Article

EX-CODE: A Robust and Explainable Model to Detect AI-Generated Code

Luana Bulla , Alessandro Midolo , Misael Mongiovì and Emiliano Tramontana *

Dipartimento di Matematica e Informatica, University of Catania, 95125 Catania, Italy;
luana.bulla@phd.unict.it (L.B.); alessandro.midolo@unict.it (A.M.); misael.mongiovi@unict.it (M.M.)
* Correspondence: tramontana@dmi.unict.it; Tel.: +39-095-7383008

Abstract: Distinguishing whether some code portions were implemented by humans or generated by a tool based on artificial intelligence has become hard. However, such a classification would be important as it could point developers towards some further validation for the produced code. Additionally, it holds significant importance in security, legal contexts, and educational settings, where upholding academic integrity is of utmost importance. We present EX-CODE, a novel and explainable model that leverages the probability of the occurrence of some tokens, within a code snippet, estimated according to a language model, to distinguish human-written from AI-generated code. EX-CODE has been evaluated on a heterogeneous real-world dataset and stands out for its ability to provide human-understandable explanations of its outcomes. It achieves this by uncovering the features that for a snippet of code make it classified as human-written code (or AI-generated code).

Keywords: ChatGPT; code classification; CodeBERT; explainability; XAI

Citation: Bulla, L.; Midolo, A.; Mongiovì, M.; Tramontana, E. EX-CODE: A Robust and Explainable Model to Detect AI-Generated Code. *Information* **2024**, *15*, 819. https://doi.org/10.3390/info15120819

Academic Editor: Rodolfo Delmonte

Received: 20 November 2024
Revised: 13 December 2024
Accepted: 15 December 2024
Published: 20 December 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 last few years have witnessed a significant increase in the use of artificial intelligence (AI) to assist software developers. Large language models (LLMs) have shown to be very effective in generating code snippets, resulting in a growing interest in the integration of AI-based tools in development environments [1]. However, recent studies have unveiled some limits for AI-generated code and highlighted the need for prudence in employing AI tools in the development process [2,3], e.g., AI-generated code was shown to have more vulnerabilities than human-produced code [4].

Identifying AI-generated code snippets can help prioritise code reviews to ensure code quality. For this, an automatic solution determining the origin of code snippets becomes a priority. For assisting the detection of AI-generated code, the proposed approach has been specifically devised to use standard transformer architectures for supervised text classification. Although detectors for AI-generated text exist, such approaches exhibit low performance on code-related tasks compared to natural language processing [5–9]. Moreover, while some previous app assessed the probability that a text (or code) was AI-generated, they cannot explain the outcome, hence making it hard for the user to interpret the result and therefore diminishing the confidence of the user in the provided assessment [10–12].

Providing explanations for the outcomes of AI-based systems is a desirable property [13]. Explanations help users understand the system's model, maintain it, and use it effectively. They also assist users in debugging the model to prevent and rectify incorrect conclusions [14]. Furthermore, explainable models have been shown to increase user trust and situation awareness [15]. In contrast, many machine learning (ML) systems are black-box models that do not reveal the reason for some results, making them difficult for humans to understand and assess. This is due to a trade-off between the model's performance and its explainability [16]. Previous studies have primarily focused on improving system performance, often neglecting transparency [10,17].

In light of these considerations, we present the Explainable Code Detection (EX-CODE) system, an innovative model that leverages the inherent statistical properties of code to distinguish between human-written and AI-generated snippets. EX-CODE leverages token probabilities originated from CodeBert [18], a pre-trained language model specifically designed for code analysis, to extract meaningful features. Such features capture key characteristics of source code, including variable naming conventions, code control flow structures, and function usage patterns. The extracted features are then aggregated and used to train a logistic regression model for classifying code snippets either as human-written or AI-generated. To test our approach, we introduce the CodeMix dataset, a novel resource consisting of 518 real-world Java code snippets paired with their corresponding AI-generated versions. CodeMix aims to bridge the gap between existing datasets and real-world scenarios, fostering the development of more robust and generalisable AI code-classification models.

EX-CODE's strength lies in its ability to adhere to the principles of explainable artificial intelligence (XAI) by providing insights into its classification process. Besides classifying code snippets as AI-generated or human-written, it highlights the most influential factors for the provided classification outcome. This breakdown helps users understand the specific code constructs that impact the model's reasoning, driving trust and user adoption.

The main contribution of this paper can be summarised as follows.

- We present EX-CODE, a novel model that leverages inherent statistical code properties for accurately distinguishing between human-written and AI-generated code snippets.
- EX-CODE adheres to XAI principles, providing insights into its classification process. It can categorise code snippets and underline the most influential factors for each provided classification result, fostering trust and user adoption.
- To advance AI-generated code detection research, we introduce the CodeMix dataset, which consists of 518 pairs of real-world Java code snippets and corresponding AI-generated versions. CodeMix aims to address existing resource limitations by incorporating real-world scenarios and advancing research into robust and generalisable AI-generated code detection models.

The paper is organised as follows. Section 2 provides an overview of the current state-of-the-art methods for AI-generated code detection, highlighting the advantages and limitations of existing approaches. Section 3 details our EX-CODE model, explaining its components and functionalities. Section 4 introduces the novel CodeMix dataset specifically designed for AI-generated code detection research. Section 5 presents the experimental setting and EX-CODE model results, and shows the assessment of the impact of each feature and the explainability of the model. Section 6 discusses key findings from the previous sections, exploring the significance of the model's outcomes and providing useful insights. Section 7 draws our conclusions.

2. Related Works

AI-generated content has become increasingly popular arousing widespread interest across several domains in the scientific community. Notably, AI-generated content detection has been tackled by numerous researchers. However, detecting AI-generated code remains an ongoing challenge with relatively little work in this area. This task is considered more challenging than the task of detecting AI-generated text due to the inherent complexity of code syntax. Unlike a natural language, which can exhibit a degree of variability while maintaining coherence, code must adhere strictly to syntactic rules [19].

One such effort, GPTSniffer [10], introduced an ML-based approach to distinguish whether a portion of code was generated by ChatGPT. The approach consisted of the fine-tuning of CodeBERT [18], a BERT model pre-trained on code snippets, for this specific task, by using a hybrid dataset of AI and human-generated code. Although the results indicated high accuracy on code snippets from the same training source, the model exhibited diminished accuracy when confronted with code from other sources.

This highlights that the model tends to overfit the training set, showing limited generalisability. Furthermore, the lack of transparency in the classification process, as an outcome is made solely by CodeBERT without elucidating contributing factors, underscores the need for improved explainability. Conversely, our approach aims at expressing the reasons for the given outcome, hence highlighting the code portion that most significantly influences the outcome.

DetectCodeGPT [17] is a zero-shot method to detect AI-generated code. Such a model used stylistic tokens, such as white spaces or newline characters, automatically inserted into the source code, to distinguish between human and AI patterns. The analysis of such patterns allowed the model to effectively reveal whether a code fragment was AI-generated. This model showed higher accuracy than GPTSniffer; however, it can take as the input a maximum length of 512 tokens per analysed code, significantly reducing its applicability on different domains. Additionally, the outcomes lack explanations, providing users with a black-box decision process. Similarly, DetectGPT4Code [20] explores a training-free method for identifying code generated by large language models, such as GPT-4 and GPT-3.5. Traditional text detection methods fail on code because of its unique statistical patterns. Although these approaches emphasize performance improvements in distinguishing between human-written code and AI-generated code, they lack mechanisms for providing insight into the basis of their detections. In contrast, our method offers a comprehensive report that highlights the specific code segments most influential in the detection process, thereby enhancing explainability and transparency.

Other strategies used watermarking techniques: the embedding of unique markers into the code [19,21,22]. By detecting these markers, machine-generated code can be identified; however, marker embedding requires changing the generation model, which is not always possible. Therefore, such approaches are mostly useful to address concerns related to code licensing and plagiarism. Additionally, watermarking may not be compatible with all programming languages, potentially causing syntax errors. The effectiveness of detection is further compromised if markers can be easily altered or removed during code optimisation or refactoring. Lastly, watermarking can introduce performance overhead, which may be undesirable in production environments, leading to a preference for alternative detection strategies.

A straightforward application of machine-generated text detection to code was investigated by Pan et al. [9], who conducted an empirical study to evaluate the performance of text AI-detectors on source code snippets. The observed disparities between natural language and programming language led to a diminished detection rate, highlighting the necessity for specialised models tailored to detect machine-generated code.

One of the leading text AI-detectors is GPTZero [5], a tool that checks whether a document was written by an LLM. Similarly, some approaches use perturbation techniques with a zero-shot method [6,7,23] to alter the text by randomly masking some of its portions and then recover it using a different LLM. Nevertheless, the perturbation procedure is both time- and resource-expensive.

Several training-based methods were proposed [8,24,25] to detect machine-generated text by accurately training models to this specific task, showing good performances in in-distribution scenarios. Nonetheless, these approaches often suffer from limited generalisation capabilities, requiring access to the training data of the target model, which are not always available.

In [26], the authors proposed the Distribution-Aligned LLMs Detection framework to improve the detection of AI-generated text in black-box settings, where internal model details are unavailable. Yang et al. [27] introduced a training-free method for identifying GPT-generated text. The approach, called Divergent N-Gram Analysis (DNA-GPT), takes advantage of the differences in how humans and machines produce text.

In general, the approaches for natural language text discussed above show poor performance in machine-generated code detection [9,10]. This may stem from the fact that these models were trained on natural language texts, which are significantly different from programming language data, which follow strict rules and conventions. Consequently, these models may struggle to adapt to the distinct linguistic patterns inherent in code. Wang et al. [28] endeavoured to address this limitation by fine-tuning some of these models within specific code domains, achieving a significant improvement in performance. However, their findings also revealed a propensity for low generalisation when tested on data outside their training distribution, a challenge shared by GPTSniffer.

3. Proposed Approach

Our approach relies on the ability of LLMs to estimate the likelihood of a token occurrence based on contextual information from a specific training corpus. Generative models leverage this ability to generate content token by token, sampling each one according to the model's probability estimation, conditioned to the previous sequence of tokens. Drawing from this, we estimate the probability of a fragment of text by aggregating the probabilities assigned to its individual tokens. Hence, a model trained on human-written code would tend to assign higher probabilities to tokens corresponding to code implemented by a human compared to an AI-generated code. In contrast, an AI-model generating code follows a training process that includes a careful selection of sources and subsequent tuning steps, e.g., Reinforcement Learning from Human Feedback (RLHF) [29]. As a result, the probabilities of a sequence of tokens estimated by a model trained on human-written code tend to diverge when applied to AI-generated code.

Figure 1 shows the overall architecture of our classifier. It takes as input a code snippet and returns the probability that this snippet was created by AI or human. The three main parts are (1) probability estimation, (2) feature extraction, and (3) logistic regression. We discuss the first part in Section 3.1, outlining a strategy for using CodeBERT [18], a BERT-based model fine-tuned for code, to estimate the probability of each token in the analysed code snippet. Section 3.2 reports the second part of our architecture, where probabilities computed in the first part are separated into four groups, representing different aspects of code structure and composition. The last part of our architecture, detailed in Section 3.3, employs logistic regression on the previously extracted features to perform a binary classification task (AI vs. human). Logistic regression models the probability distribution of the output classes by applying a logistic function to a linear combination of the input features. The outcome of logistic regression is inherently explainable since its dependence on the input features is naturally described by the sign and magnitude of the learned coefficients.

3.1. Probability Estimation

Language models estimate the probability of a sequence of tokens by multiplying the conditional probabilities of each token given the previous ones in the sequence. This is a key aspect of how transformer decoders work (e.g., GPT [30]), which are based on unidirectional self-attention. However, popular medium-size language models that are capable of handling code are based on bidirectional encoder architectures, such as BERT [31]. We exclude more sophisticated and opaque LLMs (e.g., LLama2, Mistral and GPT-3.5) [32–34] since we do not have control over their training methodology [35], and likely their generation process diverges from the human one due to sophisticated input data selection strategies and other fine-tuning steps, e.g., RLHF. As a result, such models cannot be directly used for the task of estimating the probability of token sequences, as they are not designed for unidirectional, autoregressive generation.

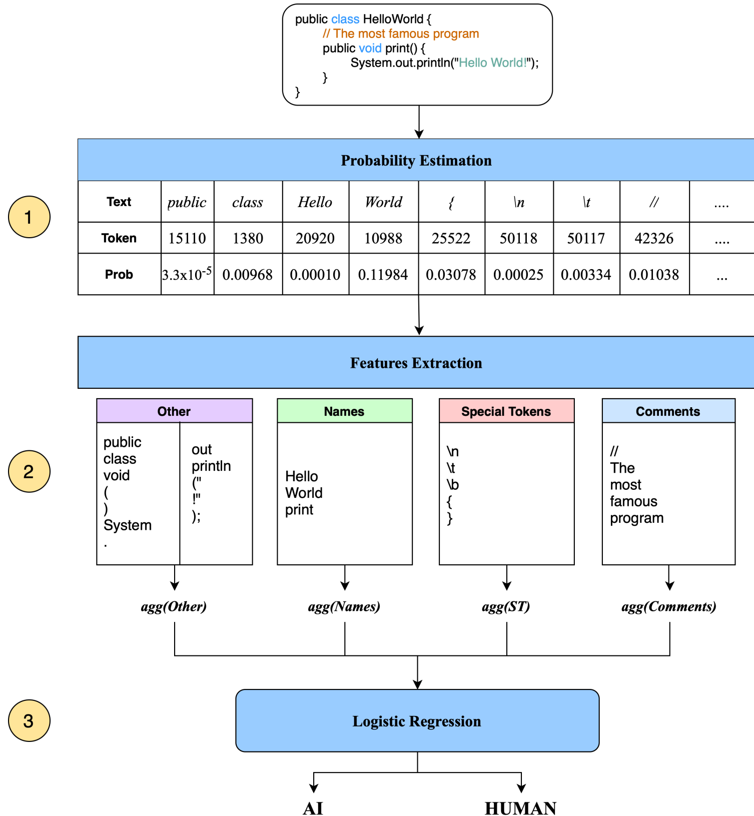

Figure 1. The proposed classifier, when a code fragment is given as input, follows a three-stage process: (1) token probability estimation, (2) feature extraction, and (3) logistic regression. Then, it determines whether the code was human-written or AI-generated.

We adopted CodeBERT, a BERT-based model [31], pre-trained on the CodeSearchNet dataset [36], that encompasses both natural language and programming languages. More specifically, we employed the Masked Language Modeling (MLM) technique and trained CodeBERT (https://huggingface.co/microsoft/codebert-base-mlm, accessed on 4 November 2024) using an MLM objective where certain tokens in the input sequence were substituted by gaps (special "<mask>" tokens), and the goal was to determine the original tokens based on the surrounding context. We employed this model for token probability estimation by adopting a unidirectional approach that replaces one token at a time by <mask> and considered the probability distribution estimated on the corresponding token [37]. Our probability computation was performed sequentially. We processed a token of the input code snippet at a time and extracted the probability returned by the model for that token, indicating the likelihood of the specified token being in that exact position considering the context of the code.

Algorithm 1 outlines all the steps involved in the generative unidirectional approach. A snippet of code is the input of the procedure. The first step is to apply the tokeniser to the code and initialise the sequence list that will be passed to the model and the probs list that will contain all the probabilities extracted from each token. Next, for each token derived from the tokeniser applied to the input code, the following procedure is executed: firstly, the size of the sequence is checked since CodeBERT takes as input a maximum number of 512 tokens. When the sequence exceeds this limit, we shift the whole sequence to the left to remove older tokens and make room for new tokens. Secondly, the <mask> token is added to the sequence, which is then passed as input to the run_model function. This function executes the model and returns the probability of the actual token t passed as a second parameter, which is added to the probs list. Finally, the <mask> token in the sequence is replaced with the actual token to prepare the sequence for the next step.

Algorithm 1 Generative unidirectional probability estimation

Require: $code \neq null$
  $tokens \leftarrow tokenizer(code)$
  $sequence \leftarrow [\,]$
  $probs \leftarrow [\,]$
  for t in $tokens$ **do**
    if $sequence.size \geq 512$ **then**
      $sequence \leftarrow shift_to_left(sequence)$
    end if
    $prob \leftarrow run_model(sequence + \text{<mask>}, t)$
    $probs.append(prob)$
    $sequence.append(t)$
  end for

3.2. Feature Extraction

The source code inherently carries a wealth of information aimed at controlling the execution of specific tasks, while also aiding humans comprehend its behaviour and structure. This information encompasses all the parts that constitute source code, including indentation [38], naming conventions [39], comments [40] and instructions [41]. Such parts, while making the source code unique, also highlight the style and the experience of its developer.

To systematically characterise source code based on its structure and content, we group tokens into four distinct sets. We define $\mathcal{N}$ as the set of Names, Σ as the set of Special Tokens, Γ as the set of Comments, and P as the set of all remaining tokens (Others), each having the following meaning.

- $\mathcal{N}$ contains all the tokens that are identifiers used in the input code for classes, methods and variables. Such names were extracted beforehand using a parser, which was Javaparser [42] in our case, as it provides support to inspect the source code and collect relevant data.
- Σ represents all tokens associated with indentation, including spaces, new lines, tabs, and curly brackets. Such data can increase the comprehension of the program and define the programming style of the code.
- Γ contains all the tokens that are comments found in the source code, both single-line and multi-line ones, e.g., we have selected all the strings located after the "//" symbol or enclosed within "/* */" symbols, since the analysed code is in Java programming language.
- P collects all the remaining tokens inside the source code, such as Java language keywords (i.e., if, for, while, etc.), numbers, strings, the names of types provided in the standard Java library, round and square brackets, and punctuation.

Once the four sets of tokens in the source code were passed to the model to determine the probability for each token in the set, an aggregation function was applied to each set to consolidate all the probabilities for the tokens belonging to the same set into a single feature.

We have used three different aggregation functions: *sum*, *avg*, *scaled_sum*, defined as follows.

$$sum(S) = \sum_{t \in S} p(t) \tag{1}$$

$$avg(S) = \frac{\sum_{t \in S} p(t)}{n} \tag{2}$$

$$scaled_sum(S) = \frac{\sum_{t \in S} p(t)}{\sqrt{n}} \tag{3}$$

where $p(t)$ is the probability associated with token t, and S is one among the previously defined set of tokens $\mathcal{N}$, Σ, Γ or P.

Each function above gives a score. The obtained scores represent three single features of the source code, and they are applied to each of the four token sets. This results in a total of twelve scores, each representing a different facet of the code's composition and semantics in relation to the CodeBERT model. These scores collectively identify the code as unique based on its characteristics.

3.3. Logistic Regression

The classification task was performed through logistic regression (LR), a statistical method commonly used for binary classification. LR leverages a set of predictor variables to predict the presence or absence of a particular outcome or characteristic [43]. Unlike linear regression, it is more suitable for models where the dependent variable is dichotomous [44]. The logistic function was defined as:

$$P(Y=1|X) = \frac{1}{1+e^{-(\beta_0+\beta_1 X_1+\beta_2 X_2+\ldots+\beta_n X_n)}}$$

where $P(Y=1|X)$ is the probability of the dependent variable being labelled as 1 given the values of the independent variables; thus, $\beta_0, \beta_1, \ldots, \beta_n$ are the coefficients of the model and $X_1, X_2, \ldots, X_n$ are the values of the independent variables.

In our model, the dependent variable distinguishes between "human" and "AI", indicating which category the classified code snippet most likely belongs to, where 1 is AI and 0 is human. The independent variables consist of the twelve features extracted by the previous step (Section 3.2).

An empty Comment (Γ) set, which would result when the analysed code has no comments, could significantly impact analyses, leading to biased estimates of the investigated associations [45]. To address this issue, we have replaced all missing values with the average value of the respective class, e.g., a human-written code without comments would have the average value of comments features assigned. Assigning the average value allows us to mitigate the impact of missing values while maintaining balanced values of the features for the code with comments. Since this approach standardises the feature according to the same class distribution, we add a binary feature called "Comment", which is set to 0 if the source code does not present any comments, and 1 otherwise. This prevents the risk that the LR treats the code snippets without comments in the same manner as those with comments.

The coefficients β were estimated using maximum likelihood estimation, typically by minimising the negative log-likelihood function. Once the coefficients were estimated, the model could be used to predict the probability of the outcome for new data. These coefficients represent the weight that each independent variable, i.e., the feature, contributes to the final prediction. Analysing these coefficients provides insights into which features have a major impact on the classification, e.g., higher coefficients related to comments highlight substantial differences between human and AI-generated code. Therefore, the analysis of these coefficients offers a deeper understanding of the reasons behind each outcome, showing the distinctions between the two classes.

We train the LR algorithm with the liblinear solver, the L2 regularisation, 10,000 maximum number of iterations to facilitate convergence, the inverse of regularisation strength ('C' value) equals to 1.0, the tolerance for stopping criteria (tol) equals to 1×10^{-4} (default value) and the class_weight set to None, meaning that all classes are weighted equally.

3.4. Model Variants

To better convey the effectiveness of our EX-CODE approach, we explored the impact of different classification strategies on performance. We examined three variations implemented in our framework: unidirectional (PAD), unidirectional (MASK), and bidirectional strategies. The unidirectional (PAD) strategy progressively masks tokens starting from the beginning and moving toward the end of the code (see Section 3.1), and remove (padding) downstream tokens. This sequential masking simulates the generation performed by many

AI models, where tokens are produced one by one [37] based on previous content. By replacing the current token with <mask> and removing downstream tokens, the model is forced to infer the probability distribution for the <mask> token from the upstream context. The remaining tokens of the input are filled with the special <pad> token, a standard technique called padding which is interpreted by the encoder as if the <mask> token is the last of the sequence.

The unidirectional (MASK) strategy is similar to the previous one; however, it fills the remaining tokens with other <mask> tokens. This multi-masking approach compels the model to consider that the sequence has remaining tokens and to grasp the overall context length which might help for an accurate prediction.

The bidirectional strategy leverages the strengths of BERT's bidirectional pre-training. It employs the close task concept, intermittently masking tokens and providing the entire context to the model [31]. This approach relies on BERT's ability to understand the relationships between words regardless of their order. As the model predicts each masked token, it gains additional information from the surrounding context, enhancing its accuracy in estimating the probability distribution of single tokens.

4. CodeMix Dataset

We have assembled a new dataset consisting of code snippets gathered from GitHub Gists (https://gist.github.com/discover, accessed on 4 November 2024). Within GitHub, a Gist allows users to share snippets of code quickly and easily, without the need to create a full repository. Gists are often used for different purposes, e.g., education and learning, software development and open-source contribution, among others. We chose these for two main reasons: firstly, the platform is used by both novices and experienced developers. This guarantees a wider spectrum of programming styles, topics covered, and fields of applications. The presence of a heterogeneous dataset helps our model to better generalise with code from different domains, thereby reducing the risk of overfitting the training data. Secondly, since CodeBert was trained on the CodeSearchNet dataset [36], we ensured that the code used in our experiments had not been used for training CodeBert, making the experimental evaluation reliable.

We have collected 518 code snippets written in Java. To generate the respective AI versions, we have queried ChatGPT with two different sessions. In the first session, we set the context of the model by providing the following prompt: "You are a software engineer skilled in describing code functionality without using class, method, and variable names. Please provide a description of the following code's behaviour in simple terms". For each code snippet, we then asked: "Can you please describe what this code does without using class, method, and variable names?". The output of ChatGPT is a brief description of the code's functionalities. Since names are used as features in our model, we instruct ChatGPT to omit them to preserve the generality of the descriptions. In the second session, we set the ChatGPT's context with the following prompt: "You are a software engineer skilled in writing Java source code from functionalities' description". For each description of a snippet of code, we then asked "Can you please generate a Java code according to this description?". ChatGPT returned a code snippet implementing the functionalities described in the text passed as input.

By employing two distinct sessions, we ensure that ChatGPT referred to the code passed as input for the description, and then it separately had just the description available to generate the corresponding code. As a result, we obtained a balanced dataset consisting of 1036 snippets of code, evenly split between those generated by an AI and those written by a human. This dataset forms the basis of our experiments.

5. Evaluation

We evaluated the effectiveness of the EX-CODE model by testing it both on the CodeMix dataset (see Section 4) and on Nguyen et al. dataset [10], which included 1484 code snippets, divided into human-written (i.e., 738) and machine-generated (i.e., 746) code snippets.

The Nguyen et al. dataset is structured into unpaired and paired snippet categories, delineated by two distinct data collection methodologies. The unpaired category includes code generated by ChatGPT, prompted with queries spanning a broad spectrum of programming tasks (i.e., 137 snippets), alongside human-written snippets sourced from GitHub Gist (i.e., 137 snippets). These snippets have no direct correlation, mirroring real-world scenarios. In contrast, paired snippets exhibit a coupled structure, juxtaposing human-written (i.e., 601 snippets) and AI-generated code (i.e., 609 snippets). Here, each code snippet sourced from a Java programming book's solution repository was associated with code generated by ChatGPT, which leverages the task descriptions provided in the book and aligns them with human-authored scripts. The real-world examples for the dataset in [10] represent just 18% of the total number of snippets, while the remaining 82% are the snippets from the Java programming book. This significant disproportion underscores the predominantly quantity of code which is human-written. Conversely, the CodeMix dataset contains 1036 snippets from Github gists, which introduce greater heterogeneity in the data. This variety brings CodeMix data closer to the distribution of real-world data.

Our evaluation strategy takes a multi-faceted approach, considering both overall model performance and the effectiveness of different methodological approaches. We also conduct an in-depth analysis to enhance the model's interpretability and reliability. Specifically, Section 5.1 evaluates the EX-CODE model's ability to discriminate between machine-generated and human-written code. Section 5.2 shows the performance of different methodological strategies (crf. Section 3.4) to identify the most effective approach for the EX-CODE model. Section 5.3 presents an in-depth analysis to unveil the key factors influencing EX-CODE's classification decisions. This enhances the model's interpretability and reliability, aligning with the principles of XAI [14].

5.1. Overall Performance

To evaluate the performance of the EX-CODE approach in distinguishing between human-written and AI-generated code, we assessed the model both on the Nguyen et al. [10] and CodeMix datasets.

Table 1 presents the overall performance of EX-CODE using the Nguyen et al. dataset, in terms of precision, recall, and F1-score. EX-CODE is effective at distinguishing between AI-generated and human-written code, with balanced precision, recall, and F1-scores across both categories. There is a slightly higher precision for AI-generated code (0.83) compared to human-written code (0.76), and a higher recall for human-written code (0.86) compared to AI-generated code (0.72). Finally, the macro and weighted averages being identical (0.80 for precision, 0.79 for recall and F1-score) depends on the fact that the two classes (AI and human) are well balanced in the dataset.

Table 1. EX-CODE performance in discriminating between AI-generated and human-written code. "Macro average" represents the arithmetic average, while in "Weighted average", the values are weighted by support. The model was trained and tested on the dataset in [10].

Label	Precision	Recall	F1-Score	Support
AI	0.83	0.72	0.77	138
Human	0.76	0.86	0.81	146
Macro average	0.80	0.79	0.79	284
Weighted average	0.80	0.79	0.79	284

Table 2 presents the overall performance of EX-CODE when using the CodeMix dataset in terms of precision, recall, and F1-score. EX-CODE shows a strong ability to identify AI-generated code, with a high recall (0.87), suggesting that the model is effective at detecting AI-generated samples. There is a higher precision for human-written code (0.82) compared to AI-generated code (0.75). Conversely, the higher recall for AI-generated code (0.87)

compared to human-written code (0.67) suggests that the model has a slight bias towards predicting AI-generated code.

Table 2. EX-CODE performance in distinguishing AI-generated from human-written code. The model was trained and tested on our CodeMix dataset.

Label	Precision	Recall	F1-Score	Support
AI	0.75	0.87	0.81	111
Human	0.82	0.67	0.74	97
Macro average	0.79	0.77	0.77	208
Weighted average	0.78	0.79	0.77	208

In summary, the model is effective on both datasets, with balanced precision, recall, and F1-scores. The macro and weighted averages are consistent, indicating stable performance. These results highlight the models' robustness and effectiveness in different contexts, while also pointing to areas for potential improvement, particularly in balancing precision and recall for both AI-generated and human-written code.

5.2. Probability Estimation Strategies

To assess the influence of the different probability estimation approaches detailed in Section 3.4, we trained three distinct variants of EX-CODE, each incorporating a different strategy: unidirectional (PAD), unidirectional (MASK), and bidirectional. The unidirectional (PAD) approach replaces one token at a time with <mask> and removes downstream tokens, reproducing the generative models' step-by-step text generation process. In addition, the unidirectional (MASK) strategy sets <mask> as the downstream tokens. The bidirectional approach, leveraging BERT's bidirectional nature, replaces the <mask> one token at a time and leaves all surrounding code unchanged.

Table 3 presents a comprehensive evaluation of these strategies in terms of weighted precision, recall and F1-score. We trained and tested each variant on the Nguyen et al. dataset to provide a thorough analysis of their performance. The unidirectional (PAD) model demonstrates the highest precision, recall, and F1-score among the three variants. This suggests that masking tokens individually and reproducing the generative models' step-by-step text generation process is highly effective for identifying AI-generated code. The high precision indicates that the model is effective at correctly identifying AI-generated code without many false positives. The high recall shows that it can identify most of the AI-generated code instances. The unidirectional (MASK) model performs the worst among the three variants. This approach, which hides multiple tokens simultaneously but reveals them sequentially, seems to struggle with accurately identifying AI-generated code. The poor performance may be due to the model's difficulty in predicting all downstream tokens, which could adversely affect the estimation of the first <mask> token probability distribution. The bidirectional model, leveraging BERT's bidirectional nature, performs better than the unidirectional (MASK) model but not as well as the unidirectional (PAD) model. This approach, which replaces with <mask> only one token at a time while leaving the code context unmasked, shows a balanced performance. The precision and recall are relatively high, indicating that this model can effectively identify AI-generated code with fewer errors compared to the unidirectional (MASK) model.

Table 3. Evaluation of the EX-CODE probability estimation strategies in terms of weighted precision, recall, and F1-score. We train and test each strategy on the dataset in [10].

Model	Precision	Recall	F1-Score
Unidirectional (PAD)	0.80	0.79	0.79
Unidirectional (MASK)	0.64	0.61	0.58
Bidirectional	0.73	0.70	0.69

In summary, the unidirectional (PAD) approach appears to be the most effective strategy for this task, likely because it closely mimics the generative process of AI-generated code, allowing the model to better understand and predict the patterns. The unidirectional (MASK) approach may be less effective due to the complexity introduced by masking multiple tokens simultaneously, which might confuse the model and lead to less accurate predictions. The bidirectional approach benefits from BERT's ability to consider the context from both directions, making it a stronger performer, though not as effective as the unidirectional (PAD) approach in this specific task. These observations suggest that for tasks involving the identification of AI-generated code, methods that closely replicate the generative process (like unidirectional (PAD)) may offer superior performance. Based on this evaluation, we select the unidirectional (PAD) strategy as a primary choice for our EX-CODE approach.

5.3. Features Analysis

This section describes and evaluates the explainable aspect of the proposed EX-CODE model. As previously discussed in Section 3.2, the model extracts four different features from the code: (i) comments, single and multi-line; (ii) names of classes, methods and variables; (iii) Special Tokens, as indentation elements (i.e., spaces, newlines, tabs, curly brackets); (iv) other, instruction identifiers, numbers, strings, types, brackets, punctuation. Each feature characterises the code, which may exhibit different traits depending on whether it was generated by an AI model or written by a human. By analysing these features, the model distinguishes between human-written and AI-generated code.

The logistic regression step (see Section 3.3) assigns a coefficient β_f to each feature f, which represents the contribution that the feature has to the final prediction. Given a prediction, we are able to define the score of each feature (f) as:

$$score(f) = (\beta_f \times value_f) + (\beta_0/12) \tag{4}$$

where β_f represents the weight assigned by the logistic regressor to the feature, $value_f$ is the actual value of the aggregated feature (see Section 3.2) and β_0 is the intercept computed by the logistic regressor. We distributed the intercept across our 12 features so that the regressor falls on the decision boundary when all scores are zero.

We computed twelve scores, one for each aggregated feature (i.e., *sum*, *average*, and *scaled_sum* for each of the four features), which were then combined to yield four final scores corresponding to 'Other', 'Special Tokens', 'Names', and 'Comments'. Finally, the relative contribution of each score was computed as a percentage of the total. For every prediction made by the model, we generated a plot showing the distribution of each feature relative to the final prediction, distinguishing contributions in favour of AI-generated from contributions in favour of human-written. Positive values indicate features associated with human traits, while negative values indicate features associated with AI traits. Additionally, the plot displays the model's probability for both AI and human-written code.

Figure 2 illustrates the plot for the code provided in Listing 1. This AI-generated code was correctly classified as such, with a probability of 81.45%. The key features influencing this classification were 'Names' and 'Comments', both of which have notable negative (towards AI-generated) contributions to the overall prediction, with scores of -0.32 and -0.35, respectively. Upon closer inspection of Listing 1, we see that the only comment in the code is "Getters and setters". This comment was identified as AI-generated, which is consistent with how LLMs typically produce comments. In this case, the comment seems to serve only as an introduction to the functions, without providing any meaningful detail. Interestingly, while the 'Names' and 'Comments' features strongly indicate AI characteristics, the 'Special Tokens' feature lean toward more human-like traits. This is likely due to how the AI handles spaces and indentation. The patterns of spacing, tabbing, and overall formatting in this case resemble code written by a human developer, suggesting that although the content reflects AI generation, the structural formatting mimics human coding practices.

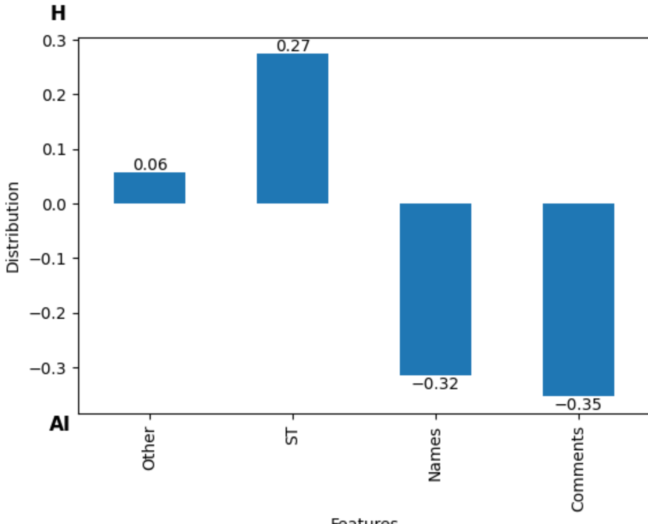

Figure 2. Contribution of each feature for the prediction of an AI-generated code (where ST represents Special Tokens). The analysed file is shown in Listing 1. The estimated probability that the code is AI-generated is 81.45%, whereas the probability that the code is written by human is 18.55%.

Listing 1. An extract of the AI-generated code that the model has correctly classified as AI.

```
public PersonData(int id, String name, String email) {
    this.id = id;
    this.name = name;
    this.email = email;
}

// Getters and setters
public int getId() {
    return id;
}
```

Figure 3 shows the plot for the code in Listing 2, which was written by a human. The model correctly classified the code as human-written, with a high probability (96.39%). The most influential feature in this prediction is the 'Special Tokens' (ST) feature, with a positive contribution of 0.66, indicating that the patterns of spaces, newlines, and other formatting elements are characteristic of human-written code. The only feature with a negative score is 'Names', though its contribution is minor, having a value of −0.13. Upon examining the code in Listing 2, the formatting and indentation reflect a human coding style, as human-written code tends to be more compact, with consistent use of indentation and less reliance on excessive whitespace between lines. Instead, AI models often insert more whitespace to make the code visually cleaner, following a more rigid, template-like structure. Human developers, however, frequently follow conventions that are shaped by practical experience and efficiency, leading to code that may appear more condensed. Additionally, the use of parentheses and indentation within the code suggests a more organic and intuitive structure, which is typical of human-written code. This differs from AI-generated code, where indentation is often applied in a more formulaic manner. Another key difference is the nature of the comments. In human-written code, comments are often more nuanced and context-dependent. As observed in Listing 2, the comments require a deeper understanding of the code's purpose and context to fully grasp its meaning. In contrast, AI-generated comments are usually more straightforward and functional, often describing the code at a surface level without providing deeper insights. Instead, human developers often write comments that assume a certain level of familiarity with the

surrounding code or the broader project, making them less explicit but more integrated into the development process.

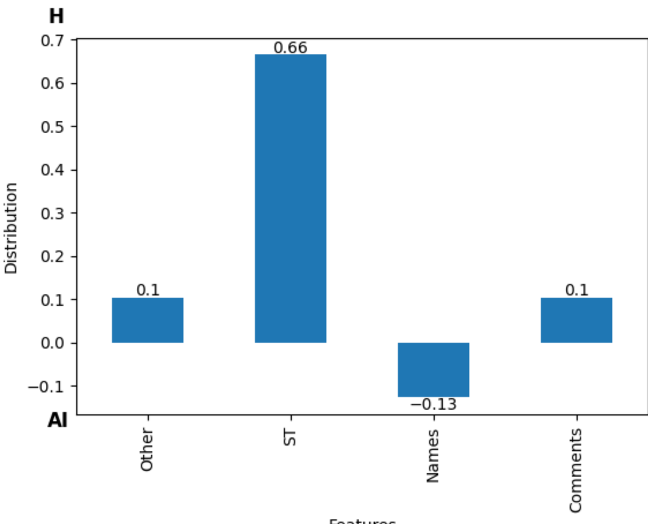

Figure 3. The weight of each feature for the prediction of human-written code (where ST represents Special Tokens). The analysed file is listed in Listing 2. The probability that the code is written by human is 96.39%, whereas the probability that the code is generated by an AI model is 3.61%.

Listing 2. An extract of the human-written code that the model has correctly classified as human.

```
static void dangerous(List<String>... stringLists) {
    List<Integer> intList = List.of(42);
    Object[] objects = stringLists;
    objects[0] = intList; // Heap pollution occurs
    String s = stringLists[0].get(0); // ClassCastException
}
```

6. Discussion

EX-CODE interpretability goes beyond simple feature analysis, extending to identifying key factors that drive predictions during inference. The model categorises code elements, highlights the most significant ones, and transparently attributes them to AI or human origin. This process enhances explainability, enabling users to gain a deeper grasp of the model's rationale behind its predictions.

In our analysis, we scrutinise the factors that influence EX-CODE's decision-making, focusing on four primary categories: Comments, Names, Special Tokens, and Others. Among these, comments stand out as the most telling feature for distinguishing human-written from AI-generated code. This distinction points to a notable divergence in how comments are handled by humans versus AI systems. Through qualitative analysis, we found that human-written comments tend to be more comprehensive, providing structured, detailed explanations in natural language. These comments often serve as guides, explaining the purpose of the code, assumptions made, and other critical details. In contrast, AI-generated comments frequently serve as placeholders. They often provide only surface-level descriptions, summarising what the code does without offering deeper insights into the logic or design choices. The result is that AI-written code, while functional, may lack the contextual richness found in human-authored comments.

Additionally, the way the model interprets naming conventions reveals interesting patterns. AI systems generally prefer simple, unambiguous names that prioritise clarity, avoiding any creative or domain-specific nuances that human developers might employ. Humans, by contrast, often choose more personalised or contextual names that reflect

their understanding of the problem domain. This highlights a subtle difference: while AI seeks clarity and directness, human developers balance functional clarity with creativity, potentially drawing on domain knowledge or personal coding habits.

EX-CODE's interpretability not only reveals the key features behind its predictions but also highlights broader patterns in the nature of AI versus human-written code, particularly in the use of comments and naming conventions. This depth of analysis fosters a deeper understanding of how different coding approaches influence model outcomes, reinforcing the explainability of EX-CODE.

Validity Treats

This work introduces a robust and explainable model to distinguish between human-written and AI-generated code. However, several considerations must be addressed to ensure the approach's effectiveness.

Firstly, the dataset labelling is inherently accurate since we know its origin beforehand. For code hosted on GitHub, there is a possibility that ChatGPT may have encountered it before. Nevertheless, this does not necessarily bias our study, as ChatGPT generates code rather than retrieving specific snippets, incorporating its unique and recognisable elements such as imports, formatting, and other code-style features. Additionally, we experimented with various prompts, both with and without prior context, to generate code that resembles what a novice programmer might produce. Despite these efforts, there may still be more sophisticated methods to circumvent the proposed approach behind EX-CODE. For example, advanced prompt engineering or the use of different AI models could potentially bypass the detection mechanisms we have in place.

Secondly, the findings of this paper may apply only to the datasets used. We diversified the data by collecting them from different sources to simulate real-world scenarios. This included code from various repositories and different types of projects to ensure a broad representation. In our evaluation, we focused on method definitions rather than entire software projects, which allowed us to isolate specific coding styles and patterns. Although CodeBERT imposes a limit of 500 tokens, this limit did not impact the size of the code snippet given as input, as we focused on methods, and additionally, we employed a sliding window. With proper training, EX-CODE can be adapted to function at the project level, analysing entire codebases for AI-generated content. By employing a sliding window having a size of 500 tokens on the code, we can input significantly longer code to the model without altering our methodology. Moreover, we could employ the above approach for each method, then aggregate the scores collected and give a final result for the whole code. Finally, to avoid introducing additional variables, we focused on Java code. Other programming languages could be investigated as CodeBERT supports several of them. In this case, while the whole approach remains unchanged, a specific parser for each language would be needed.

Lastly, in this study, the LLM behind AI-generated code was ChatGPT. A direction for future research would be to test EX-CODE with code generated by other LLMs. This would involve analysing whether the findings of this study apply to code produced by different LLMs. By doing so, we could assess the generalisability and robustness of EX-CODE across various AI-generated code sources, potentially uncovering new insights and enhancing the model's effectiveness in diverse coding environments.

7. Conclusions

This work introduced EX-CODE, a robust and explainable model for detecting AI-generated code. EX-CODE leverages the probabilities that CodeBERT provides for a missing token in a given context to identify patterns that distinguish human-written from AI-generated code. Therefore, EX-CODE relies on the statistical properties of text (and programming code) to accurately classify code snippets. By adhering to the principles of explainable AI, EX-CODE provides accurate classifications for a given snippet and offers transparency in its classification goal, as it tells which feature has contributed more to the

outcome. This transparency is crucial for fostering its adoption among users in various critical domains.

Our comprehensive assessment of EX-CODE, using the novel CodeMix dataset, demonstrated its effectiveness and robustness. The insights gained from this research contribute to the broader field of AI code detection, highlighting the importance of explainability and the need for reliable datasets that reflect real-world scenarios.

Author Contributions: Conceptualization, A.M. and E.T.; methodology, L.B., A.M. and M.M.; software, A.M.; validation, L.B., A.M., M.M. and E.T.; formal analysis, A.M., M.M. and E.T.; resources, L.B. and A.M.; data curation, L.B. and A.M.; writing—original draft preparation, L.B., A.M. and M.M.; writing—review and editing, E.T.; visualization, A.M.; supervision, E.T.; project administration, E.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data reporting the results is contained within the article. The snippets of code used for the experiments are part of ongoing study and requests to access them will be evaluated by the authors.

Acknowledgments: The authors acknowledge the support of the University of Catania PIACERI Project "TEAMS".

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

References

1. Fan, A.; Gokkaya, B.; Harman, M.; Lyubarskiy, M.; Sengupta, S.; Yoo, S.; Zhang, J.M. Large Language Models for Software Engineering: Survey and Open Problems. In Proceedings of the IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), Melbourne, Australia, 14–20 May 2023; pp. 31–53. [CrossRef]
2. 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 Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 754–768. [CrossRef]
3. Barrett, C.; Boyd, B.; Bursztein, E.; Carlini, N.; Chen, B.; Choi, J.; Chowdhury, A.R.; Christodorescu, M.; Datta, A.; Feizi, S.; et al. Identifying and Mitigating the Security Risks of Generative AI. *Found. Trends Priv. Secur.* **2023**, *6*, 1–52. [CrossRef]
4. Bang, Y.; Cahyawijaya, S.; Lee, N.; Dai, W.; Su, D.; Wilie, B.; Lovenia, H.; Ji, Z.; Yu, T.; Chung, W.; et al. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. In Proceedings of the International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers), Nusa Dua, Bali, Indonesia, 1–4 November 2023; Association for Computational Linguistics: Kerrville, TX, USA, 2023; pp. 675–718. [CrossRef]
5. Tian, E.; Cui, A. GPTZero: Towards detection of AI-generated text using zero-shot and supervised methods. *GPTZero* **2024** Available online: https://gptzero.me (accessed on 4 November 2024).
6. 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 International Conference on Machine Learning (ICML), Honolulu, HI, USA, 23–29 July 2023.
7. Su, J.; Zhuo, T.Y.; Wang, D.; Nakov, P. Detectllm: Leveraging log rank information for zero-shot detection of machine-generated text. *arXiv* **2023**, arXiv:2306.05540.
8. Zhan, H.; He, X.; Xu, Q.; Wu, Y.; Stenetorp, P. G3detector: General gpt-generated text detector. *arXiv* **2023**, arXiv:2305.12680.
9. 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.
10. Nguyen, P.T.; Di Rocco, J.; Di Sipio, C.; Rubei, R.; Di Ruscio, D.; Di Penta, M. GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT. *J. Syst. Softw.* **2024**, *214*, 112059. [CrossRef]
11. Saranya, A.; Subhashini, R. A systematic review of Explainable Artificial Intelligence models and applications: Recent developments and future trends. *Decis. Anal. J.* **2023**, *7*, 100230. [CrossRef]
12. Hassija, V.; Chamola, V.; Mahapatra, A.; Singal, A.; Goel, D.; Huang, K.; Scardapane, S.; Spinelli, I.; Mahmud, M.; Hussain, A. Interpreting black-box models: A review on explainable artificial intelligence. *Cogn. Comput.* **2024**, *16*, 45–74. [CrossRef]
13. Confalonieri, R.; Coba, L.; Wagner, B.; Besold, T.R. A historical perspective of explainable Artificial Intelligence. *WIREs Data Min. Knowl. Discov.* **2021**, *11*, e1391. [CrossRef]
14. Angelov, P.P.; Soares, E.A.; Jiang, R.; Arnold, N.I.; Atkinson, P.M. Explainable artificial intelligence: An analytical review. *WIREs Data Min. Knowl. Discov.* **2021**, *11*, e1424. [CrossRef]

15. Atakishiyev, S.; Salameh, M.; Yao, H.; Goebel, R. Explainable artificial intelligence for autonomous driving: A comprehensive overview and field guide for future research directions. *IEEE Access* **2024**, *12*, 101603–101625. [CrossRef]
16. Minh, D.; Wang, H.X.; Li, Y.F.; Nguyen, T.N. Explainable artificial intelligence: A comprehensive review. *Artif. Intell. Rev.* **2022**, *55*, 3503–3568. [CrossRef]
17. Shi, Y.; Zhang, H.; Wan, C.; Gu, X. Between Lines of Code: Unraveling the Distinct Patterns of Machine and Human Programmers. *arXiv* **2024**, arXiv:2401.06461.
18. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
19. Lee, T.; Hong, S.; Ahn, J.; Hong, I.; Lee, H.; Yun, S.; Shin, J.; Kim, G. Who wrote this code? watermarking for code generation. *arXiv* **2023**, arXiv:2305.15060.
20. 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]
21. Sun, Z.; Du, X.; Song, F.; Li, L. CodeMark: Imperceptible Watermarking for Code Datasets against Neural Code Completion Models. In Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), San Francisco, CA, USA, 3–9 December 2023; pp. 1561–1572. [CrossRef]
22. Li, B.; Zhang, M.; Zhang, P.; Sun, J.; Wang, X. Resilient Watermarking for LLM-Generated Codes. *arXiv* **2024**, arXiv:2402.07518.
23. Bao, G.; Zhao, Y.; Teng, Z.; Yang, L.; Zhang, Y. Fast-detectgpt: Efficient zero-shot detection of machine-generated text via conditional probability curvature. *arXiv* **2023**, arXiv:2310.05130.
24. Tian, Y.; Chen, H.; Wang, X.; Bai, Z.; Zhang, Q.; Li, R.; Xu, C.; Wang, Y. Multiscale positive-unlabeled detection of ai-generated texts. *arXiv* **2023**, arXiv:2305.18149.
25. Chen, Y.; Kang, H.; Zhai, V.; Li, L.; Singh, R.; Ramakrishnan, B. Gpt-sentinel: Distinguishing human and chatgpt generated content. *arXiv* **2023**, arXiv:2305.07969.
26. Zeng, C.; Tang, S.; Yang, X.; Chen, Y.; Sun, Y.; Li, Y.; Chen, H.; Cheng, W.; Xu, D. DALD: Improving Logits-based Detector without Logits from Black-box LLMs. *arXiv* **2024**, arXiv:2406.05232. [CrossRef]
27. Yang, X.; Cheng, W.; Wu, Y.; Petzold, L.; Wang, W.Y.; Chen, H. DNA-GPT: Divergent N-Gram Analysis for Training-Free Detection of GPT-Generated Text. *arXiv* **2023**, arXiv:2305.17359. [CrossRef]
28. Wang, J.; Liu, S.; Xie, X.; Li, Y. Evaluating AIGC detectors on code content. *arXiv* **2023**, arXiv:2304.05193.
29. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. Training language models to follow instructions with human feedback. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 27730–27744.
30. Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. Gpt-4 technical report. *arXiv* **2023**, arXiv:2303.08774. [CrossRef]
31. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
32. Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv* **2023**, arXiv:2307.09288.
33. Mistral. Frontier AI in Your Hands. Available online: https://mistral.ai (accessed on 4 November 2024).
34. OpenAI. GPT 3.5 Turbo. Available online: https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates (accessed on 4 November 2024).
35. Liesenfeld, A.; Lopez, A.; Dingemanse, M. Opening up ChatGPT: Tracking openness, transparency, and accountability in instruction-tuned text generators. In Proceedings of the International Conference on Conversational User Interfaces, Eindhoven, The Netherlands, 19–21 July 2023; pp. 1–6.
36. Husain, H.; Wu, H.H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv* **2019**, arXiv:1909.09436.
37. Gokul, Y.; Ramalingam, M.; Chemmalar, S.G.; Supriya, Y.; Gautam, S.; Praveen, K.R.M.; Deepti, R.G.; Rutvij, H.J.; Prabadevi, B.; Weizheng, W.; et al. Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions. *arXiv* **2023**, arXiv:2305.10435.
38. Bauer, J.; Siegmund, J.; Peitek, N.; Hofmeister, J.C.; Apel, S. Indentation: Simply a Matter of Style or Support for Program Comprehension? In Proceedings of the IEEE/ACM International Conference on Program Comprehension (ICPC), Montreal, QC, Canada, 25 May 2019; pp. 154–164. [CrossRef]
39. Alsuhaibani, R.S.; Newman, C.D.; Decker, M.J.; Collard, M.L.; Maletic, J.I. On the Naming of Methods: A Survey of Professional Developers. In Proceedings of the International Conference on Software Engineering (ICSE), Madrid, Spain, 22–30 May 2021; pp. 587–599. [CrossRef]
40. Steidl, D.; Hummel, B.; Juergens, E. Quality analysis of source code comments. In Proceedings of the International Conference on Program Comprehension (ICPC), San Francisco, CA, USA, 20–21 May 2013; pp. 83–92. [CrossRef]
41. Kuhn, A.; Ducasse, S.; Gîrba, T. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.* **2007**, *49*, 230–243. [CrossRef]
42. Smith, N.; Van Bruggen, D.; Tomassetti, F. *Javaparser: Visited*; Leanpub: Victoria, BC, Canada, 2017.
43. Hosmer, D.W., Jr.; Lemeshow, S.; Sturdivant, R.X. *Applied Logistic Regression*; John Wiley & Sons: Hoboken, NJ, USA, 2013.

44. Kurt, I.; Ture, M.; Kurum, A.T. Comparing performances of logistic regression, classification and regression tree, and neural networks for predicting coronary artery disease. *Expert Syst. Appl.* **2008**, *34*, 366–374. [CrossRef]
45. Donders, A.R.T.; van der Heijden, G.J.; Stijnen, T.; Moons, K.G. Review: A gentle introduction to imputation of missing values. *J. Clin. Epidemiol.* **2006**, *59*, 1087–1091. [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.