsulate a minimal of the total explained variance, with two components $\approx 10.81\%$ and three $\approx 14.07\%$. Thus supporting the t-SNE visualization outcomes and illustrating a substantial loss of information while emphasizing the necessity to aim for higher dimensions to preserve the intricate differentiation between AI and human samples. This essentially frames the trade-off between interpretability and information preservation, where reducing dimensions too drastically can lead to a loss of critical information, hindering accurate analyses.

To further bolster this investigation, we extended our analytics to include cosine similarity analyses of the unformatted embeddings generated by human and AI sources in the 1536-dimensional space. Cosine similarity is particularly effective in high-dimensional spaces, helping to identify the cosine of the angle between two vectors, which can provide insights into their similarity; in this context, it aids in discerning potential distinctions or convergences between different code embeddings based on their directional closeness in the multi-dimensional space. Leveraging the ADA text embedder, we presumed that distinct clusters could emerge in the high-dimensional space, potentially delineated based on the unique coding styles and syntax intrinsic to different programming languages. Our analysis revealed a notable average cosine similarity of $\approx 75.50\%$ between all samples, indicating a common direction of python-code-embedded vectors in that embedding space. Paying attention to the diagonal of the heatmap, which represents the cosine similarity of paired AI and human responses, we observed a significantly higher value of $\approx 87.61\%$. This high degree of similarity potentially refers to the underlying syntactic and stylistic consistencies inherent in a specific coding task, substantiating the notion that both human and AI-generated codes for a given task share a substantial degree of semantic space, possibly hinting at a foundational ground truth in coding solutions that both humans and AIs adhere to. This observation amplifies the existence of a high-dimensional hyperplane where a rich representation of both AI and human responses could potentially be segregated effectively, thereby offering a fertile ground for deeper exploration and analysis in distinguishing the nuanced differences between AI and human coding patterns.

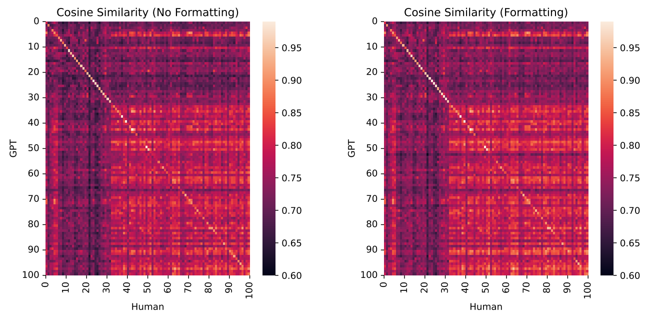


Fig. 3. COSINE SIMILARITY OF ADA-EMBEDDING OF RANDOM FORMATTED AND UNFORMATTED PAIRED SAMPLES FROM THE FINAL DATASET

The abovementioned utilization of high-dimensional embedding models for unsupervised feature extraction lacks interpretability. Consequently, we pursued the creation of features grounded in sample syntax. Following experimentation with various feature constructs, a set of nine features delineated in Table III was curated for subsequent model training. The data is split by their class, and independently, the features are extracted from the corresponding samples. All feature values are normalized to make them comparable.

Examination of subfigure V-B reveals discernible distinctions in mean values across diverse features between GPT-generated and human-authored samples. This observation holds promise, suggesting the feasibility of training a classification model using this dataset. Notably, the feature denoted as NUMBER OF LEADING WHITESPACES exhibits a substantial mean difference of 0.34. This differential provides a robust basis for classifier models to discriminate between these classes.

To further substantiate this supposition, we employed a decision tree classifier model with default parameter settings to train on the dataset. The resultant feature importance scores, as illustrated in Figure 4, yielded intriguing insights. Surprisingly, the feature NUMBER OF LEADING WHITESPACES garnered a meager feature importance score of 0.002, whereas the feature NUMBER OF PUNCTUATIONS achieved a considerably higher score of 0.234. It is imperative to emphasize that these feature importance scores are specific to the Decision Tree Classifier and should not be extrapolated to other model types.

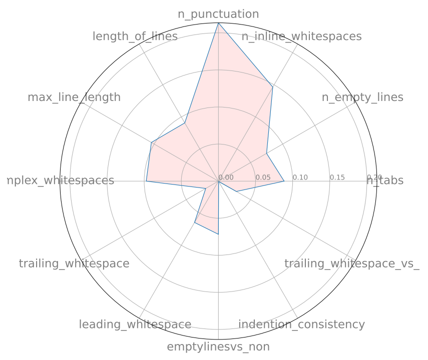


Fig. 4. FEATURE IMPORTANCES OF DECISION TREE CLASSIFIER (**DTC**)

Because the extracted features are exclusively based on the structural layout of code samples, they encompass only one aspect, namely, method 1, of the fraudulent solution creation methods discussed in Section III-A. This opens the possibility that the structural style of GPT-generated code samples could be altered to resemble a ”human” formatting style, rendering the features obsolete. In order to enhance the robustness of these features, human-authored and GPT-generated samples were subjected to formatting using the Black code formatter [58], a Python code formatting tool. This processing step standardized all samples to a uniform formatting style.

Looking at the heatmap for formatted code snippet embeddings, depicted in figure 3 on the right, we can see that the average cosine similarity between all formatted samples increased to 81%, and even to 93% on the main diagonal. This shows that there is not much deviation from GPT-generated to human-written samples anymore. As anticipated, this standardization process reduced the disparity of feature means between the two classes. Classifiers trained on these standardized samples are expected to exhibit lower performance compared to classifiers trained on the original, unformatted samples.

B. Supervised Methods

Most supervised methodologies predominantly rely on training using the aforementioned interpretable features. For this experiment, traditional classifier models, which have garnered validation in industrial applications, have been employed to shift the analytical focus away from the classification model and toward the input variables, specifically the interpretable features. All models have been initialized with default configurations as prescribed by the scikit-learn Python library [59]. The selected models encompass a Random Forest Classifier (**RFC**), a Logistic Regression (**LR**), a Decision Tree Classifier (**DTC**), an Oblique Prediction Clustering Tree (**OPCT**), a Gradient Boosting Classifier (**XGBC**), and an Adaptive Boosting Classifier (**ADAC**).

Among these models, the Random Forest Classifier (RFC) achieved the highest accuracy, reaching an impressive 88%. Furthermore, the RFC model also delivered the highest precision score, at 86%. On the other hand, the Logistic Regression (LR) model consistently exhibited the best recall score, averaging a remarkable 95% across ten runs. These outcomes demonstrate the models' strong performance, suggesting that the extracted features effectively capture the distinctions between human-written and AI-generated code. This observation directly addresses Research Questions 1 and 2.

More interestingly is the performance of such models on the formatted dataset. On formatted samples, the RFC model scores an accuracy of 81%, a precision score of 81%, and a recall score of 83%, beating the other models. As assumed in section V-A, the models perform worse than in the unformatted format. For example, the RFC model has a gap of 7% accuracy, 5% precision score, and 8%, compared to the results on unformatted data. While these results are still strong, we can see that by applying simple modification, the performance of our models can be diminished. This raises the question of what other ways of altering the code samples can be applied to blur the line between the two classes.

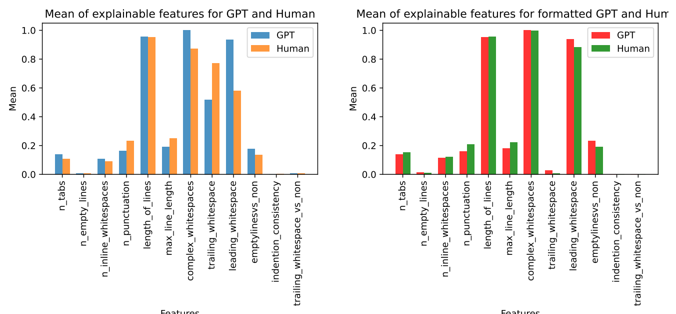


Fig. 5. EXPLAINABLE FEATURE MEANS FOR FORMATTED AND UNFORMATTED HUMAN AND GPT SAMPLES

The performance of these models is particularly interesting when applied to the formatted dataset. In this context, the Random Forest Classifier model stands out with an accuracy of 81%, a precision score of 81%, and a recall score of 83%, surpassing the other models. As hypothesized in Section V-A, the models exhibit diminished performance on the formatted samples. For instance, the RFC model experiences a reduction of 7% in accuracy, 5% in precision score, and 8% in recall compared to the results obtained on unformatted data. While these outcomes still showcase robust performance, they underscore the notion that by implementing straightforward modifications, the efficacy of our models can be compromised. This prompts further exploration into alternative methods of altering code samples to blur the distinction between the two classes.

Due to the inherent opaque nature of embeddings and DNNs, they do not provide transparency in their decision-making processes. However, they achieve superior performance compared to the White-box models. Our evaluations indicate that the DNN-based models almost perfectly classify human- and AI-generated code by exceeding accuracies of 98%, see table IV. From an algorithmic point of view, our procedure is not particularly complex: Let $\mathcal{E} : \mathcal{C} \to \mathbb{R}^m$ be the function that embeds single code snippets $x \in \mathcal{C}$, then we embed each code snippet in our dataset, label them respectively with either 0 for human or 1 for AI, and train the DNN. However, an additional extension is given by adding noise. We want to mimic the change of code by a human by replacing 20% of the training dataset with perturbed embeddings collected from $\mathcal{N}\left(\bar{X}, \alpha \cdot \mathrm{Cov}(X)\right)$, as introduced in section III.

The disparities in performance among the various modeling approaches underscore the pivotal role of embedding techniques in discerning between human- and AI-generated code. While grounded in domain-specific intuition, traditional feature engineering finds itself somewhat constrained, unable to capture the complexities inherent in code snippets as comprehensively as embedding models like Ada and TF-IDF. Specifically, the ability of the Ada embedding model to harness abstract patterns and semantic relations, as well as to discern the underlying logic and intent of code snippets, serves to position it as an invaluable asset in this classification task. Its training across diverse domains imbues it with a richness of representation, enabling it to comprehend and interpret the code snippets in a multifaceted manner beyond the capabilities of handcrafted features. Moreover, the TF-IDF embedding, despite its limitations in only accessing tokens within its training dataset, showcases a noteworthy performance by focusing on a more granular view of tokens. Surpassing 95% across all metrics evaluated demonstrates its effectiveness, even more so when considering that handcrafted features were unable to achieve similar results. In light of this, it becomes evident that the depth and breadth of information captured by embedding models are indispensable. Although valuable, traditional, intuition-based feature engineering approaches exhibit limitations in capturing the intricate nuances and relationships within the code snippets, as reflected in their relatively lower performance. Given these rich embeddings, DNNs come into play and unfold as an ideal deciphering

Feature	Description
NUMBER OF TABS	The code sample is split by every new line, and the number of indentations is summed up.
NUMBER OF EMPTY LINES	The code sample is split by every new line, and all empty lines are counted.
NUMBER OF INLINE WHITESPACES	All whitespaces not at the beginning or end of a line are summed up.
NUMBER OF PUNCTUATIONS	The code sample is filtered with a regular expression [^\w\s] and the left characters are counted
AVERAGE LENGTH OF LINES	The average length of characters per line in a sample.
MAX LENGTH OF LINES	The code sample is split by every new line, and the maximum count of characters of a line is returned.
NUMBER OF TRAILING WHITESPACES	The code sample is split by every new line, and all trailing whitespaces are summed up.
NUMBER OF LEADING WHITESPACES	The code sample is split by every new line, and all leading whitespaces are summed up.
COMPLEX WHITE SPACES	The ratio of used tabs to white spaces. With N_s being the number of spaces and N_t being the number of tabs, it is calculated with $N_{st} = \begin{cases} \frac{N_s}{N_t+N_s} & \text{for } N_t + N_s > 0 \\ 0 & \text{for } N_t + N_s \leq 0 \end{cases}$

TABLE III
DETAILED DESCRIPTION OF ALL FEATURES USED FOR SUPERVISED CLASSIFIERS EXCEPT FOR DNN

tool. Their multi-layered structure allows them to discover and learn complex non-linear patterns in these high-dimensional embeddings, absent from human intuition. We further reinforce the hypothesis that embeddings capture a more nuanced and comprehensive representation of code snippets, as evidenced by the comparative performance of DNNs trained on these embeddings ($\geq 92\%$) versus handcrafted features ($\geq 83\%$) for all evaluated metrics. The diminished efficacy of DNNs when utilizing handcrafted features highlights the inherent limitations of such features in encapsulating the multifaceted nature and complexities of code. Consequently, the compelling distinction in performance serves an indispensable role in embedding techniques to enhance the capability of DNNs to discern intricate patterns and relationships, thereby significantly advancing the field of code classification.

In this experiment, intentionally incorporating noise to mimic the variability and unpredictability inherent in human coding behavior has enhanced the model's robustness and generalization capabilities, showing peak performance among the other modeling approaches. This approach recognizes that human-generated code can exhibit inconsistencies, and training the model on such varied data ensures its adaptability to diverse real-world scenarios. Striking a balance in determining the optimal value of α, respectively, the noise level, is crucial, as excessive noise might misrepresent the underlying data distribution and adversely affect the learning process. While it remains advantageous for this methodology to simulate a human influence, an inherent uncertainty exists as we lack comprehensive insight into the intricacies of the 1536-dimensional space spanned by the embedding. Consequently, the tangible impact of variations within a single dimension remains elusive, rendering any assurances regarding the precise emulation of human impact speculative.

C. Unsupervised Methods

In contrast to the previously mentioned SL or DNN models, the presented GMM is not initially designed for classification tasks but also represents a black-box method. To transfer the methodology of GMMs into a classification problem, we proceeded as follows: First, we train two independent GMMs, one exclusively on human samples and the other on GPT samples. For the prediction of an embedded sample $\vec{x}$, we then calculate the log-likelihood:

$$\log p(\vec{x}) = \log\left(\sum_{k=1}^{K} \psi_k \mathcal{N}(\vec{x}|\vec{\mu}_k, \Sigma_k)\right)$$
$$\text{with } f(x) = \begin{cases} 0 & \text{if } \log p(\vec{x}; \mathcal{G}_{\text{HU}}) \geq \log p(\vec{x}; \mathcal{G}_{\text{AI}}) \\ 1 & \text{otherwise} \end{cases},$$

where $p(\vec{x}; \mathcal{G}_{\text{AI}})$ and $p(\vec{x}; \mathcal{G}_{\text{HU}})$ denote the probability density functions of $\vec{x}$ under the GMMs, respectively. In the case of ADA, we can embed individual code snippets $\mathcal{E}(x)$ as before and let the GMM determine k clusters in them, which are described by the parameters $\vec{\mu}$ and Σ. This is different when using TF-IDF embedding, which requires prior tokenization $\mathcal{T} : \mathcal{C} \to \mathcal{C}^m$ to determine the number of different tokens in each code snippet. Thus, the embedding of TF-IDF is performed on each tokenized code snippet with $\mathcal{E}(\mathcal{T}(x))$. For tokenization, we used OpenAIs tokenizer `tiktoken` [25], which is also responsible for tokenizing text or code in GPT-4 [14]. Still, this method does not consider tokens individually but always depends on their associated code snippet. Nevertheless, the two embeddings, in combination with GMMs, achieve accuracies that are partially above 90% and, hence, outperform all white-box models.

However, the idea of using GMMs comes from approximating the probability distribution of ChatGPT. This distribution does not extend over entire code snippets but over individual tokens, which are considered for the output of the next token. Therefore, we use Word2Vec for embedding single tokens instead of whole snippets in ADA or TF-IDF. Theoretically, this approach would also work with ADA. Still, with $\approx 2.6 \times 10^6$ tokens and an average API response time of ≈ 5 seconds, the time overhead would be far too high, so this embedding is irrelevant for us. Using Word2Vec with $k = 5$ components based on this achieves near-best performance with 97.82% accuracy. Figure

Model	Unformatted			Formatted		
	Accuracy (%)	Precision (%)	Recall (%)	Accuracy (%)	Precision (%)	Recall (%)
Black-box models						
DNN	86.41 ± 0.23	85.40 ± 1.51	88.29 ± 2.53	84.69 ± 0.32	83.92 ± 1.01	88.01 ± 2.71
DNN + Ada	98.32 ± 0.31	98.08 ± 0.59	98.61 ± 1.05	93.18 ± 0.57	92.67 ± 1.80	94.52 ± 1.53
DNN + Noisy Ada	$\mathbf{98.45 \pm 0.33}$	$\mathbf{98.37 \pm 0.83}$	$\mathbf{98.85 \pm 0.48}$	$\mathbf{95.35 \pm 1.10}$	$\mathbf{95.17 \pm 1.10}$	96.31 ± 1.57
DNN + TF-IDF	97.60 ± 0.32	97.76 ± 0.60	97.50 ± 1.11	93.32 ± 0.34	93.29 ± 1.49	94.06 ± 1.59
GMM + Ada	90.09 ± 0.00	89.51 ± 0.00	90.82 ± 0.00	86.87 ± 0.00	86.00 ± 0.00	89.44 ± 0.00
GMM + Word2Vec	97.82 ± 0.00	96.77 ± 0.00	97.52 ± 0.00	94.53 ± 0.00	93.23 ± 0.00	$\mathbf{96.82 \pm 0.00}$
GMM + TF-IDF	89.99 ± 0.00	88.52 ± 0.00	89.08 ± 0.00	87.28 ± 0.00	86.92 ± 0.00	87.61 ± 0.00
White-box models						
RFC	$\mathbf{88.19 \pm 0.10}$	$\mathbf{86.30 \pm 0.17}$	91.10 ± 0.16	$\mathbf{81.81 \pm 0.12}$	$\mathbf{81.79 \pm 0.20}$	$\mathbf{83.95 \pm 0.19}$
ADAC	83.06 ± 0.00	80.02 ± 0.00	88.63 ± 0.00	74.92 ± 0.00	74.23 ± 0.00	79.80 ± 0.00
LR	80.72 ± 0.00	76.40 ± 0.00	$\mathbf{95.47 \pm 0.00}$	72.13 ± 0.00	73.40 ± 0.00	73.40 ± 0.00
XGBC	87.25 ± 0.00	84.87 ± 0.00	91.01 ± 0.00	79.99 ± 0.00	79.71 ± 0.00	82.89 ± 0.00
DTC	82.92 ± 0.22	82.85 ± 0.21	83.50 ± 0.33	74.13 ± 0.26	74.38 ± 0.25	77.17 ± 0.34
OPCT	84.44 ± 0.58	77.56 ± 4.02	91.54 ± 5.11	75.26 ± 0.68	71.99 ± 1.53	79.50 ± 2.69

TABLE IV
FINAL RESULTS - SHOWN HERE ARE THE MEAN AND STANDARD DEVIATION VALUES $\mu \pm \sigma$ FROM 10 INDEPENDENT RUNS; EACH RUN CORRESPONDS TO A DIFFERENT DISTRIBUTION OF TRAINING AND TEST SAMPLES.

6 shows the power of this technique, assigning each token with a log-likelihood of the respective GMM. Leveraging this approach enables a detailed comparison of the likelihood distribution of tokens between human-generated and GPT-generated samples. It reveals the nuanced differences in how each source generates tokens, contributing to the high accuracy achieved by the model. This level of granularity in examining token likelihoods is instrumental in distinguishing between the subtleties of natural and artificial code generation.

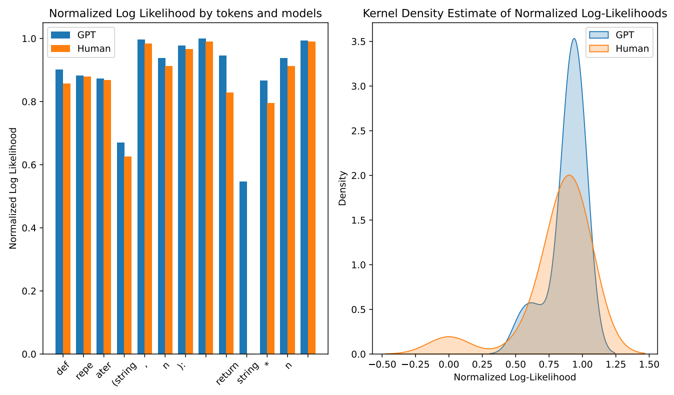


Fig. 6. NORMALIZED LOG-LIKELIHOOD AND KERNEL DENSITY OF EACH TOKEN IN GPT-GENERATED CODE SNIPPET OF GMMS: `def repeater(string):\n return string * n\n`

Similarly, an instance of code generated by ChatGPT, as depicted in Figure 6, demonstrates a distinctive characteristic. Here, the naming convention of the parameter, denoted as ”string,” is atypical for human developers. Conventionally, naming variables or parameters directly after data types is not considered best practice, elucidating a divergence in naming strategies between human and AI-generated code. This divergence is subtly mirrored in the log-likelihoods assigned by the two models and is visually represented in the adjacent density plot. The differences may seem negligible; however, a closer examination reveals that this deviation in naming convention contributes to a peak in the kernel density, serving as a pivotal factor for classifying the sample as AI-generated.

Further, motivated by the lack of interpretability in the decision-making of black-box models, we reduced the number of components k in the GMMs. We found that a single component $k = 1$, i.e., considering a simple Gaussian normal distribution with $\mathcal{N}(\mu, \sigma)$, is sufficient to distinguish, with an accuracy of 95.71%, the distributions of ChatGPT-generated and human-generated code, shown in figure 7.

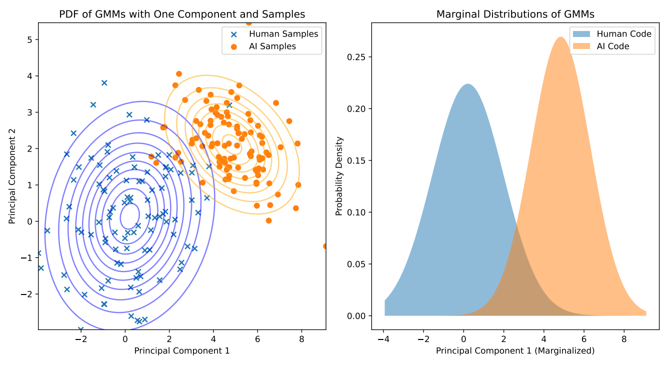


Fig. 7. COMPARISON OF GMM DISTRIBUTIONS AND MARGINAL DISTRIBUTIONS FOR HUMAN- AND AI-GENERATED CODE IN PCA REDUCED SPACE

With this visualization, we claim that two differentiated normal distributions in the reduced space spanned by PCA distinguish human and AI-generated code. Moreover, from the higher density in the middle of the AI-generated samples, we can see that these dimensions, spanned by a linear combination of the embedding space, acquire components that are a clear feature for classification as AI. This observation implies that AI-generated code tends to have more commonality or consistency in certain dimensions, potentially reflecting the model's biases or preferences, while human-generated code exhibits greater variability. The clear separation between the two distributions indicates the presence of discernible patterns and features characteristic of each source, providing a robust basis for classification.

Addressing our research questions, we conclude that the high-performing black-box models presented can differentiate between human- and ChatGPT-generated code. While DNNs have demonstrated superior performance across various embeddings, their decision-making processes remain opaque. In contrast, GMMs have not only excelled but also provided valuable insights. By mimicking ChatGPT through token-wise approximation of the probability distribution, we can infer which token selections influence classification. Remarkably, we observed that two simple normal distributions can adequately describe both classes, given the embedding and its reduced form through PCA.

VI. Discussion

Despite the promising results presented in this paper, some limitations and potential extensions warrant discussion. We will explore the strengths and weaknesses of the employed methodologies, assess the practical implications of our findings, and consider how emerging technologies and approaches could further advance this field of study.

Formatting: Upon retrospective analysis of the findings, it is identifiable that, under controlled experimental conditions, differentiating code generated by humans and code generated by AI is feasible by scrutinizing code formatting syntax. Moreover, after subjecting all samples to the Black code formatter tool [58], the evaluated classification models still exhibited the capacity to yield satisfactory outcomes. However, it is important to note that the Black formatter represents just one among several code formatting tools available, each with its distinct style and rules. Utilizing a different formatter such as AutoPEP8 [60] could potentially introduce variability in the formatting of code snippets, impacting the differentiation capability of the models. Examining the robustness of our classification models against diverse formatting styles remains an area warranting further exploration. Additionally, the adaptation of models to various formatters can lead to the enhancement of their generalization ability, ensuring consistent performance across different coding styles.

Feature Selection: Although the features we chose were quite successful for both the original and formatted code snippets, there is still a need for features that lead to higher accuracy. The exemplary performance of black-box models indicates the existence of more robust features, thereby necessitating a more profound analysis of the embedding space. Despite numerous research attempts to address this issue, success has been limited. Another not-yet-mentioned result of our work was the analysis of coding-specific features. By analyzing coding snippets, the desire for coding-specific features in which humans and machines differ is desirable. Still, with more than 50 of such features, we could only find white-box models that achieved $\approx 63\%$ accuracy at best and only guessed with $\approx 50\%$ accuracy in most experiments.

Test cases: Tested code, having undergone rigorous validation, offers a reliable and stable dataset, enhancing the model's accuracy and generalization by mitigating the risk of incorporating errors or anomalies. This reliability fosters a robust training environment, enabling the model to learn discernible patterns and characteristics intrinsic to human and AI-generated code. However, focusing solely on tested code may limit the model's exposure to diverse and unconventional coding styles or structures, potentially narrowing its capability to distinguish untested, novel, or outlier instances. Integrating untested code could enrich the diversity and comprehensiveness of the training dataset, accommodating a broader spectrum of coding styles, nuances, and potential errors, thereby enhancing the model's versatility and resilience in varied scenarios. In future work, exploring the trade-off between the reliability of tested code and the diversity of untested code might be beneficial to optimize the balance between model accuracy and adaptability across various coding scenarios.

Data availability: One implication of committing to only tested code is the availability of such code. Utilizing solely tested code might constrain the volume and variety of accessible data, as it necessitates passing through extensive validation processes, thus potentially limiting the scope of training datasets. This restriction can impede the model's learning capacity, potentially yielding a less versatile and adaptive model. Conversely, including untested code can significantly expand data availability, encompassing a more varied and comprehensive range of coding styles, structures, and potential anomalies. This increased data availability can contribute to developing a more robust and flexible model that distinguishes between human and AI-generated code across a broader spectrum of instances. Balancing the commitment to tested code with the integration of untested code is pivotal in addressing data availability concerns and ensuring the development of an effective and adaptable model. Furthermore, it is imperative to note that this experiment exclusively employed AI-generated samples generated using the `gpt-3.5-turbo` API developed by OpenAI. While this model is the most widely employed model by students, rendering it highly suitable for the objectives of this research paper, it is not the most capable of the GPT series. Consequently, the integration of additional models such as `gpt-4` [14] or `T5`$^+$ [61], which have demonstrated superior performance in coding-related tasks, can serve to not only enhance the utilization of the available data but also introduce increased variability. Particularly concerning the distributions presented in GMMs, it would be intriguing to investigate whether distinct models formulate their unique distributions or align with a generalized AI distribution. This exploration could provide pivotal insights into the heterogeneity or homogeneity of AI-generated code, thereby contributing to the refinement of methodologies employed in differentiating between human- and AI-generated instances.

Programming languages: Moreover, it is essential to underscore that this experiment exclusively encompassed Python code. However, our approach remains programming-language-agnostic, maintaining the capability to extract features or embed code snippets token-wise or as a whole to apply our methodology. More evidently, we applied the ADA + DNN approach to Java code, sourced from the Softwaretechnik 2 of our university, yielding similar performance as reported in table IV, with an accuracy of $\approx 97\%$ in distinguishing human- and AI-generated code. This inclusion served as a preliminary exploration into the model's adaptability across varying programming languages and syntax structures. Nonetheless, encapsulating the diversity of all programming languages within one model poses inherent challenges due to each language's unique syntactical characteristics.

Deployment: In section III-A, we have previously outlined two primary methods of cheating when using ChatGPT. The first approach, which involves utilizing the model's pure output, has been explored in depth. In contrast, the second method, which entails modifying existing solutions, has only been partially investigated by introducing noise to AI solutions. Examining this second method further is essential to gain comprehensive insights into how fraudulent activities can be conducted and subsequently detected. Nevertheless, within the context of our proposed framework, the models trained are demonstrably feasible for detecting fraudulent activities involving ChatGPT.

VII. Conclusion

The goal of this research was to find a classification model that is capable of differentiating AI- and human-written code samples. In order to enable the feasibility of such a model in an application, emphasis was also put on explainability. Upon thoroughly examining existing AI-based text sample detection research, we strategically transposed the acquired knowledge to address the novel challenge of identifying AI-generated code samples.

Overall, the best performance in our experiment yielded the DNN + NOISY ADA approach with an accuracy of 98% on unformatted and 95% accuracy on formatted data. Unfortunately, this model does not explain its decision process. On the other hand, the tested RFC model was able to score 88% prediction accuracy using features that depend on the formatting style of the sample. Even after counteracting the gap in format between AI and human samples by applying a formatting tool to all samples, the model only experienced a loss in accuracy of 7%. Further, other black-box approaches like the DNN and GMM show promising results, indicating differences between the two classes.

Within the structured context of our experimental framework, and through the application of the methodologies and evaluative techniques delineated in this manuscript, we have successfully demonstrated the validity of our posited hypotheses $\mathrm{H}_{\mathbf{1.0}}$, $\mathrm{H}_{\mathbf{2.0}}$, and $\mathrm{H}_{\mathbf{2.1}}$. The empirical results affirm with a significant degree of accuracy that hypotheses $\mathrm{H}_{\mathbf{1.0}}$ and $\mathrm{H}_{\mathbf{2.0}}$, postulating the discernibility between AI-generated coding style and human coding style, are substantiated. While our investigation did not extend to a detailed examination of the disparities in individual coding styles among humans, our meticulous data preprocessing yielded discernable differences. Hypothesis $\mathrm{H}_{\mathbf{2.1}}$ posits the existence of syntactical variances between AI and human code, differences that can be exploited to distinguish between the two. By employing a methodological approach focused on extracting features elucidating the syntactical formatting characteristics inherent in the samples, we were able to affirm this hypothesis. Furthermore, we enhanced the interpretability of the unsupervised classification approach embodied by the GMM model. This was achieved by conducting an in-depth analysis of individual tokens, as opposed to an examination of entire code snippets, thereby facilitating a more nuanced understanding of the decision-making processes inherent in the GMM models.

A notable outcome of this experiment is the acquisition of a comprehensive dataset comprising Python code samples produced by humans and AI, originating from various online coding task sources. This dataset serves as a valuable resource for conducting in-depth investigations into the fundamental structural characteristics of AI-generated code, and it facilitates a comparative analysis between AI-generated and human-generated code, highlighting distinctions and similarities. Moreover, within this dataset, one can access API responses encompassing explanations provided by the AI model for the code samples it generates, enhancing our understanding of its decision-making processes.

Having only experimented with Python code, further research could be focused on investigating the coding style of AI with other programming languages. Moreover, AI code samples were only created with OpenAI's `gpt-3.5-turbo` API. To make a more general statement about the coding style of language models, further research on other models should be conducted. This study, one of the first of its kind, can be used as a foundation for further research to understand how language models write code and how it differs from human-written code.

In light of the rapid evolution and remarkable capabilities of recent language models, which, while designed to benefit society, also harbor the potential for malicious use, developers and regulators must implement stringent guidelines and monitoring mechanisms to mitigate risks and ensure ethical usage. To effectively mitigate the potential misuse of these advances, the continued development of detection applications, informed by research such as that presented in this paper, remains indispensable.

References

[1] M. Alawida, S. Mejri, A. Mehmood, B. Chikhaoui, and O. Isaac Abiodun, "A comprehensive study of chatgpt: Advancements, limitations, and ethical considerations in natural language processing and cybersecurity," *Information*, vol. 14, no. 8, p. 462, 2023.

[2] A. Ziegler, E. Kalliamvakou, S. Simister, G. Sittampalam, A. Li, A. Rice, D. Rifkin, and E. Aftandilian, "Productivity assessment of neural code completion," 2022.

[3] J. Zhang, X. Ji, Z. Zhao, X. Hei, and K.-K. R. Choo, "Ethical considerations and policy implications for large language models: Guiding responsible development and deployment," *arXiv preprint arXiv:2308.02678*, 2023.

[4] S. Russell, Y. Bengio, G. Marcus, P. Stone, C. Muller, and E. Mostaque, "Pause giant ai experiments: An open letter," https://futureoflife.org/open-letter/pause-giant-ai-experiments/, 2023, accessed: 2023-08-18.

[5] ——, "Policymaking in the pause," https://futureoflife.org/wp-content/uploads/2023/04/FLI_Policymaking_In_The_Pause.pdf, 2023, accessed: 2023-08-18.

[6] B. D. Lund, T. Wang, N. R. Mannuru, B. Nie, S. Shimray, and Z. Wang, "Chatgpt and a new academic reality: Artificial intelligence-written research papers and the ethics of the large language models in scholarly publishing," *Journal of the Association for Information Science and Technology*, vol. 74, no. 5, pp. 570–581, 2023. [Online]. Available: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/asi.24750

[7] E. Mitchell, Y. Lee, A. Khazatsky, C. D. Manning, and C. Finn, "Detectgpt: Zero-shot machine-generated text detection using probability curvature," *arXiv preprint arXiv:2301.11305*, 2023.

[8] S. Gehrmann, H. Strobelt, and A. M. Rush, "Gltr: Statistical detection and visualization of generated text," *arXiv preprint arXiv:1906.04043*, 2019.

[9] H. Alamleh, A. A. S. AlQahtani, and A. ElSaid, "Distinguishing human-written and chatgpt-generated text using machine learning," in *2023 Systems and Information Engineering Design Symposium (SIEDS)*. IEEE, 2023, pp. 154–158.

[10] O. Zheng, M. Abdel-Aty, D. Wang, Z. Wang, and S. Ding, "Chatgpt is on the horizon: Could a large language model be all we need for intelligent transportation?" *arXiv preprint arXiv:2303.05382*, 2023.

[11] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," 2023.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[13] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," 2023.

[14] OpenAI, "Gpt-4 technical report," 2023.

[15] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 161–168.

[16] N. Burkart and M. F. Huber, "A survey on the explainability of supervised machine learning," *Journal of Artificial Intelligence Research*, vol. 70, pp. 245–317, 2021.

[17] R. M. Yasir and D. A. Kabir, "Exploring the impact of code style in identifying good programmers," 2023.

[18] P. Raghavan and N. El Gayar, "Fraud detection using machine learning and deep learning," in *2019 international conference on computational intelligence and knowledge economy (ICCIKE)*. IEEE, 2019, pp. 334–339.

[19] J. O. Awoyemi, A. O. Adetunmbi, and S. A. Oluwadare, "Credit card fraud detection using machine learning techniques: A comparative analysis," in *2017 international conference on computing networking and informatics (ICCNI)*. IEEE, 2017, pp. 1–9.

[20] I. Solaiman, M. Brundage, J. Clark, A. Askell, A. Herbert-Voss, J. Wu, A. Radford, G. Krueger, J. W. Kim, S. Kreps, M. McCain, A. Newhouse, J. Blazakis, K. McGuffie, and J. Wang, "Release strategies and the social impacts of language models," 2019.

[21] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," 2020.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[23] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[24] Y. Chen, H. Kang, V. Zhai, L. Li, R. Singh, and B. Raj, "Gpt-sentinel: Distinguishing human and chatgpt generated content," 2023.

[25] OpenAI, "Models," https://platform.openai.com/docs/models/overview, 2023, accessed: 2023-07-29.

[26] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[27] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[28] L. R. Medsker and L. Jain, "Recurrent neural networks," *Design and Applications*, vol. 5, no. 64-67, p. 2, 2001.

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[30] D. Jurafsky, *Speech & language processing*. Pearson Education India, 2000.

[31] S. Robertson, "Understanding inverse document frequency: on theoretical arguments for idf," *Journal of documentation*, vol. 60, no. 5, pp. 503–520, 2004.

[32] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[33] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," 2013.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[35] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," 2020.

[36] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013, vol. 398.

[37] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Cart," *Classification and regression trees*, 1984.

[38] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[39] T. Stepišnik and D. Kocev, "Oblique predictive clustering trees," *Knowledge-Based Systems*, vol. 227, p. 107228, 2021.

[40] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[41] A. L. Caterini and D. E. Chang, *Deep neural networks in a mathematical framework*. Springer, 2018.

[42] M. Hossin and M. N. Sulaiman, "A review on evaluation metrics for data classification evaluations," *International journal of data mining & knowledge management process*, vol. 5, no. 2, p. 1, 2015.

[43] D. Reynolds and R. Rose, "Robust text-independent speaker identification using gaussian mixture speaker models," *IEEE Transactions on Speech and Audio Processing*, vol. 3, no. 1, pp. 72–83, 1995.

[44] T. Moon, "The expectation-maximization algorithm," *IEEE Signal Processing Magazine*, vol. 13, no. 6, pp. 47–60, 1996.

[45] A. A. Neath and J. E. Cavanaugh, "The bayesian information criterion: background, derivation, and applications," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 4, no. 2, pp. 199–203, 2012.

[46] Y. Sakamoto, M. Ishiguro, and G. Kitagawa, "Akaike information criterion statistics," *Dordrecht, The Netherlands: D. Reidel*, vol. 81, no. 10.5555, p. 26853, 1986.

[47] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[48] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," 2021.

[49] CodeChef, "Codechef," 2023, accessed: 2023-05-17. [Online]. Available: https://www.codechef.com

[50] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.abq1158

[51] HackerEarth, "Hackerearth," 2023, accessed: 2023-09-11. [Online]. Available: https://www.hackerearth.com

[52] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021.

[53] M. Trajkovski, "Mtrajk," 2023, accessed: 2023-09-11. [Online]. Available: https://github.com/MTrajK/coding-problems

[54] S. A. Alasadi and W. S. Bhaya, "Review of data preprocessing techniques in data mining," *Journal of Engineering and Applied Sciences*, vol. 12, no. 16, pp. 4102–4107, 2017.

[55] F. Pukelsheim, "The three sigma rule," *The American Statistician*, vol. 48, no. 2, pp. 88–91, 1994.

[56] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[57] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[58] Black, "The uncompromising code formatter," https://github.com/psf/black, 2023, accessed: 2023-07-29.

[59] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[60] H. Hattori, "Autopep8," https://github.com/hhatto/autopep8, 2023, accessed: 2023-09-27.

[61] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," 2023.