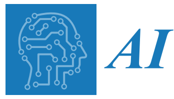

MDPI

Article

ChatGPT Code Detection: Techniques for Uncovering the Source of Code

Marc Oedingen [1], Raphael C. Engelhardt [1], Robin Denz [2], Maximilian Hammer [1] and Wolfgang Konen [1,*]

[1] Faculty of Computer Science and Engineering Science, Cologne Institute of Computer Science, TH Köln, 50678 Gummersbach, Germany; marc.oedingen@th-koeln.de (M.O.); raphael.engelhardt@th-koeln.de (R.C.E.); maximilian.hammer@smail.th-koeln.de (M.H.)

[2] Department of Medical Informatics, Biometry and Epidemiology, Faculty of Medicine, Ruhr-University Bochum, 44801 Bochum, Germany; denz@amib.rub.de

* Correspondence: wolfgang.konen@th-koeln.de

Abstract: In recent times, large language models (LLMs) have made significant strides in generating computer code, blurring the lines between code created by humans and code produced by artificial intelligence (AI). As these technologies evolve rapidly, it is crucial to explore how they influence code generation, especially given the risk of misuse in areas such as higher education. The present paper explores this issue by using advanced classification techniques to differentiate between code written by humans and code generated by ChatGPT, a type of LLM. We employ a new approach that combines powerful embedding features (black-box) with supervised learning algorithms including Deep Neural Networks, Random Forests, and Extreme Gradient Boosting to achieve this differentiation with an impressive accuracy of 98%. For the successful combinations, we also examine their model calibration, showing that some of the models are extremely well calibrated. Additionally, we present white-box features and an interpretable Bayes classifier to elucidate critical differences between the code sources, enhancing the explainability and transparency of our approach. Both approaches work well, but provide at most 85–88% accuracy. Tests on a small sample of untrained humans suggest that humans do not solve the task much better than random guessing. This study is crucial in understanding and mitigating the potential risks associated with using AI in code generation, particularly in the context of higher education, software development, and competitive programming.

Keywords: AI; machine learning; code detection; ChatGPT; large language models

Citation: Oedingen, M.; Engelhardt, R.C.; Denz, R.; Hammer, M.; Konen, W. ChatGPT Code Detection: Techniques for Uncovering the Source of Code. *AI* **2024**, *5*, 1066–1094. https://doi.org/10.3390/ai5030053

Academic Editors: Kenji Suzuki, Demos T. Tsahalis, Antony Bryant, Roberto Montemanni, Min Chen, Paolo Bellavista and Jeanine Treffers-Daller

Received: 23 May 2024
Revised: 23 June 2024
Accepted: 28 June 2024
Published: 2 July 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The recent performance of ChatGPT has astonished scholars and the general public, not only regarding its seemingly human way of using natural language but also its proficiency in programming languages. While the training data (e.g., for the Python programming language) ultimately stem from human programmers, ChatGPT has likely developed its own coding style and idiosyncrasies, just as human programmers do. In this paper, we train and evaluate various machine learning (**ML**) models on distinguishing human-written Python code from ChatGPT-written code. These models achieve very high performance even for code samples with few lines, a seemingly impossible task for humans. The following subsections present our motivation, scientific description of the task, and elaboration of research questions.

1.1. 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 [3–7].

The inherently opaque nature of these black-box LLM 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 [8]. Earlier research has extensively focused on detecting natural language (**NL**) text content generated by LLMs [9–11]. A recent survey [12] discusses the strengths and weaknesses of these approaches. 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, the ability to distinguish between human- and AI-generated code becomes increasingly important. In this case, it is not exclusively about the distinction but also the implications, including the code's trustworthiness, efficiency, and security. Moreover, the rapid development and improvement of AI models may lead to an arms race in which detection techniques must continuously evolve to stay ahead of the curve.

Earlier research has shown that using LLMs to generate code can lead to security vulnerabilities, and that 40% of such code fails to solve the given task [13]. On the other hand, using LLMs can also lead to significant increases in productivity and efficiency [2]. This dual-edged nature of LLMs necessitates a balanced approach. Harnessing the potential benefits of such models while mitigating risks is key; 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. Moreover, robust fraud detection is essential to prevent cheating in the context of examinations, ensuring that assessments accurately reflect students' capabilities and do not check the non-deterministic output of a prompt due to the stochastic decision-making of LLMs based on transformer models (**TM**) during inference. Under the assumption that AI-generated code has a higher chance of security vulnerabilities, it can also be critical to have the ability to distinguish between human- and AI-generated code beyond the educational context, for example in the software industry or when testing an unknown piece of software. As the boundaries of what AI can achieve expand, 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.

1.2. Problem Introduction

This paper delves deeply into the challenge of distinguishing between human-generated and AI-generated code, 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 differentiating between human- and AI-generated code is represented as a fundamental binary classification problem. Specifically, given a code snippet $x \in \mathcal{C}$ as input, a function $f : \mathcal{C} \to \{0,1\} = \mathcal{Y}$ is searched for that indicates whether the code's origin is human $\{0\}$ or GPT $\{1\}$. This allows us to use well-known and established ML models. We represent the code snippets as human-designed features (white-box) and embeddings (black-box) in order to apply ML models. The use of embeddings requires prior tokenization, which is carried out either implicitly by the model or explicitly by us. Hence, we use embeddings to obtain constant dimensionality across all code snippets or single tokens. Figure 1 provides an overview of our approach in the form of a flowchart, with the details explained in the following sections.

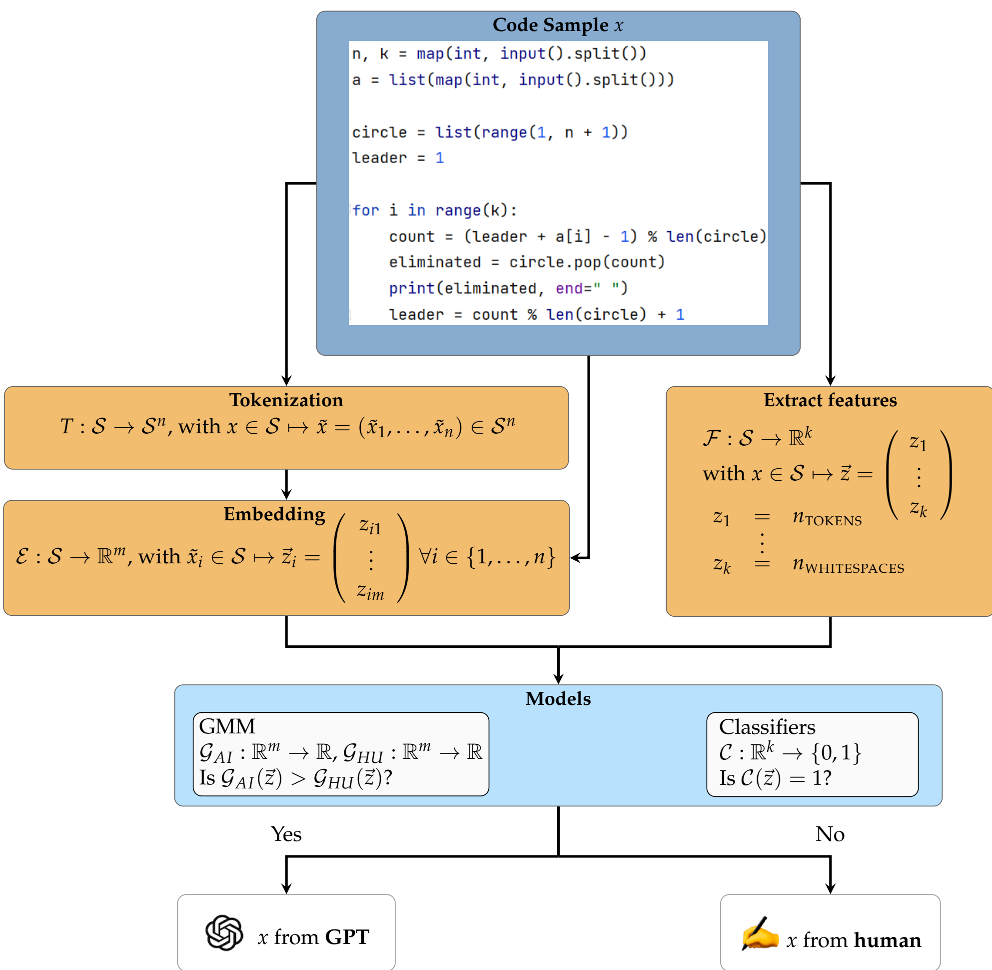

Figure 1. Flowchart of our code detection methodology.

In order for a model to truly generalize in the huge field of software development across many tasks, it requires training on vast amounts of data. However, volume alone is insufficient for a model. Data quality is paramount, as it ensures that meaningful and discriminating features are present within the code snippets. An ideal dataset would consist of code snippets that satisfy a variety of test cases to guarantee their syntactic and semantic correctness for a given task. Moreover, to overcome the pitfalls of biased data, we emphasize snippets prior to the proliferation of GPT in code generation. However, a significant blocker emerges in the stark paucity of publicly available GPT-generated solutions that match the above-mentioned criteria, underscoring the necessity of finding and generating solutions that can serve as fitting data sources for our classifiers.

1.3. Research Questions and Contributions

We formulate the following research questions based on the problem introduction above 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 (on unseen tasks) between human- and AI-generated code?*

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

- $\mathbf{H_1}$: There are detectable differences in style between AI-generated and human-generated code.

- **H_2**: These differences are only attributable to code formatting to a minor extent; even if both code snippets are formatted in the same way, there are still many detectable differences.

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 AI-generated code diverge from its human counterpart? We postulate that LLMs follow 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, and code formatting, as well as by selecting specific algorithms for certain scenarios [14]. Given the vast amount of code that the machine has seen during training, it can be anticipated that it will adopt a more generalized coding style. Thus, identifiable discrepancies between machine-generated code snippets and individual human-authored code are to be expected.

The main contributions of this paper are as follows:

1. Several classification models are evaluated on a large corpus of code data. While the human-generated code comes from many different subjects, the AI-generated code is (currently) only produced by ChatGPT-3.5.
2. The best model–feature combinations are models operating on high-dimensional vector embeddings (black-box) of the code data.
3. Formatting all snippets with the same code formatter decreases the accuracy only slightly. Thus, the format of the code is *not* the key distinguishing feature.
4. The best models achieve a classification accuracy of 98%. An explainable classifier with almost 90% accuracy is obtained with the help of Bayes classification.

1.4. Structure

The remainder of this article is structured as follows: Section 2 provides a comprehensive review of the existing literature, evaluating and discussing its current state; Section 3 offers an overview of the techniques and frameworks employed throughout this study, laying the foundation for understanding the subsequent sections; Section 4 details our experimental setup, outlining procedures from data collection and preprocessing to the training of models and their parameters; Section 5 presents the empirical findings of our investigations, followed by a careful analysis of the results; in Section 6, we provide a critical discussion and contextualization within the scope of alternative approaches; finally, Section 7 synthesizes our findings and outlines potential directions for further research.

2. Related Work

Fraud detection is a well-established area of research in the domain of AI. However, most methodologies focus on AI-generated NL content [9,10,15,16] rather than on code [17,18]. Nevertheless, findings from text-based studies remain relevant in light of the potential for cross-application and transferability of techniques.

2.1. Zero-Shot Detection

One of the most successful models for differentiating between human- and AI-generated text is DetectGPT by Mitchell et al. [9], which employs zero-shot detection, i.e., it requires neither labeled training data nor specific model training. Instead, DetectGPT follows a simple hypothesis: minor rewrites of model-generated text tend to have lower log probability under the model than the original sample, while minor rewrites of human-written text may have higher or lower log probability than the original sample. DetectGPT only requires log probabilities computed by the model of interest and random perturbations of the passage from another generic pretrained LLM (e.g., `T5` [19]). Applying this method to textual data of different origins, Mitchell et al. [9] reported very good classification results of AUC = 0.90–0.99, which was considerably better than other zero-shot detection methods on the same data.

Yang et al. [18] developed DetectGPT4Code, an adaptation of DetectGPT for code detection that also operates as a zero-shot detector, by introducing three modifications: (**1**) replacing `T5` with the code-specific `Incoder-6B` [20] for code perturbations, addressing the need for maintaining code's syntactic and semantic integrity; (**2**) employing smaller surrogate LLMs to approximate the probability distributions of closed black-box models such as `GPT-3.5` [21] and `GPT-4` [22]; and (**3**) using fewer tokens as anchors, which turned out to be better than using the full-length code. Preliminary experiments found that the ending tokens were more deterministic given enough preceding text, making for better indicators. Yang et al. [18] tested DetectGPT4Code on a relatively small set of 102 Python and 165 Java code snippets. Their results of AUC = 0.70–0.80 were clearly better than using plain DetectGPT (AUC = 0.50–0.60); however, this is still not reliable enough for practical use.

2.2. Text Detectors Applied to Code

Recently, several detectors that are good at distinguishing AI-generated from human-generated NL texts have been developed, including DetectGPT [9], GPTZero [23], and others [10,15,16,24], often achieving accuracy better than 90%. This makes it tempting to apply these NL text detectors to code snippets as well. As written above, Yang et al. [18] used DetectGPT as a baseline for their code detection method.

Two recent works [25,26] have compared a larger variety of NL text detectors on code detection tasks. Wang et al. [25] collected a large code-related content dataset, including 226,000 code snippets, to which they applied six different text detectors. When using the text detectors as-is, they only reached a low AUC $= 0.46 \pm 0.19$ on code snippets, which they considered unsuitable for reliable classification. In a second experiment, they fine-tuned an open-source detector (RoBERTa-QA [24]) by training it using a portion of their code data. It is unclear whether this fine-tuning and its evaluation used training and test samples originating from the same coding problem. Interestingly, after fine-tuning, they reported a considerably higher AUC $= 0.86 \pm 0.09$. The authors concluded that "While fine-tuning can improve performance, the generalization of the model still remains a challenge".

Pan et al. [26] provided a similar study on a medium-size database with 5000 code snippets, testing five different text detectors on their ability to recognize the origin. As a special feature, they considered thirteen variant prompts. For the tested detectors and across all variants, they reported an accuracy of of $53 \pm 7\%$, only slightly better than random choice, with a peak accuracy of 61% for the best detector.

In general, text detectors work well on NL detection tasks, but are not reliable enough on code detection tasks.

2.3. Embedding-Based and Feature-Based Methods

In modern LLMs, embeddings constitute an essential component by transforming text or code into continuous representations within a dense vector space of constant dimension, where proximity indicates similarity among elements. Hoq et al. [17] used a prior term frequency-inverse document frequency (**TF-IDF**) [27] embedding for classic ML algorithms, and used Code2Vec [28] and abstract syntax tree-based neural networks (**ASTNN**) [29] for predicting the code's origin. While TF-IDF reflects the frequency of a token in a code snippet over a collection of code snippets, Code2Vec converts a code snippet, represented as an abstract syntax tree (**AST**), into a set of path-contexts, thereby linking pairs of terminal nodes. It then computes the attention weights of the paths and uses them to compute the single aggregated weighted code vector. Similarly, ASTNN parses code snippets as an AST and uses pre-order traversal to segment the AST into a sequence of statement trees, which are further encoded into vectors with pretrained embedding parameters of Word2Vec [30]. These vectors are processed through a Bidirectional Gated Recurrent Unit (Bi-GRU) [31] to model statement naturalness, with pooling of Bi-GRU hidden states used to represent the code fragment. Hoq et al. [17] used 3.162×10^3 human-generated and 3×10^3 ChatGPT-

generated code snippets in Java from a CS1 course with a total of ten distinct problems, yielding 300 solutions per problem. Further, they selected 4×10^3 random code snippets for training and distributed the remaining samples equally on the testing and validation set. All models achieved similar accuracy, ranging from 0.90–0.95; however, the small number of unique problems, the large number of similar solutions, and their splitting procedure render the results challenging to generalize beyond the study's specific context, potentially limiting the applicability of the findings to broader scenarios.

Li et al. [32] presented an interesting work in which they generated features in three groups (lexical, structural layout, and semantic) for discrimination of code generated by ChatGPT from human-generated code. Based on these rich feature sets, they achieved detection accuracy between 0.93–0.97 with traditional ML classifiers such as random forest (**RF**) and sequential minimal optimization (**SMO**). Limitations of the method mentioned by the authors include the relatively small ChatGPT code dataset (1206 code snippets) and lack of prompt engineering (specific prompt instructions may lead to different results).

3. Algorithms and Methodology

In this section, the fundamental algorithms are presented and the approaches mandatory for our methodology are described. 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.

3.1. General Prerequisites

Detecting fraudulent use of ChatGPT in software development or coding assessment scenarios requires an appropriate dataset. Coding tasks consisting of a requirement text, human solutions, and test cases are of interest. To the best of our knowledge, one of the most used methods for detecting fraudulent usage is to represent the requirement text as the prompt for ChatGPT's input and then use the output's extracted code for submission. Tasks that fulfill the above criteria are sampled from several coding websites, and human solutions are used as a baseline to compare to the code from ChatGPT. The attached test cases were applied to both the human and AI solutions before comparison to guarantee that the code is non-arbitrary, functional, and correct. To generate code, `gpt-3.5-turbo` [33] was used, as it is the most commonly used AI tool for fraudulent content and also powers the ChatGPT application.

3.2. ChatGPT

Fundamentally, ChatGPT is a fine-tuned sequence-to-sequence learning model [34] with an encoder–decoder structure based on a pretrained transformer [21,35]. Due to its positional encoding and self-attention mechanism, it can process data in parallel rather than sequentially, unlike previous models such as recurrent neural networks [36] and long short-term memory (**LSTM**) models [37]. Although 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 to output further tokens. Interacting with the model requires the user to provide input, which is then processed and passed into the decoder, which generates the output sequence token-by-token. When a token is generated, the model incorporates this new token into the input from the preceding forward pass, continuously generating subsequent tokens until a termination criterion is met. Upon completion, the model stands by for the next user input, seamlessly integrating it with the ongoing conversation. This process effectively simulates an interactive chat with a GPT model, maintaining the flow of the conversation.

3.3. Embeddings

Leveraging the contextual representation of embeddings in a continuous and constant space allows 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 [38] incorporates an initial step of prior tokenization of code snippets, setting the foundation to capture two primary components: (**1**) term frequency (TF), which is the number of times a token appears in a code snippet, and (**2**) inverse document frequency (IDF), which reduces the weight of tokens that are common across multiple code snippets. Formally, TF-IDF is defined as

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

where N is the number of code snippets, $N_{t,d}$ is the number of times token t appears in code snippet d, N_d is the number of tokens in code snippet d, and N_t is the number of code snippets that include token t. The score emphasizes tokens that occur frequently in a particular code snippet but are less frequent in the entire collection of code snippets, thereby underlining the unique relevance of those tokens to that particular code snippet.

Word2Vec [39] is a neural network-based technique used to generate dense vector representations of words in a continuous vector space. It fundamentally operates on one of two architectures: (**1**) skip-gram (**SG**), where the model predicts the surrounding context given a word, or (**2**) continuous bag of words (**CBOW**), where the model aims to predict a target word from its surrounding context. Given a sequence of words $w_1, \ldots, w_T$, the objective is to maximize the average log probability:

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

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

where v_w and v'_w denote the input and output vector representations of word $w_i \in \mathcal{V}$ in the sequence of all words in the vocabulary, $W \in \mathbb{N}$ is the number of words in that vocabulary $\mathcal{V}$, and $c \in \mathbb{N}$ is the size of the training context. The probability of a word, given its context, is calculated using the softmax function, with $p(w_O|w_I)$. Training the model efficiently involves the use of hierarchical softmax and negative sampling to avoid the computational challenges of using softmax over large vocabularies [30].

OpenAI ADA [33] does not have an official paper, but we strongly suspect that a methodology related to OpenAI's paper [40] was used to train the model. In their approach, Neelakantan et al. [40] used a contrastive objective on semantically similar paired samples $\{(x_i, y_i)\}_{i=1}^{N}$ and an in-batch negative in training. Therefore, a transformers pretrained encoder E [35], initialized with GPT models [21,41] was used to map each pairs elements to their embeddings and calculate the cosine similarity:

$$\begin{aligned} v_x &= E([\texttt{SOS}]_x \oplus x \oplus [\texttt{EOS}]_x) \\ v_y &= E\left([\texttt{SOS}]_y \oplus y \oplus [\texttt{EOS}]_y\right) \end{aligned} \quad \text{and} \quad \text{sim}(x,y) = \frac{v_x \cdot v_y}{\|v_x\| \cdot \|v_y\|} \tag{1}$$

where $\oplus$ denotes the operation of string concatenation and EOS, SOS denotes special tokens delimiting the sequences. Fine-tuning the model includes contrasting the paired samples against in-batch negatives provided by supervised training data such as natural language inference (**NLI**) [42]. Mini-batches of M samples are considered for training, which consist of $M-1$ negative samples from NLI and one positive

example (x_i, y_i). Thus, the logits for one batch is an $M \times M$ matrix in which each logit is defined as $\hat{y} = \text{sim}(x_i, y_i) \cdot \exp(\tau)$, where τ is a trainable temperature parameter. The loss is calculated as the cross-entropy losses across each row and column direction, with positives examples lying on the diagonal of the matrix. Currently, embeddings from ADA can be obtained using OpenAIs API, `text-embedding-ada-002` [33], which returns a non-variable dimension $\tilde{x} \in \mathbb{R}^{1536}$.

3.4. 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. The supervised learning (**SL**) models employed in our analysis are listed below:

Logistic Regression (LR) [43], which makes a linear regression model usable for classification tasks.
Classification and Regression Tree (CART) [44], a well-known form of decision tree (**DT**) that offers transparent decision-making. Its simplicity, consisting of simple rules, makes it easy to use and understand.
Oblique Predictive Clustering Tree (OPCT) [45]: in contrast to regular DTs such as CART, an OPCT split at a decision node is not restricted to a single feature, but rather to a linear combination of features, cutting the feature space along arbitrary slanted (oblique) hyperplanes.
Random Forest (RF) [46]: a random forest is an ensemble method, i.e., the application of several DTs, and is subject to the idea of bagging. RF tends to be much more accurate than individual DTs due to their ensemble nature, although usually at the price of reduced interpretability.
eXtreme Gradient Boosting (XGB) [47] is an ensemble technique that aims to create a strong classifier from several weak classifiers. In contrast to RF, with its independent trees, in boosting the weak learners are trained sequentially, with each new learner attempting to correct the errors of their predecessors. In addition to gradient boosting [48], XGB employs a more sophisticated objective function with regularization to prevent overfitting and improve computational efficiency.
Deep Neural Network (DNN) [49,50]: consisting of a feedforward neural network with multiple layers, 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 in contrast to other methods their highly nonlinear nature makes them something of a "black-box" making it difficult to interpret their predictions.

3.5. Gaussian Mixture Models

In addition to SL methods, we incorporate Gaussian mixture models (**GMM**s). Generally, a GMM is characterized by a set of K Gaussian distributions $\mathcal{N}(x|\mu, \sigma)$. Each distribution $k = 1, \dots, K$ has a mean vector $\vec{\mu}_k$ and a covariance matrix Σ_k. Additionally, there are mixing coefficients ψ_k associated with each Gaussian component k, satisfying the condition $\sum_{k=1}^{K} \psi_k = 1$ to ensure that the probability is normalized to 1. Further, all components k are initialized with k-means in order to model each cluster with the corresponding Gaussian distribution. The probability density function of a GMM is defined as follows:

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

The predefined clusters serve as a starting point for optimizing the GMM with the expectation–maximization (**EM**) algorithm, which refines the model through iterative expectation and maximization steps. In the expectation step, it calculates the posterior probabilities $\hat{\gamma}_{ik}$ of data points belonging to each Gaussian component k using the current

parameter estimates according to Equation (2). Subsequently, the maximization step in Equation (3) updates the model parameters ($\hat{\psi}_k$, $\hat{\vec{\mu}}_k$, $\hat{\Sigma}_k$) to maximize the data likelihood.

$$\hat{\gamma}_{ik} = \frac{\hat{\psi}_k \mathcal{N}(\vec{x}_i | \hat{\vec{\mu}}_k, \hat{\Sigma}_k)}{\sum_{j=1}^{K} \hat{\psi}_j \mathcal{N}(\vec{x}_i | \hat{\vec{\mu}}_j, \hat{\Sigma}_j)} \tag{2}$$

$$\begin{aligned} \hat{\psi}_k &= \frac{1}{N} \sum_{i=1}^{N} \hat{\gamma}_{ik}, \quad \hat{\vec{\mu}}_k = \frac{1}{N\hat{\psi}_k} \sum_{i=1}^{N} \hat{\gamma}_{ik} \vec{x}_i \\ \hat{\Sigma}_k &= \frac{1}{(N-1)\hat{\psi}_k} \sum_{i=1}^{N} \hat{\gamma}_{ik} (\vec{x}_i - \hat{\vec{\mu}}_k)(\vec{x}_i - \hat{\vec{\mu}}_k)^\intercal \end{aligned} \tag{3}$$

The iterative repetition of this process guarantees that at least one local optimum is always achieved, and possibly even the global optimum.

4. Experimental Setup

In this section, the requirements to carry out our experiments are outlined. We cover the basic hardware and software components as well as the collection and preprocessing of data used to apply the methodology and models presented in the previous section.

For data preparation and all our experiments, we used Python version 3.10 with different packages, as delineated in our repository https://github.com/MarcOedingen/ChatGPT-Code-Detection (accessed on 27 June 2024). Due to the large amount of code snippets, we recommend a minimum of 32 GB of RAM, especially when experimenting with Word2Vec.

4.1. Data Collection

As previously delineated in the general prerequisites, an ideal dataset for our intended purposes is characterized by the inclusion of three fundamental elements: (**1**) a problem description, (**2**) one or more human solutions, and (**3**) various test cases. This is exemplified in Figure 2. The problem description (1) should clearly contain the minimum information required to solve a programming task or to generate a solution through ChatGPT. In contrast, unclear problem descriptions may lead to solutions that overlook the main problem, thereby lowering the solution quality and potentially omitting useful solutions from the limited available samples. The attached human solutions (2) for a coding problem play an important role in the subsequent analysis, and serve as a referential benchmark for the output of ChatGPT. Furthermore, a set of test cases (3) facilitates the elimination of syntactically correct solutions that do not fulfill the functional requirements specified in the problem description. To this end, a controlled environment is created in which the code's functionality is rigorously tested, preventing the inclusion of snippets of code that are based on incorrect logic or that could potentially produce erroneous outputs. Hence, we strongly focus on syntactic and executable code while ignoring a possibly typical behavior of ChatGPT in case of uncertainties or wrong answers. For certain problems, a function is expected to solve them, while others expect a console output. We have considered both by using either the function name or the entire script, referred to as the entry point, for the enclosed test cases.

Programming tasks from programming competitions are particularly suitable for the above criteria; see Table 1 for the sources of the coding problems. Coincidentally, competitions and teaching are arguably the two fields which would benefit the most from reliable detection. Due to Python having been listed consistently among the most relevant programming languages throughout the last several years (see PYPL PopularitY of Programming Languages [51] or TIOBE Index [52]) as well as to the variety and high availability of such competition tasks in Python, we decided to use this programming language. We exclusively included human solutions from a period preceding the launch of ChatGPT. We used OpenAI's `gpt3.5-turbo` API with the default parameters to generate code.

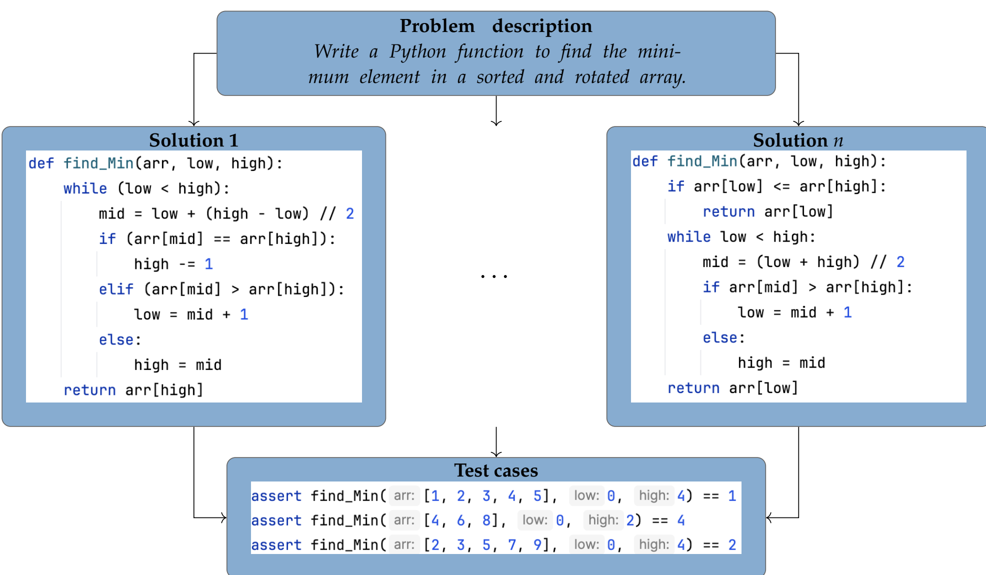

Figure 2. Example of a row in the dataset.

The OpenAI report [22] claims that `GPT-3.5` has an accuracy of 48.1% in a zero-shot evaluation for generating a correct solution on the HumanEval dataset [53]. After a single generation, our experimental verification yielded a notably lower average probability of 21.3%. Due to the low success rate, we conducted five distinct API calls for each collected problem. This strategy improved the accuracy rate considerably to 45.6%, converging towards the accuracy reported in [22] and substantiating the theory that an increment in generation attempts correlates positively with heightened accuracy levels [53]. 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 human solutions; see Table 1 in the 'before preprocessing' column for the number of problems n_{PROBLEMS}, average human solutions per problem $\bar{n}_{\text{PROBLEMS}}$, and total human samples n_{SAMPLES}.

When generating code with `gpt3.5-turbo`, the prompt strongly influences the success rate. Prompt engineering is a separate area of research that aims to utilize the intrinsic capabilities of an LLM while mitigating potential pitfalls related to unclear problem descriptions or inherent biases. During this project, we tried different prompts to increase the yield of successful solutions. Our most successful prompt, which we subsequently used, was the following: '*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, but without the solution in code.

Table 1. Code datasets overview: n_{PROBLEMS}—number of distinct problems, $\bar{n}_{\text{SOLUTIONS}}$—average number of human solutions per problem, and n_{SAMPLES}—total human samples per data source.

Dataset	Before Preprocessing			After Preprocessing		
	n_{PROBLEMS}	$\bar{n}_{\text{SOLUTIONS}}$	n_{SAMPLES}	n_{PROBLEMS}	$\bar{n}_{\text{SOLUTIONS}}$	n_{SAMPLES}
APPS [54]	1.00×10^4	21	2.10×10^5	1.95×10^3	1.1	2.17×10^3
CCF [55]	1.56×10^3	18	2.81×10^4	4.89×10^2	2.4	1.17×10^3
CC [56]	8.26×10^3	15	1.23×10^5	3.11×10^3	3.0	9.30×10^3
HAEA [57]	1.49×10^3	16	2.38×10^4	5.24×10^2	3.9	2.05×10^3
HED [53]	1.64×10^2	1	1.64×10^2	1.27×10^2	1.0	1.27×10^2
MBPPD [58]	9.74×10^2	1	9.74×10^2	6.91×10^2	1.2	8.40×10^2
MTrajK [59]	1.44×10^2	1	1.44×10^2	3.20×10^1	1.0	3.20×10^1
Sum (Average)	$\mathbf{2.26 \times 10^4}$	**(10)**	$\mathbf{3.86 \times 10^5}$	$\mathbf{7.01 \times 10^3}$	**(1.9)**	$\mathbf{1.57 \times 10^4}$

4.2. Data Preprocessing

Due to impurities in both the GPT-generated and human-generated solutions, it was necessary to preprocess the data prior to use as input for the ML models. Hence, we first extracted the code from the GPT-generated responses and checked whether both it and the human-generated solutions could be executed, reducing the whole dataset to 3.68×10^5 samples. This also eliminated missing values due to miscommunication with the API, server overloads, or the absence of Python code in the answer. Further, to prevent the overpopulation of particular code snippet subsets, duplicates in both classes were removed. A duplicate in this case is a code snippet for problem P that is identical to another code snippet for the same problem P. This first preprocessing step leaves us with 3.14×10^5 samples.

Numerically, the largest collapse for the remaining samples, and especially for the GPT-generated code snippets, was provided by the application of the test cases. This reduced the number of remaining GPT samples by 72.39% and the number of human samples by 28.59%, leaving a total of 1.71×10^5 samples. Furthermore, we considered a balanced dataset to ensure that our models would be less likely to develop biases or favor a particular class, reducing the risk of overfitting and making the evaluation of the model's performance more straightforward. Given n individual coding problems P_i, $i = 1, \ldots, n$ with h_i human solutions and g_i GPT solutions, we take the minimum $k_i = \min(h_i, g_i)$ and choose k_i random and distinct solutions from each of the two classes for P_i. The data collection and preprocessing workflow is illustrated in Figure 3.

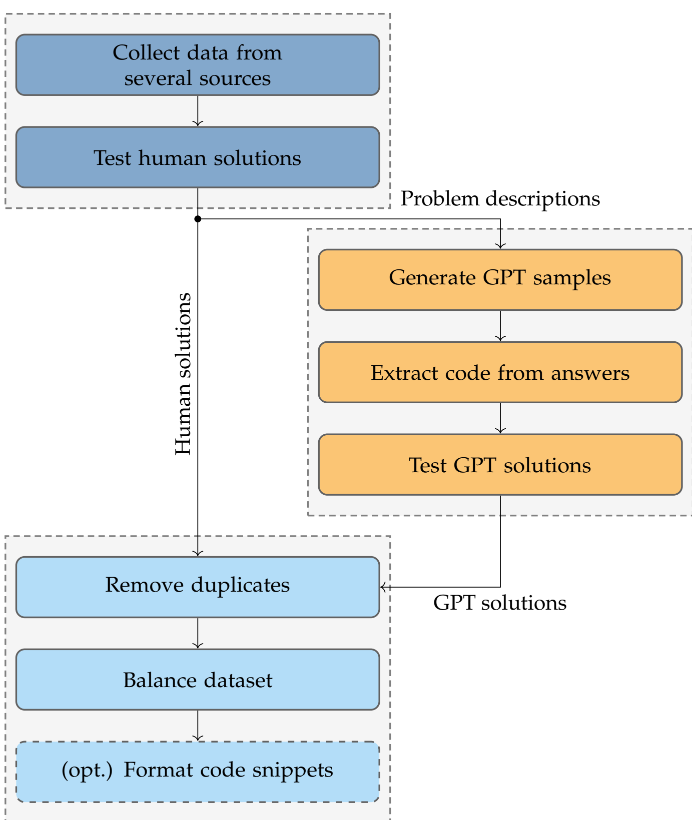

Figure 3. Flowchart of the data collection and preprocessing procedures. The boxes 'Test ... solutions' indicate that the test cases are applied to the solutions and unsuccessful solutions are discarded.

The figures for human samples after the preprocessing procedure are listed in Table 1 in the 'after preprocessing' column. Based on a balanced dataset, there are as many average GPT solutions $\bar{n}_{\text{SOLUTIONS}}$ and total GPT samples n_{SAMPLES} as human solutions and samples for each source in the final processed dataset. Thus, the preprocessed, balanced, and cleaned dataset contains 3.14×10^4 samples in total.

4.3. Optional Code Formatting

A discernible method for distinguishing between the code sources lies in the analysis of code formatting patterns. Variations in these patterns may manifest through the presence of spaces over tabs for indention purposes or the uniform application of extended line lengths. Thus, we used the Black code formatter [60], a Python code formatting tool, for both human-generated and GPT-generated code, thereby standardizing all samples into a uniform formatting style in an automated way. This methodology effectively mitigates the model's tendency to focus on stylistic properties of the code. Consequently, it allows the models to emphasize more significant features beyond mere formatting. A comparison of the number of tokens for all code snippets of the unformatted and formatted datasets is shown in Figure 4.

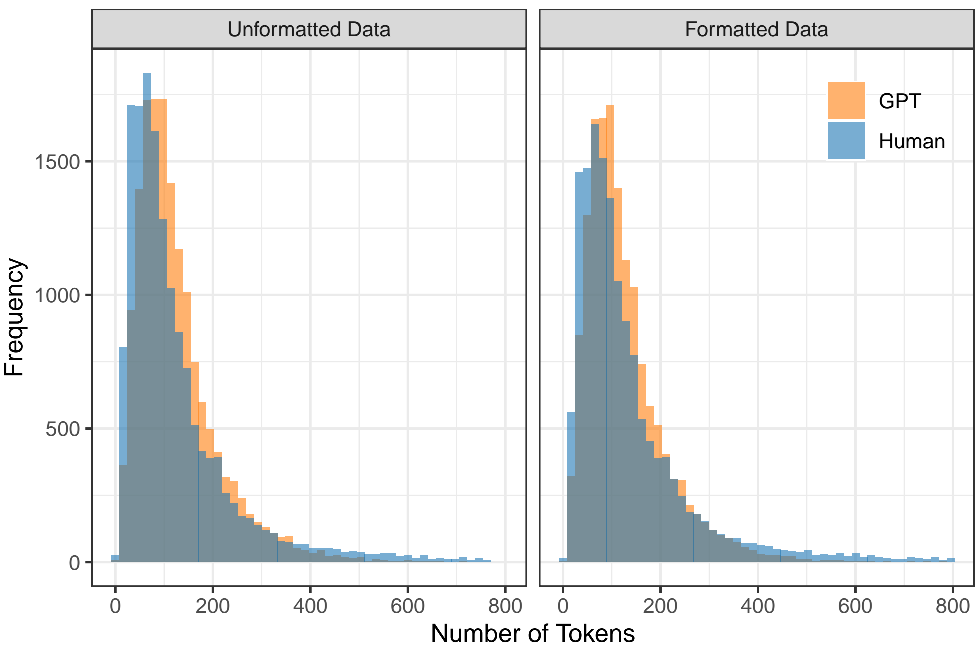

Figure 4. Distribution of the code length (number of tokens according to `cl100k_base` encoding) across the unformatted and formatted dataset. Values larger than the 99% quantile were removed to avoid a distorted picture.

4.4. Training/Test Set Separation

In dividing our dataset into training and test sets, we employed a problem-wise division instead of a sample-wise approach, allocating 80% of the problems to the training set and 20% to the test set. This decision stemmed from the structure of our dataset, which includes multiple solutions per problem. The sample-wise approach could include similar solutions for the same problem within the training and test sets. We opted for a problem-wise split in order to avoid this issue and enhance the model's generalization by ensuring that the model was tested on unseen problem instances. Additionally, each experiment was repeated ten times for statistical reliability, using another seed each time to ensure a different distribution of problems in the training and test sets.

4.5. Modeling Parameters and Tokenization

For all SL methods, we used the default parameters proposed by scikit-learn [61] for RF, GB, LR, and DT, those of xgboost [47] for XGB, those of spyct [45] for OPCT, and those of TensorFlow [62] for DNN, with two notable exceptions for DNNs (**1**) and OPCTs (**2**). For DNNs (1), we configured the network architecture to `[1536, 768, 512, 128, 32, 8, 1]`, employing the `relu` activation function across all layers except for the output layer, where `sigmoid` was used alongside binary cross-entropy as the loss function. In response to the

strong fluctuation of the OPCTs (2), we created ten individual trees and selected the best of them. Additionally, we standardized the number of Gaussian components $K = 2$ for all experiments with GMMs.

While embedding ADA uses an internal tokenization, which is the `cl100k_base` encoding, we must explicitly tokenize the prepared formatted or unformatted code for TF-IDF and Word2Vec. We decided to use the same tokenization `cl100k_base` encoding implemented in the `tiktoken` library [33], which was uniformly applied across all code snippets. Considering the fixed size of the embedding of `text-embedding-ada-002` with $\tilde{x} \in \mathbb{R}^{1536}$, we adopted this dimensionality for the other embeddings as well. For TF-IDF, we retained sklearn's default parameters, while for gensim's [63] Word2Vec we adjusted the threshold for a word's occurrence in the vocabulary to $\texttt{min_count} = 1$ and used CBOW as the training algorithm.

5. Results

In this section, the primary outcomes from deploying the models introduced in Section 3 are presented, with the models operating on the preprocessed dataset as shown in Section 4. We discuss the impact of different kinds of features (human-designed vs. embeddings) and the calibration of ML models. We then put the results into perspective by comparing them to the performance of untrained humans and a Bayes classifier.

5.1. Similarities between Code Snippets

The representation of the code snippets as embeddings describes a context-rich and high-dimensional vector space. However, the degree of similarity among code snippets within this space remains to be determined. Based on our balanced dataset and the code's functionality, we assume that the code samples are very similar. They are potentially even more similar when a code formatter, e.g., the Black code formatter [60], is used, which presents the models with considerable challenges in distinguishing subtle differences.

Mathematically, similarities in high-dimensional spaces can be particularly well calculated using cosine similarity. Let $H_P, G_P \in \mathcal{C}$ be code snippets for problem P originating from humans and GPT, respectively. The cosine similarity equivalent to Equation (1) is computed as $\text{sim}(H_P, G_P) = \frac{H_P \cdot G_P}{\|H_P\| \|G_P\|}$. Concerning the embeddings of all formatted and unformatted code snippets generated by ADA, the resulting distributions of cosine similarities are presented in Figure 5. This allows us to mathematically confirm our assumption that the embeddings of the codes are very similar. The cosine similarities for both the embeddings of the formatted and unformatted code samples in the ADA-case are approximately normally distributed, resulting in very similar mean and standard deviation: $\bar{x}_{\text{FORM}} = 0.859 \pm 0.065$ and $\bar{x}_{\text{UNFORM}} = 0.863 \pm 0.067$. Figure 5 also shows the TF-IDF embeddings for both datasets. In contrast to the ADA embeddings, significantly lower cosine similarities can be identified. We suspect that this discrepancy arises because TF-IDF embeddings are sparse and based on exact word matches. In contrast, ADA embeddings are dense and capture semantic relationships and context. Finally, the average cosine similarities in the TF-IDF-case are $\bar{y}_{\text{FORM}} = 0.316 \pm 0.189$ for the formatted dataset and $\bar{y}_{\text{UNFORM}} = 0.252 \pm 0.166$ for the unformatted dataset.

Figure 5 demonstrates that the cosine similarities of embeddings vary largely depending on the type of embedding. However, as the results in Section 5.3 show, ML models can effectively detect the code's origin from those embeddings. Unfortunately, embeddings are black-box, in the sense that the meaning of certain embedding dimensions is not explainable to humans. Consequently, we also investigated features that can be interpreted by humans in order to avoid the black-box setting with embeddings.

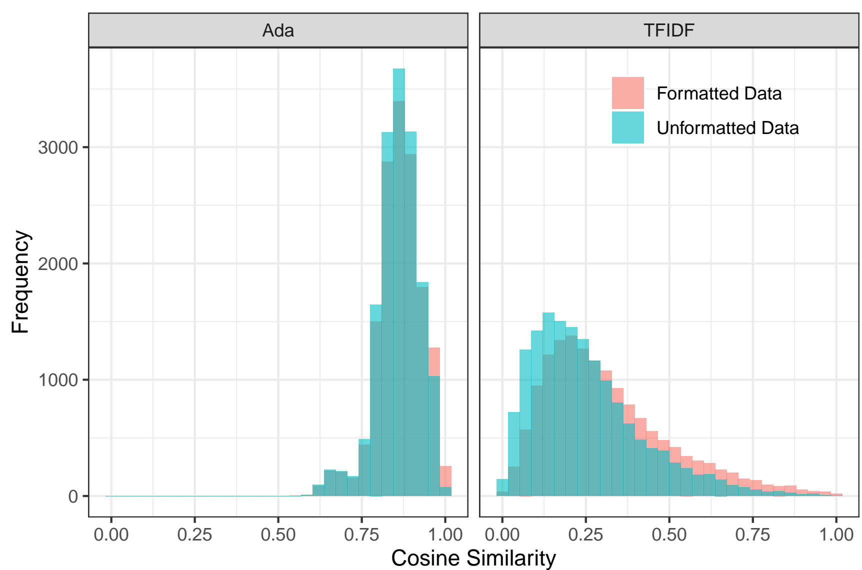

Figure 5. Cosine similarity between all human and GPT code samples embedded using ADA and TFIDF, both formatted and unformatted.

5.2. Human-Designed Features (White-Box)

A possible and comprehensible differentiation of the code samples can be attributed to their formatting. Even if these differences are not immediately visible to the human eye, they can be determined with the help of calculations. To illustrate this, we have defined the features in Table 2. Their applicability is assessed using the SL models presented in Section 3.4. The results for the unformatted samples are displayed in Table 3 and the results for the formatted samples in Table 4.

Table 2. Detailed description of all human-designed features.

Feature	Abbreviation	Description
Number of leading whitespaces	n_{LW}	The code sample is divided into individual lines, and the number of leading whitespaces is summed up.
Number of empty lines	n_{EL}	The code sample is divided into individual lines, and the number of empty is summed up.
Number of inline whitespaces	n_{IW}	The code sample is divided into individual lines, and the number of spaces within the trimmed content of each line is summed up.
Number of punctuations	n_{PT}	The code sample is filtered using the regular expression [^\w\s] to isolate punctuation characters, which are then counted.
Maximum line length	n_{ML}	The code sample is split by every new line, and the maximum count of characters of a line is returned.
Number of trailing whitespaces	n_{TW}	The code sample is divided into individual lines, and the number of trailing whitespaces is summed up.
Number of lines with leading whitespaces	n_{LWL}	The code sample is split by every new line, and all lines starting with whitespaces are summed up.

We find that the selected features capture a large proportion of the differences on the unformatted dataset. The XGB model stands out as the most effective, achieving an average

accuracy of 88.48% across various problem splits. However, when assessing the formatted dataset there is a noticeable performance drop, with the XGB model's effectiveness decreasing by approximately 8 percentage points across all metrics. While outperforming all other models on the unformatted dataset, XGB slightly trails behind the RF model on the formatted dataset, where the RF model leads with an accuracy of 80.50%. This aligns with our expectations that the differences will be minimized when employing formatting. While human-designed features are valuable for distinguishing unformatted code, their effectiveness diminishes significantly when formatting variations are reduced. This fact is further supported by Figure 6, which reflects the normalized values of the individual features from Table 2 over the two datasets. The figure demonstrates an alignment of the feature value distributions for humans and ChatGPT after formatting, which is especially evident in features such as n_{LWL} and n_{TW}, which show nearly identical values post-formatting.

Table 3. Final results. Shown are the mean $\mu \pm$ standard deviation σ from ten independent runs on **unformatted** data; each run corresponds to a different seed, i.e., different distribution of training and test samples. **Boldface**: column maximum of μ in the 'Human-designed (white-box)' part and for each individual embedding model in the 'Embedding (black-box)' part.

	Unformatted				
Model	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1 (%)**	**AUC (%)**
	Human-designed (white-box)				
CART	82.54 ± 0.23	82.70 ± 0.36	82.29 ± 0.57	82.49 ± 0.26	82.54 ± 0.23
DNN	85.37 ± 0.80	84.58 ± 2.73	86.84 ± 4.77	85.54 ± 1.21	93.73 ± 0.42
GMM	79.68 ± 0.50	74.74 ± 0.66	89.74 ± 0.58	81.55 ± 0.39	89.05 ± 0.25
LR	76.69 ± 0.63	74.80 ± 0.64	80.50 ± 0.94	77.54 ± 0.63	84.57 ± 0.41
OPCT	84.12 ± 0.60	80.89 ± 1.95	89.52 ± 2.37	84.94 ± 0.43	89.60 ± 0.81
RF	88.10 ± 0.40	87.29 ± 0.41	89.19 ± 0.66	88.23 ± 0.41	95.34 ± 0.22
XGB	**88.48 ± 0.26**	**87.39 ± 0.46**	**89.93 ± 0.42**	**88.64 ± 0.24**	**95.59 ± 0.20**
	Embedding (black-box)				
	ADA				
CART	80.82 ± 0.50	80.63 ± 0.55	81.46 ± 0.67	79.83 ± 1.03	80.82 ± 0.50
DNN	**97.79 ± 0.36**	**97.40 ± 0.77**	**98.22 ± 0.60**	**97.80 ± 0.35**	**99.76 ± 0.05**
GMM	92.71 ± 0.39	95.10 ± 0.48	90.06 ± 0.66	92.51 ± 0.42	97.37 ± 0.21
LR	95.87 ± 0.13	95.93 ± 0.13	94.60 ± 0.24	97.30 ± 0.25	99.06 ± 0.07
OPCT	95.53 ± 0.27	95.59 ± 0.25	94.35 ± 0.81	96.87 ± 0.65	97.24 ± 0.46
RF	92.39 ± 0.35	92.55 ± 0.33	90.67 ± 0.63	94.52 ± 0.55	97.85 ± 0.14
XGB	95.05 ± 0.23	95.09 ± 0.22	94.30 ± 0.42	95.89 ± 0.30	98.94 ± 0.09
	TF-IDF				
CART	95.82 ± 0.31	95.82 ± 0.31	95.93 ± 0.26	95.71 ± 0.52	95.82 ± 0.31
DNN	97.61 ± 0.30	97.79 ± 0.79	97.45 ± 1.17	97.61 ± 0.32	99.68 ± 0.05
GMM	94.60 ± 0.32	95.24 ± 0.26	93.89 ± 0.60	94.56 ± 0.34	96.18 ± 0.26
LR	97.06 ± 0.24	97.09 ± 0.24	96.11 ± 0.28	98.09 ± 0.31	99.46 ± 0.05
OPCT	96.72 ± 0.26	96.72 ± 0.26	96.78 ± 0.50	96.66 ± 0.50	97.62 ± 0.41
RF	98.04 ± 0.11	**98.04 ± 0.11**	97.77 ± 0.18	**98.31 ± 0.23**	99.72 ± 0.05
XGB	**98.28 ± 0.09**	97.87 ± 0.15	**98.70 ± 0.22**	98.28 ± 0.09	**99.84 ± 0.02**
	WORD2VEC				
GMM	93.57 ± 0.20	92.49 ± 0.37	94.86 ± 0.31	93.66 ± 0.19	94.14 ± 0.14

Table 4. Same as Table 3 but for **formatted** data.

	Formatted				
Model	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1 (%)**	**AUC (%)**
			Human-designed (white-box)		
CART	72.74 ± 0.58	72.94 ± 0.54	72.31 ± 1.24	72.62 ± 0.72	72.75 ± 0.57
DNN	77.51 ± 0.89	78.15 ± 3.65	77.18 ± 6.72	77.31 ± 2.33	86.69 ± 0.30
GMM	74.09 ± 0.65	73.37 ± 1.78	75.83 ± 2.45	74.53 ± 0.53	82.17 ± 0.50
LR	73.03 ± 0.56	73.22 ± 0.77	72.65 ± 0.68	72.93 ± 0.52	80.61 ± 0.36
OPCT	73.63 ± 1.50	69.81 ± 3.70	$\mathbf{84.38 \pm 4.88}$	76.18 ± 0.85	80.78 ± 1.76
RF	$\mathbf{80.50 \pm 0.43}$	$\mathbf{81.83 \pm 0.49}$	78.42 ± 0.69	80.09 ± 0.47	$\mathbf{88.86 \pm 0.30}$
XGB	80.29 ± 0.54	80.57 ± 0.62	79.84 ± 0.69	$\mathbf{80.20 \pm 0.55}$	88.65 ± 0.35
			Embedding (black-box)		
			ADA		
CART	74.33 ± 0.58	73.89 ± 0.58	75.19 ± 0.67	72.63 ± 0.68	74.33 ± 0.58
DNN	$\mathbf{93.14 \pm 0.87}$	$\mathbf{92.46 \pm 2.95}$	$\mathbf{94.12 \pm 2.21}$	$\mathbf{93.22 \pm 0.69}$	$\mathbf{98.66 \pm 0.10}$
GMM	86.10 ± 0.40	88.16 ± 0.55	83.41 ± 0.55	85.71 ± 0.41	93.03 ± 0.31
LR	91.16 ± 0.35	91.21 ± 0.35	90.72 ± 0.48	91.70 ± 0.56	97.29 ± 0.17
OPCT	89.99 ± 0.40	90.03 ± 0.41	89.67 ± 1.42	90.44 ± 1.72	94.56 ± 0.86
RF	85.89 ± 0.34	85.94 ± 0.37	85.63 ± 0.59	86.26 ± 0.89	94.25 ± 0.18
XGB	89.62 ± 0.15	89.63 ± 0.18	89.58 ± 0.25	89.67 ± 0.48	96.72 ± 0.14
			TF-IDF		
CART	89.69 ± 0.41	89.65 ± 0.43	90.01 ± 0.43	89.30 ± 0.67	89.69 ± 0.41
DNN	93.17 ± 0.29	93.48 ± 0.90	92.83 ± 1.06	93.14 ± 0.30	98.36 ± 0.16
GMM	87.53 ± 0.52	88.14 ± 0.61	86.74 ± 0.67	87.43 ± 0.53	91.01 ± 0.58
LR	92.50 ± 0.36	92.52 ± 0.37	92.32 ± 0.37	92.72 ± 0.58	97.95 ± 0.22
OPCT	90.74 ± 0.57	90.78 ± 0.53	90.48 ± 1.61	91.12 ± 1.51	94.76 ± 0.64
RF	93.36 ± 0.37	93.29 ± 0.38	$\mathbf{94.33 \pm 0.36}$	92.27 ± 0.44	98.55 ± 0.15
XGB	$\mathbf{93.98 \pm 0.44}$	$\mathbf{93.98 \pm 0.44}$	93.96 ± 0.51	$\mathbf{94.01 \pm 0.48}$	$\mathbf{98.80 \pm 0.16}$
			WORD2VEC		
GMM	87.92 ± 0.45	87.61 ± 0.82	88.35 ± 0.95	87.97 ± 0.45	89.26 ± 0.41

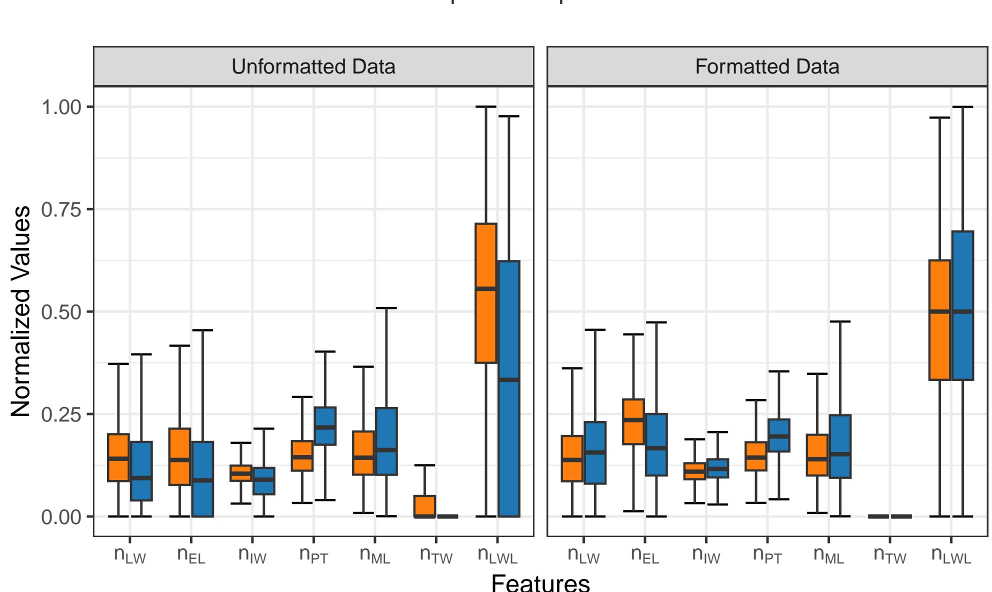

Figure 6. Box plot of human-designed features for formatted and unformatted dataset.

Despite their astonishing performance at first glance, human-designed features do not lead to near-perfect classification results. This is likely due to the small number of features and the fact that they do not capture all available information. ML models that use much higher-dimensional (richer) embedding spaces address these limitations by capturing implicit patterns not readily recognizable by human-designed features.

5.3. Embedding Features (Black-Box)

ML models operating on embeddings achieve superior performance compared to models using the human-designed features; see Tables 3 and 4 for results on the unformatted and formatted datasets, respectively. The results on the unformatted dataset show that the highest values are achieved by XGB + TF-IDF, with an accuracy greater than 98% and an astonishing AUC value of 99.84%. With a small gap to XBG + TF-IDF across all metrics, RF + TF-IDF is in second place. With less than one percentage point difference to XGB + TF-IDF, DNN + ADA is in third place. The models achieved an identical top three ranking on the formatted dataset. As with the human-designed features, *'formatted'* shows weaker performance than *'unformatted'*, with a deterioration of about 4 percentage points on all metrics except AUC, which only decreased by about 1 percentage point. In a direct comparison of models based on either human-designed features or embeddings, the best-performing models of each type show a difference of approximately 10 and 14 percentage points on the unformatted and formatted datasets, respectively, across all metrics except for the AUC score.

The disparities in performance between the white-box and black-box approaches, despite employing identical models, highlight the significance of embedding techniques. While traditional feature engineering is based on domain-specific expertise or interpretability, it often cannot capture the complex details of code snippets as effectively as embedding features.

5.4. Gaussian Mixture Models

Although GMMs are in principle capable of unsupervised learning, they reach better classification accuracy in supervised or semi-supervised settings; while the class labels are provided during training, the assignment of data points to one of the K GMM components has to be found by the EM method described in Section 3.5. Our method proceeds as follows. First, we train two independent GMMs, one exclusively on human samples and the other on GPT samples. For prediction of an embedded sample $\vec{x}$, we then calculate the likelihood as follows:

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

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 respective GMMs, and assign $\vec{x}$ to the class with the higher probability.

When using ADA, the embedding can be performed on individual code snippets x directly before the GMM finds clusters in the embeddings $\mathcal{E}(x)$. When applying TF-IDF embedding, tokenization $\mathcal{T}$ is required instead to determine the number of tokens in each snippet, then the embeddings are computed on the tokenized code snippets $\mathcal{E}(\mathcal{T}(x))$. With both embeddings, GMMs achieve accuracies of over 90%, outperforming any of the models based on human-designed features (see Table 3).

The underlying concept of GMMs is to approximate the probability distributions of ChatGPT, which do not extend over entire snippets but are limited to the individual tokens used to predict the following token. Therefore, it seems natural to apply Word2Vec for the embedding of single tokens instead of the snippet-level embedding used in ADA and TF-IDF. It should be noted that while this approach would technically also work for

ADA, the computational overhead makes it infeasible; as a rough estimate we can consider $\approx 2.6 \times 10^6$ tokens and an average API response time of ≈ 5 s. With the single-token Word2Vec embedding, the GMMs reach an accuracy of 93.57%.

To summarize, in Figure 7 we show the box plots for all our main results. Each box group contains all models trained on a particular (feature set, format) combination. Within a certain format choice, the box plots for the embedding feature sets do not overlap with those for the human-designed feature set. This shows that the choice of embedding vs. human-designed is more important than the specific ML model.

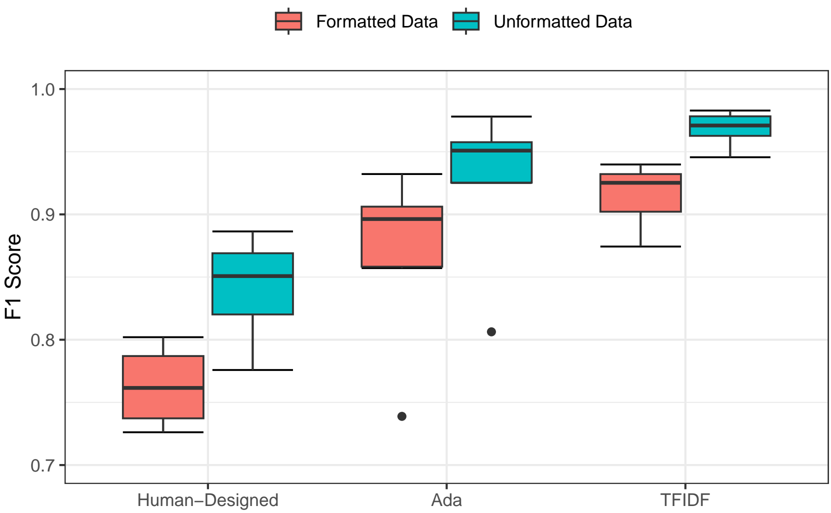

Figure 7. Distribution of the F1 Scores of all considered algorithms in different conditions. In both ADA cases, the single outlier is CART.

5.5. Model Calibration

Previously, we focused on how well the considered algorithms can discriminate between code snippets generated by humans and GPT-generated code snippets. This is an important aspect when judging the performance of these algorithms; however, it also reduces the problem to one of classification, e.g., the algorithm only tells us whether a code snippet is generated by a human or by GPT. In many cases, we may also be interested in more nuanced judgments, e.g., in how likely it is that a code snippet is GPT-generated or human-generated. All the considered algorithms are theoretically able to output such predicted class probabilities. In this section, we evaluate how well these predicted probabilities correspond to the proportion of actual observed cases.

We mainly use calibration plots to do this for a subset of the considered algorithms. In such plots, the predicted probability of a code snippet being generated by GPT is shown on the x-axis, while the actually observed value (0 if human-generated, 1 if GPT generated) is shown on the y-axis. A simple version of this plot would divide the predicted probabilities into categories, for example, ten equally wide ones [64]. The proportion of code snippets labeled as GPT inside those categories should ideally be equal to the mean predicted probability inside this category. For example, in the 10–20% category, the proportion of actual GPT samples should roughly equal 15%. We use a smooth variant of this plot by calculating and plotting a non-parametric locally weighted regression (LOESS) instead, which does not require the use of arbitrary categories [65].

Figures 8 and 10 show these calibration plots for each algorithm separately for the formatted and unformatted data. Most of the considered algorithms show adequate calibration, with the estimated LOESS regression being close to the line that passes through the origin. There are only minor differences between algorithms fitted on formatted vs. unformatted data. For certain algorithms such as RF + ADA, RF + TF-IDF, GMM + WORD2VEC, and GMM + TF-IDF, however, there seem to be issues with the calibration for predicted probabilities between 0.15 and 0.85. One possible reason for this is not a lack of calibration,

but rather a lack of suitable data points to correctly fit the LOESS regression line. For example, the GMM-based models almost always predict only probabilities very close to 0 or 1. This is not necessarily a problem of calibration if the algorithm is correct most of the time, but may lead to unstable LOESS results [65].

Therefore, we additionally plotted kernel density estimates of the predicted probabilities for each algorithm to show the range of predictions made by each (Figures 9 and 11). As can be seen quite clearly, most of the DNN- and GMM-based models relying on embeddings generated only a very few predicted probabilities between 0.1 and 0.9, making the validity of the calibration curves in these ranges questionable for those algorithms. Because these models do not discriminate perfectly between the two classes, they may not be the best choice when the main interest lies in predicting the probability of a code snippet being generated by GPT, as their output always suggests certainty even when it is wrong. On the other hand, algorithms such as XGB+TF-IDF or XGB+ADA show nearly perfect calibration and very high accuracy.

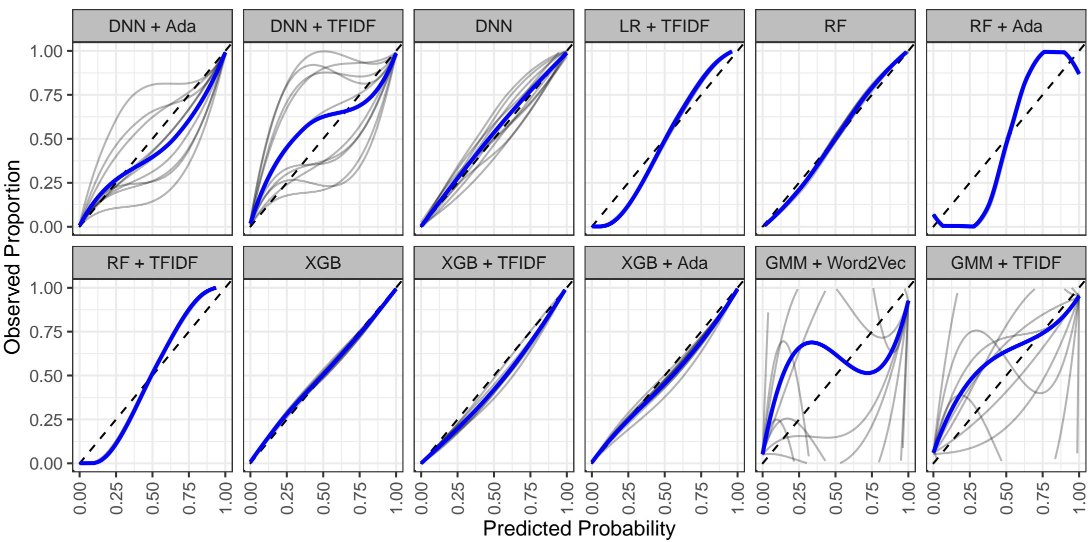

Figure 8. Calibration plots for selected classifiers trained and evaluated using **unformatted** test set data. The lines were estimated using LOESS regression models (gray: individual runs, blue: mean over all runs). The dashed diagonal line indicates perfect calibration.

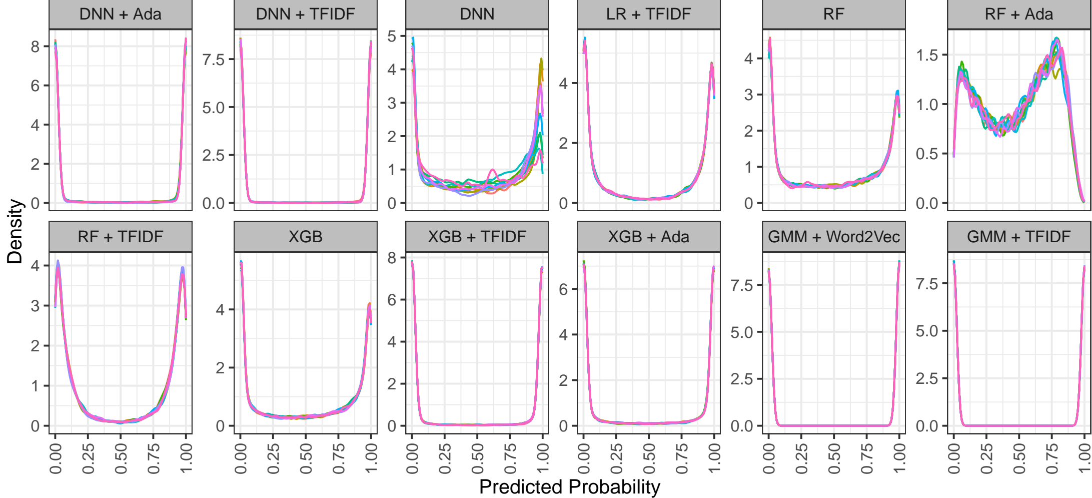

Figure 9. Kernel density estimates of the probabilities predicted by each considered classifier on the **unformatted** test set. The plot contains one line per run, each run in a different color.

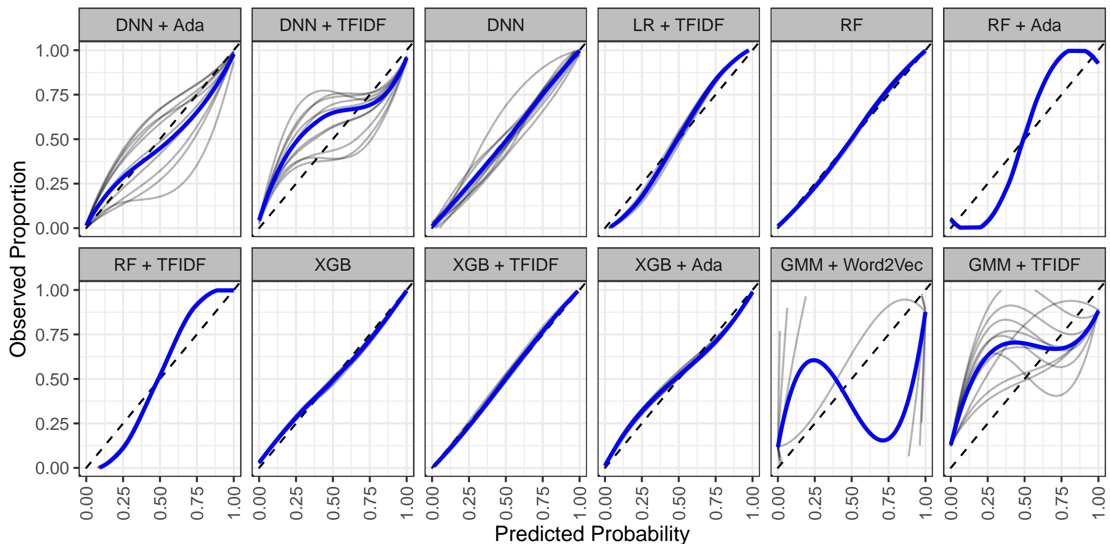

Figure 10. Calibration plots for selected classifiers trained and evaluated using **formatted** test set data. The lines were estimated using LOESS regression models (gray: individual runs, blue: mean over all runs). The dashed diagonal line indicates perfect calibration.

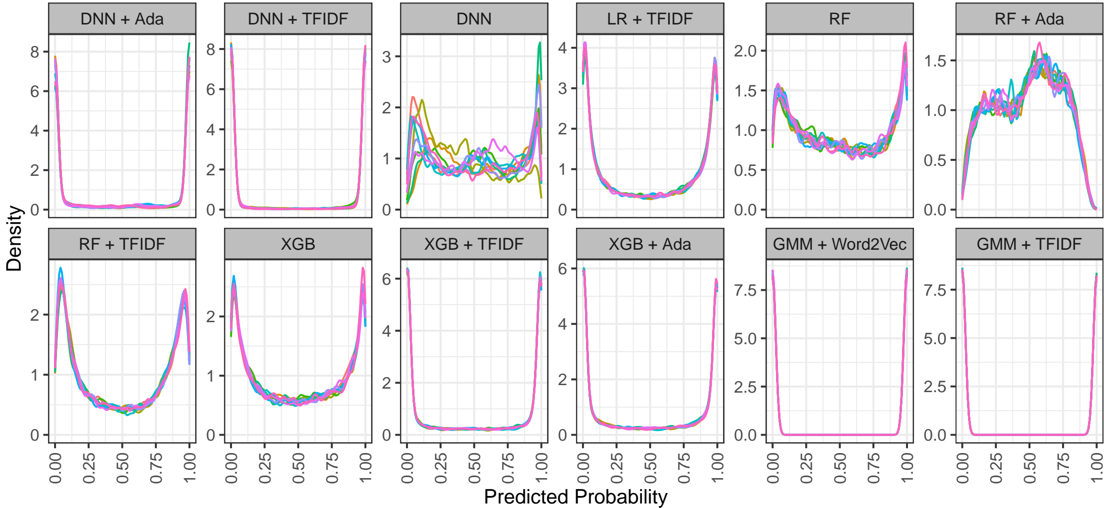

Figure 11. Kernel density estimates of the probabilities predicted by each considered classifier in the **formatted** test set. The plot contains one line per run, each run in a different color.

5.6. Human Agents and Bayes Classifiers

To gain a better understanding about the “baseline” performance, to which we compare the more sophisticated models, we briefly present the results of untrained humans and a Bayes classifier.

5.6.1. Untrained Human Agents

To assess how difficult it is for humans to classify Python code snippets according to their origin, we conducted a small study with 20 participants. Using Google Forms, the individuals were asked to indicate their educational background and experience with Python and to self-asses their programming proficiency on a 10-point Likert scale. Afterwards, they were asked to judge whether 20 randomly selected and ordered code snippets were written by humans or by ChatGPT. The dataset was balanced, but the participants were not told this. The participants, of whom 50% had a Master’s degree and 40% a PhD degree, self-rated their programming skills at an average of 6.85 ± 1.69 with a median

of 7, and indicated an average of 5.05 ± 4.08 (median 5) years of Python programming experience. The results in Table 5 show a performance slightly below random guessing, confirming this task's difficulty for untrained humans. Figure 12 shows the distribution of the participants' performances.

Table 5. Results of human performance on the classification task.

Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
48.7 ± 14.7	48.4 ± 14.8	48.0 ± 18.9	47.6 ± 16.0

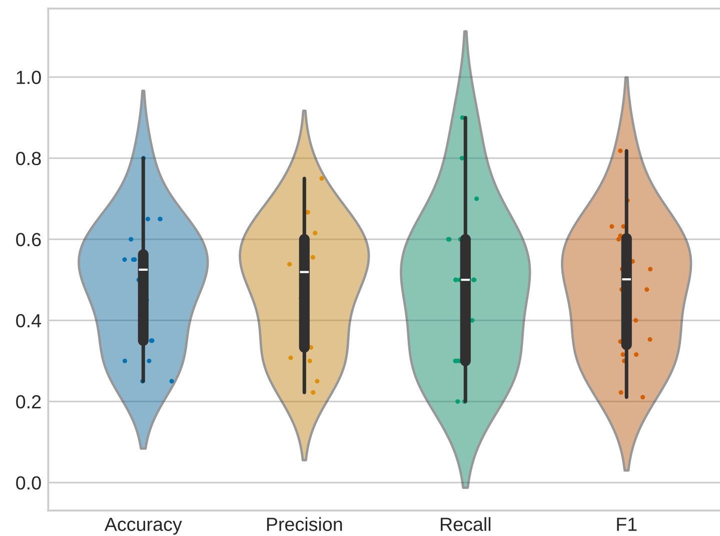

Figure 12. Performance distribution of the study participants.

In order to assess the statistical significance of these results, we first calculated the 95% confidence interval for the mean accuracy of the human participants, under the assumption that the accuracy results would be t-distributed. This resulted in a confidence interval of $[41.67\%, 55.82\%]$. Furthermore, we tested the null hypotheses *"untrained humans have a mean classification accuracy of* 60%, 70%, 80%*"* on the measured data of our sample of 20 programmers. It was found that all these hypotheses can be rejected at significance levels of 5% and 1%, with the p-values being 0.003539, 4.927×10^{-6}, 1.842×10^{-8}. To summarize, untrained people are statistically significantly worse than all models presented in Section 5.

It is worth noting that Wang et al. [25] performed a similar study with 27 participants on a completely different code dataset and found a nearly identical accuracy of 47.1%.

When comparing these results to those obtained by ML models, it should be taken into account that the participants were not trained for this task, and as such can be considered zero-shot learners. How well humans can be trained for this task with the help of labeled data has yet to be investigated.

5.6.2. Trained Bayes Classifier

Viewed from a broader perspective, the question raised by the results in the preceding subsection asks why ML models with an F1 score between 83% and 98% excel so much over (untrained) humans, who show a performance close to random guessing. In the following, we investigate the possibility (which also appeared briefly in Li et al. [32]) that ML models might use subtle differences in conditional probabilities for the appearance of tokens in their decision-making. Such probabilities might be very complex for humans to calculate, memorize, and combine.

To demonstrate this, we build a simple Bayes classifier. In the following, we abbreviate H = human origin, G = GPT origin, and X = either origin. For each token t_k in our

training dataset, we calculate the probabilities $P(t_k|X)$. For reliable estimates, we keep only those tokens t_k where the absolute frequencies $n(t_k|H)$ and $n(t_k|G)$ are both greater than or equal to equal some predefined threshold τ. The set of those tokens above the threshold is denoted as T. For illustration, Figure 13 shows the tokens with the largest probability ratio.

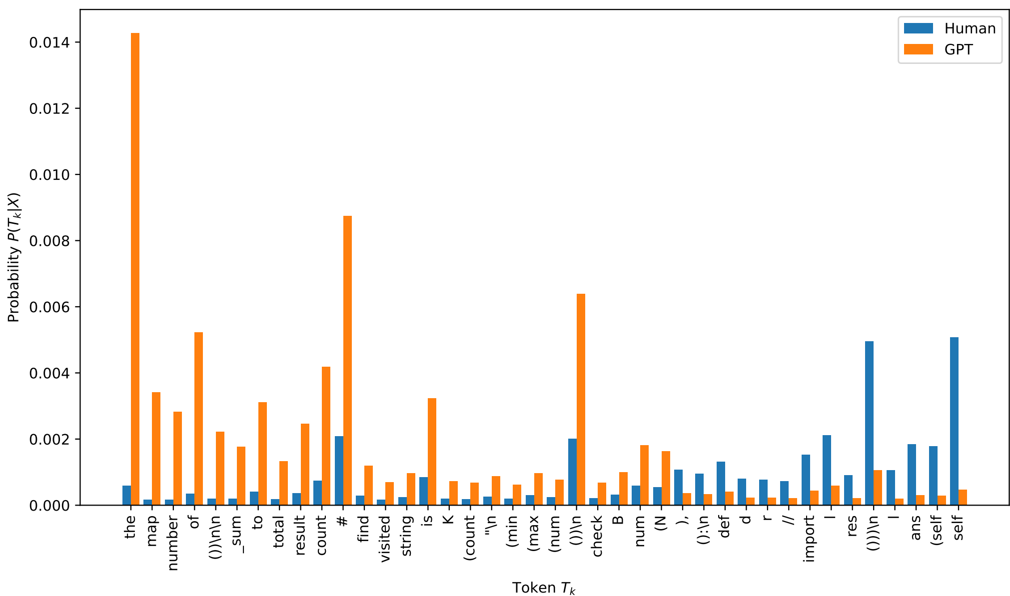

Figure 13. Top 40 tokens with the largest absolute discrepancies in their log probabilities (corresponding to the largest ratio of probabilities).

Given a new code document D from the test dataset, we determine the intersection $D \cap T$ of tokens and enumerate the elements in this intersection as $\{T_1, \ldots, T_K\}$, where T_k means the event "Document D contains token t_k". We assume statistical independence: $P(\cap_{k=1}^{K} T_k|X) = \prod_{k=1}^{K} P(T_k|X)$. Now, with Bayes' law we can calculate the probabilities of origin:

$$\begin{aligned} P(H|\cap_{k=1}^{K} T_k) &= \frac{(P(\cap_{k=1}^{K} T_k|H)P(H))}{(P(\cap_{k=1}^{K} T_k|H)P(H) + P(\cap_{k=1}^{K} T_k|G)P(G))} \\ &= \frac{\left[\prod_{k=1}^{K} P(T_k|H)\right] P(H))}{\left[\prod_{k=1}^{K} P(T_k|H)\right] P(H) + \left[\prod_{k=1}^{K} P(T_k|G)\right] P(G)} \end{aligned} \tag{4}$$

and similarly, $P(G|\cap_{k=1}^{K} T_k)$. By definition, we have $P(H|\cap_{k=1}^{K} T_k) + P(G|\cap_{k=1}^{K} T_k) = 1$.

The Bayes classifier classifies a document D as being of GPT-origin if $P(G|\cap_{k=1}^{K} T_k) > P(H|\cap_{k=1}^{K} T_k)$. Equation (4) might look complex, but it is a simple multiplication–addition of probabilities, and is straightforward to calculate from the training dataset. It is a simple calculation for a machine, but difficult for a human looking at a code snippet.

We conducted the Bayes classifier experiment with $\tau = 32$ (the best out of a number of tested τ values) and obtained the results shown in Table 6.

Table 6. Results of the Bayes classifier. Mean and standard deviation from ten runs with ten different training–test set separations (problem-wise).

	Accuracy (%)	**Precision (%)**	**Recall (%)**	**F1 (%)**
Unformatted	87.77 ± 0.56	82.99 ± 0.70	95.03 ± 0.58	88.60 ± 0.50
Formatted	86.06 ± 0.65	85.85 ± 1.23	86.46 ± 1.02	86.15 ± 0.59

It is astounding that such a simple ML model can achieve results comparable to the best human-designed feature (white-box) models (unformatted, Table 3) or the average of the GMM models (formatted, Table 4). On the other hand, the Bayes calculation is built upon a large number of features (about 1366 tokens in the training token set T, and 144 tokens on average per tested document), which makes it clear that an (untrained) human cannot perform such a calculation just by looking at the code snippets. This is notwithstanding the possibility that a trained human could somehow learn an equivalent complex pattern-matching technique that associates the presence or absence of specific tokens with the probability of origin.

6. Discussion

In this section, we explore the strengths and weaknesses of the methodologies employed in this paper, assess the practical implications of our findings, and consider how emerging technologies and approaches could further advance this field of study. Additionally, we compare our results with those of other researchers attempting to detect the origin of code.

6.1. Strengths and Weaknesses

Large code dataset: To the best of our knowledge, this study currently is the largest in terms of coding problems collected and solved by both humans and AI (see Table 1, 3.14×10^4 samples in total). The richness of the training data is probably responsible for the high accuracy and recall of around 98% that we achieved with some of the SL models.

Train–test split along problem instances: One likely important finding from our research is that we initially reached a somewhat higher accuracy (+4 percentage points for most models, not shown in the tables above) with a random-sample split. However, this split method is flawed if problems have many human solutions or many GPT solutions, as the test set may contain problems already seen in the training set. A split along problem instances, as in our final experiments, ensures that each test case comes from a so-far unseen problem, and is the choice recommended to other researchers as well. Although it has lower accuracy, it is the more realistic accuracy that can be expected on genuinely new problem instances.

Formatting: Somewhat surprisingly, after subjecting all samples to the Black code formatter tool [60], our classification models still exhibited the capacity to yield satisfactory outcomes, showing only slightly degraded performance. This shows a remarkable robustness. However, it is essential to note that the Black formatter represents only one among several code formatting tools available, each with its own distinct style and rules. Utilizing a different formatter such as AutoPEP8 [66] 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.

White-box vs. black-box features: Our experiments on unformatted code have shown that the human-designed white-box features in Table 2 can achieve a good accuracy level above 80% with most ML models. However, our exceptionally good results of 92–98% were only achieved with black-box embedding features. This is relatively independent of the ML model selected, and what is more important is the type of input features.

Feature selection: Although the human-designed features (white-box) 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 the embedding

features (black-box) indicates the existence of features with higher discriminatory power, necessitating a more profound analysis of the embedding space.

Bayes Classifier: The Bayes classifier introduced in Section 5.6.2 shows another possibility for generating a rich and interpretable feature set. The statistical properties that can be derived from the training set enable an explainable classifier that achieves an accuracy of almost 90%.

Test cases: Tested code, having undergone rigorous validation, offers a reliable and stable dataset, resulting in enhanced model 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, thereby accommodating a broader spectrum of coding styles, nuances, and potential errors and 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 in optimizing the balance between model accuracy and adaptability across various coding scenarios.

Code generators: It is important to note that only AI examples generated with OpenAI's `gpt-3.5-turbo` API were used in this experiment. This model is probably the one most frequently used by students, rendering it highly suitable for the objectives of this research paper. However, it is not the most capable of the GPT series. Consequently, the integration of additional models such as `gpt-4` [22] or `T5`$^+$ [67], which have demonstrated superior performance on coding-related tasks, can serve to not only enhance the utilization of the available data but to introduce increased variability. It is an open research question whether *one* classification model can disentangle several code generators and human code, or whether separate models are needed for each code generator. Investigating whether distinct models formulate their own unique distributions or align with a generalized AI distribution would be intriguing. 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. Given a code dataset in another programming language, the same methods shown here for Python could be used to extract features or to embed the code snippets (token-wise or as a whole) in an embedding space. We suspect that, given a similar dataset, the performance of classifiers built in such way would be comparable to the ones presented in this article.

Publicly available dataset and trained models: To support further research on more powerful models or explainability in detecting AI-generated code, we have made the preprocessed dataset publicly available in our repository https://github.com/MarcOedingen/ChatGPT-Code-Detection, accessed on 27 June 2024. The dataset can be downloaded from there via link. Moreover, we offer several trained models and a demo version of the XGB model using the TF-IDF embedding. This demonstration serves as a counterpart to public AI text detectors, allowing for the rapid online classification of code snippets.

6.2. Comparison with Other Approaches to Detect the Source of Code

Two of the works mentioned in Section 2, Hoq et al. [17] and Yang et al. [18], strive for the same goal as our paper, namely, the detection of the source of code. Here, we compare their results with ours, and summarize the other methods presented in Section 2 in Table 7.

Table 7. Summary and comparison of methods for detecting the origin of code. The accuracy values represent the values achieved by the **best** model of each work. This is in contrast to the AUC values, which are based on several models of the individual work.

Method	Key Techniques	Dataset Size	Performance	Remarks
DetectGPT4Code [18]	Zero-shot detection	102 Python and 165 Java code snippets	AUC = 0.70–0.80	Small dataset and not reliable for practical use.
Text detectors [25,26]	Vanilla NL text detectors and Fine-tuned RoBERTa-QA	Wang et al. 226k code snippets and Pan et al. 5k code snippets	Wang et al.: AUC 0.46 ± 0.19, AUC 0.86 ± 0.09 (fine-tuned); Pan et al. Acc. = 0.77 (best) and 0.53 ± 0.07 (avg.)	Fine-tuning NL detectors on code increased the performance. Standalone NL detectors are not reliable enough on code detection.
Embedding-based [17]	TF-IDF embedding and code2Vec with ASTNNs	3.162k human and 3k GPT code snippets	Accuracy = 0.97	Small number of problems with similar solutions and random splitting procedure.
Feature-based [32]	Feature grouping (lexical, structural layout, and semantic)	1.206k code snippets	Accuracy = 0.98	Small code dataset and lack of prompt engineering.
Our proposed methods	Ada, TF-IDF, and Word2Vec embedding and explainable features	15.7k human and 15.7k GPT code snippets	Accuracy = 0.98, AUC = 0.96–1.00 (TF-IDF)	Large code dataset with many problems; split problem-wise for training and testing of ML algorithms.

Hoq et al. [17] approached the problem of detecting the source of code using two ML models (SVM, XGB) and two DL models (Code2Vec, ASTNN). They found that all models delivered quite similar accuracy in the range of 90–95%. However, a drawback was that they used a dataset with only ten coding problems, for each of which they generated 300 solutions. They described this as *"Limiting the variety of code structures and syntax that ChatGPT would produce"* . Moreover, given this small number of problems, a potential flaw is that a purely random training–test split will have a high probability of each test problem also being represented in the training set (overfitting). In our approach with a larger code dataset, we carried out the training–test split in such a way that none of the test problems occurred in the training set. This somewhat tougher task may be the reason for the larger gap between simple ML models and more complex DL embedding models than was reported in [17].

Yang et al. [18] pursued the ambitious task of zero-shot classification, i.e., predicting the source of code without training. Even more ambitious, they used three different advanced LLMs as code generators (not only ChatGPT3.5, as we do). Their number of code samples for testing was 267, which is somewhat low. Nothing was said about the number of AI-generated samples. Their specific method, sketched in Section 2, is shown in [18] to be much better than text detectors applied to the code detection task. However, without specific training their TPR (= recall) of 20–60% is much lower than the recall of 98% we achieved with the best of our trained models.

7. Conclusions

This research aimed to find a classification model capable of differentiating AI- and human-written code samples. In order to enable the feasibility of such a model in application, emphasis was also placed 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.

We experimented with a variety of feature sets and a large number of ML models, including DNNs and GMMs. It turned out that the choice of the input feature set was more important than the model used. The best combinations were DNN + Ada with 97.8% accuracy and XGB + TF-IDF with 98.3% accuracy. The accuracy with human-designed low-dimensional feature sets was 10–15 percentage points lower.

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, H_1 and H_2, which postulate the distinctness between AI-generated and human-generated coding styles regardless

of formatting. The interpretability of our approach was improved by a Bayes classifier, which made it possible to highlight individual tokens and provide a more differentiated understanding of the decision-making process.

One notable outcome of this experiment is the acquisition of a substantial dataset of Python code examples created 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 can help to facilitate a comparative analysis between AI- and human-generated code while highlighting the distinctions and similarities.

Having only experimented with Python code, further research could focus on investigating the coding style of AI using other programming languages. Moreover, our 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 seeking to understand how language models write code and how the results differ from human-written code. We have plans to expand this research to include models such as GPT-4, $T5^{+}$, and Gemini as well as to analyze the transferability of their coding styles across various programming languages such as Java and C++. Future work will also focus on enhancing the explainability of our models, building on this paper's emphasis on prediction strength.

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. In order 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.

Author Contributions: Conceptualization, M.O. and M.H.; methodology, M.O., M.H., and W.K.; software, M.O., R.D., and M.H.; validation, M.O., R.C.E., R.D., and W.K.; formal analysis, M.O., R.C.E., R.D., and W.K.; investigation, M.O., R.C.E., R.D., and W.K.; resources, M.O., R.C.E., and M.H.; data curation, M.O. and M.H.; writing—original draft preparation, M.O., R.C.E., R.D., M.H., and W.K.; writing—review and editing, M.O., R.C.E., R.D., and W.K.; visualization, M.O., R.C.E., R.D., and W.K.; supervision, M.O. and W.K.; project administration, M.O. and W.K. All authors have read and agreed to the published version of the manuscript.

Funding: Author R.C.E. was supported by the research training group "Dataninja" (Trustworthy AI for Seamless Problem Solving: Next Generation Intelligence Joins Robust Data Analysis) funded by the German federal state of North Rhine-Westphalia.

Data Availability Statement: Research data is made available through the GitHub repository https://github.com/MarcOedingen/ChatGPT-Code-Detection, accessed on 27 June 2024.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AUC	Area Under the (ROC) Curve
ASTNN	Abstract Syntax Tree-based Neural Network
CART	Classification And Regression Tree
CBOW	Continuous Bag Of Words
DNN	Deep Neural Network
DT	Decision Tree
EM	Expectation Maximization
F1	F_1-score
GMM	Gaussian Mixture Model
GPT	Generative Pretrained Transformer
LLM	Large Language Model

LR	Logistic Regression
LSTM	Long Short-Term Memory
ML	Machine Learning
NL	Natural Language
OPCT	Oblique Predictive Clustering Tree
RF	Random Forest
SL	Supervised Learning
TF-IDF	Term Frequency-Inverse Document Frequency
XGB	eXtreme Gradient Boosting

References

1. Alawida, M.; Mejri, S.; Mehmood, A.; Chikhaoui, B.; Isaac Abiodun, O. A Comprehensive Study of ChatGPT: Advancements, Limitations, and Ethical Considerations in Natural Language Processing and Cybersecurity. *Information* **2023**, *14*, 462. [CrossRef] [CrossRef]
2. Ziegler, A.; Kalliamvakou, E.; Li, X.A.; Rice, A.; Rifkin, D.; Simister, S.; Sittampalam, G.; Aftandilian, E. Productivity assessment of neural code completion. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, MAPS 2022, New York, NY, USA, 13 June 2022; pp. 21–29. [CrossRef]
3. Charan, P.V.S.; Chunduri, H.; Anand, P.M.; Shukla, S.K. From Text to MITRE Techniques: Exploring the Malicious Use of Large Language Models for Generating Cyber Attack Payloads. *arXiv* **2023**, arXiv:2305.15336. [CrossRef]
4. Russell, S.; Bengio, Y.; Marcus, G.; Stone, P.; Muller, C.; Mostaque, E. Pause Giant AI Experiments: An Open Letter. 2023. Available online: https://futureoflife.org/open-letter/pause-giant-ai-experiments/ (accessed on 18 August 2023).
5. Russell, S.; Bengio, Y.; Marcus, G.; Stone, P.; Muller, C.; Mostaque, E. Policymaking in the Pause. 2023. Available online: https://futureoflife.org/wp-content/uploads/2023/04/FLI_Policymaking_In_The_Pause.pdf (accessed on 18 August 2023).
6. Weidinger, L.; Mellor, J.; Rauh, M.; Griffin, C.; Uesato, J.; Huang, P.S.; Cheng, M.; Glaese, M.; Balle, B.; Kasirzadeh, A.; et al. Ethical and social risks of harm from Language Models. *arXiv* **2021**, arXiv:2112.04359. [CrossRef]
7. Zhang, J.; Ji, X.; Zhao, Z.; Hei, X.; Choo, K.K.R. Ethical Considerations and Policy Implications for Large Language Models: Guiding Responsible Development and Deployment. *arXiv* **2023**, arXiv:2308.02678. [CrossRef]
8. Lund, B.D.; Wang, T.; Mannuru, N.R.; Nie, B.; Shimray, S.; Wang, Z. ChatGPT and a new academic reality: Artificial Intelligence-written research papers and the ethics of the large language models in scholarly publishing. *J. Assoc. Inf. Sci. Technol.* **2023**, *74*, 570–581. [CrossRef] [CrossRef]
9. Mitchell, E.; Lee, Y.; Khazatsky, A.; Manning, C.D.; Finn, C. DetectGPT: Zero-shot machine-generated text detection using probability curvature. In Proceedings of the 40th International Conference on Machine Learning, ICML'23, Honolulu, HI, USA, 23–29 July 2023.
10. Gehrmann, S.; Strobelt, H.; Rush, A. GLTR: Statistical Detection and Visualization of Generated Text. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, Florence, Italy, 28 July–2 August 2019; Costa-jussà, M.R., Alfonseca, E., Eds.; Association for Computational Linguistics Florence: Kerrville, TX, USA, 2019; pp. 111–116. [CrossRef]
11. Alamleh, H.; AlQahtani, A.A.S.; ElSaid, A. Distinguishing Human-Written and ChatGPT-Generated Text Using Machine Learning. In Proceedings of the 2023 Systems and Information Engineering Design Symposium (SIEDS), Charlottesville, VA, USA, 27–28 April 2023; pp. 154–158. [CrossRef]
12. Ghosal, S.S.; Chakraborty, S.; Geiping, J.; Huang, F.; Manocha, D.; Bedi, A.S. Towards Possibilities and Impossibilities of AI-generated Text Detection: A Survey. *arXiv* **2023**, arXiv:2310.15264. [CrossRef]
13. Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; Karri, R. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 754–768. [CrossRef]
14. Yasir, R.M.; Kabir, A. Exploring the Impact of Code Style in Identifying Good Programmers. In Proceedings of the 10th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2022), Virtual, 6 December 2023; pp. 18–24.
15. Solaiman, I.; Brundage, M.; Clark, J.; Askell, A.; Herbert-Voss, A.; Wu, J.; Radford, A.; Krueger, G.; Kim, J.W.; Kreps, S.; et al. Release Strategies and the Social Impacts of Language Models. *arXiv* **2019**, arXiv:1908.09203. [CrossRef]
16. Islam, N.; Sutradhar, D.; Noor, H.; Raya, J.T.; Maisha, M.T.; Farid, D.M. Distinguishing Human Generated Text From ChatGPT Generated Text Using Machine Learning. *arXiv* **2023**, arXiv:2306.01761. [CrossRef]
17. Hoq, M.; Shi, Y.; Leinonen, J.; Babalola, D.; Lynch, C.; Price, T.; Akram, B. Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models. In Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, SIGCSE 2024, New York, NY, USA, 20–23 March 2024; pp. 526–532. [CrossRef]
18. Yang, X.; Zhang, K.; Chen, H.; Petzold, L.; Wang, W.Y.; Cheng, W. Zero-shot detection of machine-generated codes. *arXiv* **2023**, arXiv:2310.05103. [CrossRef]
19. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* **2020**, *21*, 1–67.

20. Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, S.; Zettlemoyer, L.; Lewis, M. InCoder: A Generative Model for Code Infilling and Synthesis. In Proceedings of the Int. Conf. on Learning Repr. (ICLR), Kigali, Rwanda, 1–5 May 2023.
21. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language Models are Few-Shot Learners. In Proceedings of the Advances in Neural Information Processing Systems, Virtual, 6–12 December 2020; Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H., Eds. Curran Associates, Inc.: New York, NY, USA, 2020; Volume 33, pp. 1877–1901.
22. OpenAI.; Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; et al. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774 . [CrossRef]
23. Tian, E.; Cui, A. GPTZero: Towards Detection of AI-Generated Text Using Zero-Shot and Supervised Methods. 2023. Available online: https://gptzero.me (accessed on 27 February 2024).
24. Guo, B.; Zhang, X.; Wang, Z.; Jiang, M.; Nie, J.; Ding, Y.; Yue, J.; Wu, Y. How close is ChatGPT to human experts? Comparison corpus, evaluation, and detection. *arXiv* **2023**, arXiv:2301.07597. [CrossRef]
25. Wang, J.; Liu, S.; Xie, X.; Li, Y. Evaluating AIGC Detectors on Code Content. *arXiv* **2023**, arXiv:2304.05193. [CrossRef]
26. Pan, W.H.; Chok, M.J.; Wong, J.L.S.; Shin, Y.X.; Poon, Y.S.; Yang, Z.; Chong, C.Y.; Lo, D.; Lim, M.K. Assessing AI Detectors in Identifying AI-Generated Code: Implications for Education. *arXiv* **2024**, arXiv:2401.03676. [CrossRef]
27. Salton, G.; Buckley, C. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manag.* **1988**, *24*, 513–523. [CrossRef] [CrossRef]
28. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef] [CrossRef]
29. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 783–794. [CrossRef]
30. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of the Advances in Neural Information Processing Systems, Lake Tahoe, NV, USA, 5–8 December 2013; Burges, C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K., Eds.; Curran Associates, Inc.: New York, NY, USA, 2013; Volume 26.
31. Tang, D.; Qin, B.; Liu, T. Document Modeling with Gated Recurrent Neural Network for Sentiment Classification. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, Lisbon, Portugal, 17–21 September 2015; Màrquez, L., Callison-Burch, C., Su, J., Eds.; The Association for Computational Linguistics: Stroudsburg, PN, USA, 2015; pp. 1422–1432. [CrossRef]
32. Li, K.; Hong, S.; Fu, C.; Zhang, Y.; Liu, M. Discriminating Human-authored from ChatGPT-Generated Code Via Discernable Feature Analysis. In Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), Los Alamitos, CA, USA, 31 July 2023; pp. 120–127. [CrossRef]
33. OpenAI. Models. 2023. Available online: https://platform.openai.com/docs/models/overview (accessed on 29 July 2023).
34. Sutskever, I.; Vinyals, O.; Le, Q.V. Sequence to Sequence Learning with Neural Networks. In Proceedings of the Advances in Neural Information Processing Systems, Montreal, QC, Canada, 8–13 December 2014; Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., Weinberger, K., Eds.; Curran Associates, Inc.: New York, NY, USA, 2014; Volume 27, pp. 3104–3112.
35. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.u.; Polosukhin, I. Attention is All you Need. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R., Eds.; Curran Associates, Inc.: New York, NY, USA, 2017; Volume 30, pp. 5998–6008.
36. Jain, L.C.; Medsker, L.R. *Recurrent Neural Networks: Design and Applications*, 1st ed.; CRC Press, Inc.: Boca Raton, FL, USA, 1999.
37. Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed] [CrossRef]
38. Robertson, S. Understanding inverse document frequency: On theoretical arguments for IDF. *J. Doc.* **2004**, *60*, 503–520. [CrossRef] [CrossRef]
39. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient Estimation of Word Representations in Vector Space. *arXiv* **2013**, arXiv:1301.3781. [CrossRef]
40. Neelakantan, A.; Xu, T.; Puri, R.; Radford, A.; Han, J.M.; Tworek, J.; Yuan, Q.; Tezak, N.; Kim, J.W.; Hallacy, C.; et al. Text and Code Embeddings by Contrastive Pre-Training. *arXiv* **2022**, arXiv:2201.10005. [CrossRef]
41. Chen, T.; Kornblith, S.; Norouzi, M.; Hinton, G. A simple framework for contrastive learning of visual representations. In Proceedings of the 37th International Conference on Machine Learning (ICML), ICML'20, Virtual, 13–18 July 2020.
42. Wang, S.; Fang, H.; Khabsa, M.; Mao, H.; Ma, H. Entailment as Few-Shot Learner. *arXiv* **2021**, arXiv:2104.14690. [CrossRef]
43. Hosmer, D.W., Jr.; Lemeshow, S.; Sturdivant, R.X. *Applied Logistic Regression*; Wiley Series in Probability and Statistics; John Wiley & Sons: Hoboken, NJ, USA, 2013; Volume 398. [CrossRef]
44. Breiman, L.; Friedman, J.; Olshen, R.; Stone, C. *Classification and Regression Trees*; Chapman and Hall/CRC: Boca Raton, FL, USA, 1984. [CrossRef]
45. Stepišnik, T.; Kocev, D. Oblique predictive clustering trees. *Knowl.-Based Syst.* **2021**, *227*, 107228. [CrossRef]
46. Breiman, L. Random Forests. *Mach. Learn.* **2001**, *45*, 5–32. [CrossRef]

47. Chen, T.; Guestrin, C. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, New York, NY, USA, 13–17 August 2016; pp. 785–794. [CrossRef]
48. Friedman, J.H. Greedy Function Approximation: A Gradient Boosting Machine. *Ann. Stat.* **2001**, *29*, 1189–1232. [CrossRef]
49. Hopfield, J.J. Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci. USA* **1982**, *79*, 2554–2558. [CrossRef] [CrossRef] [PubMed]
50. Caterini, A.L.; Chang, D.E. *Deep Neural Networks in a Mathematical Framework*; SpringerBriefs in Computer Science; Springer: Cham, Switzerland, 2018. [CrossRef]
51. Carbonnelle, P. PYPL PopularitY of Programming Language. 2024. Available online: https://pypl.github.io/PYPL.html. (accessed on 13 June 2024).
52. TIOBE Software BV. TIOBE Index for June 2024. 2024. Available online: https://www.tiobe.com/tiobe-index (accessed on 13 June 2024).
53. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2405.00229. [CrossRef].
54. Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; et al. Measuring Coding Challenge Competence With APPS. In Proceedings of the Thirty-fifth Conference on Neural Information Processing Systems Track on Datasets and Benchmarks 1 (Round 2), Virtual, 6–14 December 2021.
55. CodeChef. 2023. Available online: https://www.codechef.com (accessed on 17 May 2023).
56. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Lago, A.D.; et al. Competition-level code generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. [PubMed] [CrossRef] [PubMed]
57. HackerEarth. 2023. Available online: https://www.hackerearth.com (accessed on 11 September 2023).
58. Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. Program Synthesis with Large Language Models. *arXiv* **2021**, arXiv:2108.07732. [CrossRef].
59. Trajkovski, M. MTrajK. 2023. Available online: https://github.com/MTrajK/coding-problems (accessed on 11 September 2023).
60. Black. The Uncompromising Code Formatter. 2023. Available online: https://github.com/psf/black (accessed on 29 July 2023).
61. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
62. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. *arXiv* **2015**, arXiv:1603.04467.
63. Rehurek, R.; Sojka, P. *Gensim–Python Framework for Vector Space Modelling*; NLP Centre: Faculty of Informatics, Masaryk University, Brno, Czech Republic, 2011; Volume 3.
64. Bröcker, J.; Smith, L.A. Increasing the Reliability of Reliability Diagrams. *Weather Forecast.* **2007**, *22*, 651–661. [CrossRef]
65. Austin, P.C.; Steyerberg, E.W. Graphical Assessment of Internal and External Calibration of Logistic Regression Models by using LOESS Smoothers. *Stat. Med.* **2014**, *33*, 517–535. [PubMed] [CrossRef] [PubMed]
66. Hattori, H. AutoPEP8. 2023. Available online: https://github.com/hhatto/autopep8 (accessed on 27 September 2023).
67. Wang, Y.; Le, H.; Gotmare, A.; Bui, N.; Li, J.; Hoi, S. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, Singapore, 6–10 December 2023; Bouamor, H., Pino, J., Bali, K., Eds.; Association for Computational Linguistics: Stroudsburg, PN, USA, 2023; pp. 1069–1088. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.