Contents

Chapter 1

Introduction

Generative Artificial Intelligence (AI) is a rapidly advancing branch of AI that, in recent years, has made tremendous progress, largely due to the widespread adoption of deep learning techniques. Generative models are now powerful deep learning architectures capable of producing highly realistic text, images, and even video content.

The famous Turing Test is frequently referenced in this context. In this test, a human judge had to determine whether they were communicating with another human or with a machine. This milestone was already surpassed in 2014, with early systems such as Eugene Goostman [1].

Today, it is widely accepted that much of the content generated by generative models is indistinguishable from human-produced material and indeed, for humans, **this distinction is increasingly difficult to make**.

The introduction of the Transformer architecture in 2017 [2] revolutionized the field and led to the development of Large Language Models (LLMs), which quickly became part of everyday life. This wave began with the public availability of GPT-3 [3] in June 2020 [3], followed by a growing ecosystem of LLMs including GPT-4 [4], PaLM [5], LLaMA [6], Claude [7], and Mistral [8].

Generative AI, however, is not limited to natural language processing. It has also enabled powerful models for image generation, such as Stable Diffusion [9], and more recently for video synthesis, with systems like Veo-3 [10].

Although the usefulness and extraordinary capabilities of these tools are undeniable, there are many scenarios in which it becomes necessary to have user-friendly tools to detect whether text, images, or videos were generated by an AI.

The present work pursues a more modest, yet equally important objective: the **detecting artificially generated source code** produced by code generation models such as OpenAI Codex [11], GPT-4 [4], and Code LLaMA [12].

While natural language detectors are well-established and widely studied in scientific literature, the detection of AI-generated code remains a far less consolidated and more recent research area. The contribution of this work is to **analyze**

the most relevant scientific advancements made in the past year, which are often fragmented and inconsistent due to the unique challenges involved in distinguishing machine-generated code, a task that is arguably more complex than natural text detection.

1.1 Historical Overview of Code Generation by LLMs

The history of code generation can be explored from several perspectives.

1.1.1 Natural Language Processing

One possible starting point possible perspective for analysing the evolution of code generation is to trace the development of Natural Language Processing (NLP), the field that studies how machines process and interact with human language. NLP comprises two major subdomains: Natural Language Understanding (NLU), which focuses on a machine's ability to interpret and "understand" human language, and Natural Language Generation (NLG), which concerns the generation of natural-sounding text.

This distinction is particularly relevant because the technologies currently used for code generation are essentially the same as those employed for natural language generation. In fact, Transformer-based models trained on source code data approach code generation in the same way they would handle natural text generation by predicting sequences of tokens in context using learned statistical patterns [2].

From this point of view, the starting point can be traced back to 1943, during World War II, with the invention of Colossus, one of the first digital electronic computers, developed to analyse encrypted communications from the German military

In parallel, the discipline of Natural Language Processing (NLP) began to take shape as early as the 1940s, culminating in the 1954 Georgetown experiment the first public demonstration of machine translation. Another milestone came in 1966 with the creation of ELIZA [13], considered the first chatbot in history, which simulated the behaviour of a psychotherapist using pattern-matching rules.

During the 1970s and 1980s, symbolic approaches to NLP became popular. These early attempts aimed to enable machines to "understand" language through manually encoded rules and logic-based systems. In the 1990s, the importance of statistical methods became evident, marking a shift from rule-based to probabilistic models for language processing.

In 2006 Google launched its now ubiquitous Google Translate service. The following decade saw the rise of voice-based assistants: Apple's Siri (2011), Microsoft's Cortana (2014), Amazon's Alexa (2014), and Google Assistant (2016).

A major breakthrough came with the introduction of distributed word representations, especially word2vec [14] and GloVe [15], published in 2013 and 2014 respectively. These methods enabled dense vector representations that captured semantic relationships between words in large corpora.

The most significant leap, however, occurred in 2017 with the publication of

the now seminal paper "Attention is All You Need" [2], which introduced the Transformer architecture. This architecture remains the dominant framework in NLP and underpins nearly all modern LLMs. Transformers enabled major advances in tasks such as text generation, machine translation, question answering, text summarization and, more recently, code generation.

The first LLMs capable of code generation began to appear around 2019–2020, notably with the release of GPT-2 [16]. In 2021, OpenAI released Codex [11], a GPT-3 [3] derivative trained specifically for code generation and explanation, which was later integrated into GitHub Copilot.

Between 2022 and 2024, a wave of new LLMs for code generation was released, including CodeT5 [17], CodeGen [18], CodeGeeX [19], and Code LLaMA [12].

1.1.2 Code Generator

Another possible point of view is the one focused on code generation itself. This is not a completely new concept: it dates back to 1957 with Fortran. Fortran is both a programming language with an integrated compiler, developed by IBM. A compiler can be seen as a code generation tool, since it translates source code into machine code, the only "language" truly "understandable" by a computer [20].

Compilers have a long history and have been continuously improved over time, but they are not the only tools for code generation. Already in 1976, the concept of intelligent editors emerged with Emacs, thanks to its support for custom macros [21]. Later, in 1996, IntelliSense introduced symbolic completion, namely the ability of the IDE to suggest code based on the context and the symbols already present.

In 1999, with XSLT, one of the first standardized tools for automatic transformation between markup languages was introduced [22]. During the same years, the work of Zelle[23] and Mooney[24] (1996–1999) proposed methods based on Natural Language processing to generate database queries from natural language expressions.

At the same time, template-engine-based tools for server-side spread, such as Jinja2 (2005)[25] and Mako (2006)[26], which allowed the generation of dynamic code by combining data with predefined structures.

In 2017, with research on AST-guided code generation, LSTM models began to be used to generate code in a more structured way, guided by the syntax of the programming language [27].

Finally, in 2021, with GitHub Copilot, one of the most advanced code completion tools was introduced: so efficient that it is capable of generating entire code sections from simple textual prompts, thanks to the use of generative AI models based on Transformer architectures [11].

1.1.3 LLMs code oriented

Another possible starting point is from the publication of the paper *Attention Is All You Need* in 2017, which introduced the Transformer architecture [2] and, enabled the widespread development of Large Language Models (LLMs). It should be noted that the first LLMs were typically designed to analyse and generate only natural language text, so not all LLMs were capable of generating code, or at least of generating syntactically correct or functional code.

In 2018, two foundational models were introduced: BERT [28], an encoder-only architecture designed to generate dense semantic representations of natural language, and GPT-1 [29], a decoder-only model capable of generating coherent text. Those models were not yet able to generate code, but they have been an important baseline for future code-oriented models.

In fact in 2019, OpenAI released GPT-2 [16], a more powerful decoder-only model capable of producing much more convincing text compared to GPT-1. Although GPT-2 was not specifically trained to generate code, its training corpus included code snippets. In the same year, TabNine, a popular code completion extension for several IDEs, replaced its n-gram-based next-token prediction with a version of GPT-2 fine-tuned on source code [30].

Also in 2019, Microsoft Research and GitHub introduced the CodeSearchNet dataset [31], a multilingual code dataset and one of the first large-scale corpora usable to train Transformers for code generation tasks.

In 2020, Microsoft Research released CodeBERT [32] *(based on BERT)*, trained on both natural language and code using the CodeSearchNet dataset. CodeBERT is designed for tasks such as code search and code summarization. While it does not generate code, it is an encoder-only model capable of deeply understanding the structure and semantics of source code. Through CodeBERT is possible producing rich and dense representations suitable for downstream tasks like classification, or for use as input to decoders in generative pipelines.

In 2021, several LLMs capable of generating realistic and reliable code were introduced. OpenAI published Codex [11], a GPT-3 [3] derivative fine-tuned on source code from the GitHub Code dataset. In the same paper OpenAI introduced the HumanEval benchmark, designed to assess the performance of Codex [11] on programming problems *(and in future all LLMs code oriented)*. On HumanEval, Codex (12B) was able to solve 28.8% of the problems on the first attempt, significantly outperforming all previous models on code generation.

Codex [11] sparked widespread interest in the use of LLMs for code generation, indeed in the same year is released GitHub Copilot, powered by Codex [11].

In the same year, many additional datasets were published, including MBPP (Mostly Basic Python Problems) [33], the APPS dataset [34], and CodeXGLUE [35].

In December 2021, Salesforce AI Research released CodeT5 [17], an open-source encoder-decoder model that, unlike Codex [11], supports a wider variety of

code-related tasks. Thanks to task-specific training strategies, CodeT5 proved to be highly versatile, supporting code summarization, generation, and translation.

In 2022, Google introduced AlphaCode [36], achieving a performance in the top 54.3 percentile on competitive programming tasks on Codeforces. In the same year, PolyCoder [37], a decoder-only model trained solely on 249 GB of source code, was also released as an open-source alternative.

In 2023, GPT-4 [4] (although not exclusively trained for code generation) achieved an 80% success rate on HumanEval. Claude 3 by Anthropic reportedly reached 85%, and Meta's open-source model Code LLaMA [12] scored 57%.

In 2024, Google introduced Gemini [38], which, when integrated into an inspired AlphaCode framework, reached performance within the top 15% of coding competition participants.

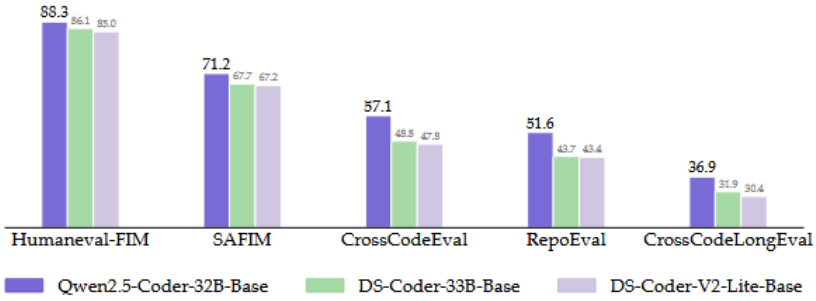


Figure 1.1: The code completion performance of competitive models on five benchmarks, Humaneval-FIM, SAFIM, CrossCodeEval, RepoEval, CrossCodeLongEval from [39].

1.2 Motivations Behind LLM-Generated Code Detection

Although it may seem exaggerated to begin the historical overview of code generation Section 1.1.2 with compilers, this choice aims to emphasize that the present work does not seek to demonize the contribution or usefulness of code generation tools. It is undeniable that such tools are extremely valuable in both professional software development, accelerating simpler phases of implementation, and in educational contexts, where they can serve as useful assistants for learning how to write code.

Nevertheless, it is equally clear that is raising the need, in certain contexts and for specific reasons, to limit or discourage the use of highly advanced tools for code autocompletion or generation.

One important reason concerns **academic and professional integrity**: the undeclared use of LLMs in evaluation contexts makes the assessment process highly problematic. Students, for example, may complete assignments or even exams using LLMs without contributing meaningfully to the generated code, potentially impairing their learning of fundamental programming concepts (getting the most out with the least effort). Similarly, during a technical interview, a candidate might rely on a tool like AlphaCode 2 to generate solutions to proposed problems. This is the main topic of *"The Impact of Large Language Models on Programming Education..."* [40] in which is demonstrated that a widespread use of LLMs code generator has negatively affects over students' capability.

Figure 1.2 illustrates possible correlation between final grade and the use of LLM. It can be observed that an increase in LLM usage tends to correlate with lower average student grades, although it is not a decisive factor in determining the final score.

Another critical issue lies in the way LLMs generate code. Since these models are trained on massive corpora, evaluating the security and efficiency of the generated code is nearly impossible. The output may be **vulnerable or inefficient** due to subtle flaws that are hard to detect, especially when the code appears well-formatted and logically structured. This happens to overreliance on commonly seen patterns in the training corpus rather than more appropriate niche solutions. The security issues are highlighted by the study *"Do Users Write More Insecure Code with AI Assistants?"* [41]. This study shows a decrease in code security simply by using an AI assistant instead of traditional programming.

Intellectual property is yet another motivation for detecting LLM-generated code, a general problem in generative AI. This is not limited to the origin of training data but also concerns the risk that an LLM may reproduce copyrighted code,

posing significant legal risks to software companies that could unknowingly integrate such code into their products. This issue is not an exaggerated concern, it's a tangible risk [42].

The least important reason, which directly concerns the development of code-oriented LLMs themselves, is the need to distinguish machine-generated code in training datasets. If a model is trained on LLMs' code, this may lead to a general LLM code-oriented **Model Collapse**. Indeed, training LLM on LLMs' code serves to deteriorate the output diversity and adaptability to real-world scenarios. This feedback loop issue is addressed for example in *"The curse of recursion..."* [43]. If no method exists to detect such code, the datasets used for training future models would have to be limited to code written before the widespread availability of LLMs, in order to avoid contamination. This would result in code generation models being trained on increasingly outdated data.

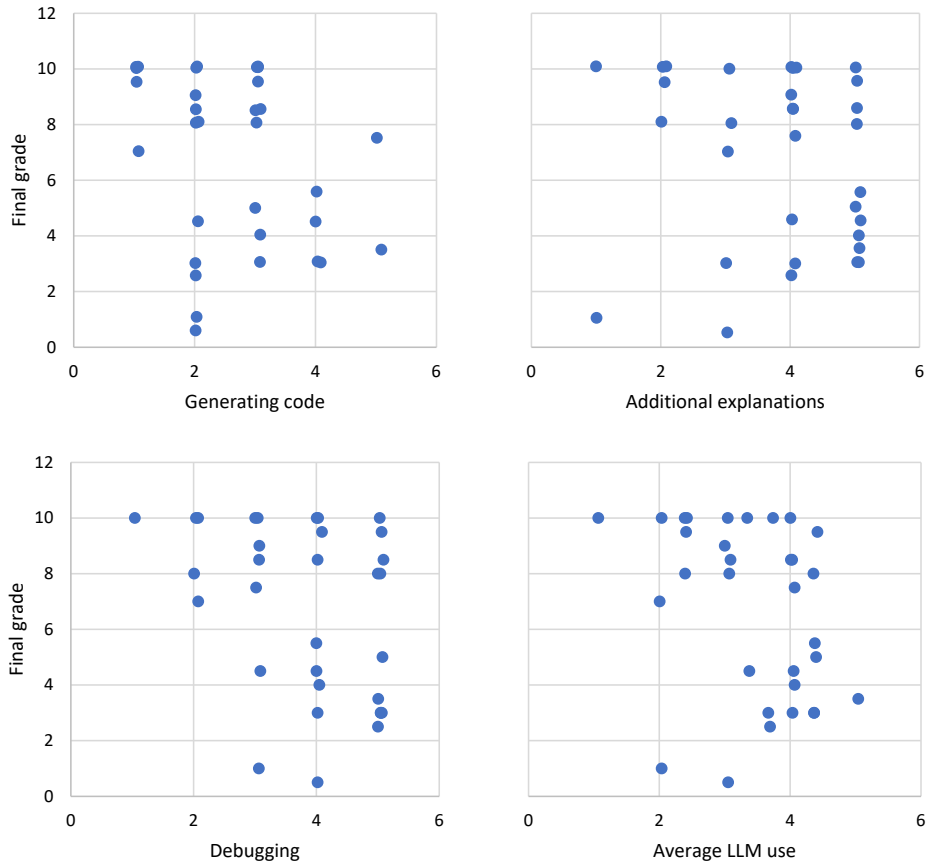


Figure 1.2: Scatter plot from the paper *The Impact of Large Language Models on Programming Education and Student Learning Outcomes*, illustrating the relationship between LLM usage and students' final performance.

1.3 Challenges in LLM-Generated Code Detection

The detection of short texts generated by LLMs remains a challenging task. While partial solutions have been proposed, the problem cannot yet be considered fully resolved. Nonetheless, it is widely acknowledged that detection tools achieve strong performance when applied to moderately sized texts (longer than just a few lines).

A well-known example is **DetectGPT** [44], a zero-shot method that does not require training a separate classifier but instead relies on an auxiliary LLM to assess whether a given passage is likely machine-generated. DetectGPT is often regarded as the state-of-the-art among open-source methods for AI-generated text detection.

In addition to open-source methods, commercial solutions such as **GPTZero** [45] have gained popularity. GPTZero can be accessed via a web interface and combines language-model-based heuristics with machine learning classifiers. While it is effective in many cases, it is not infallible. For example, GPTZero has been reported to mistakenly flag texts written by non-native speakers, whose lexical variety may be lower, as AI-generated.

Even the creators of GPTZero explicitly state that a positive detection should not be taken as conclusive proof, but rather as a probabilistic signal or indicator.

As stated in several papers analysed in this work, such as *Uncovering LLM-Generated Code: A Zero-Shot Synthetic Code Detector via Code Rewriting [46]*, **methods designed for detecting natural language text are largely ineffective when applied to code**. The causes of this limitation are primarily related to the structural and syntactic properties of programming languages.

Unlike natural language, where the same idea can be expressed using a vast variety of words and syntactic structures, source code is governed by strict and formal grammar rules. As a result, many tokens must appear in a specific and rigid order. Consequently, techniques based on lexical probability, such as those used by DetectGPT, tend to fail when applied to code, as they cannot meaningfully capture the constrained nature of programming syntax.

Moreover, many natural language detectors rely on input perturbation to assess sensitivity or likelihood distributions, a process that is significantly more difficult to perform on source code without introducing semantic or syntactic errors.

A further practical limitation is the lack of publicly available datasets specifically designed for training code-based LLM detectors. While large and well-established code corpora do exist, researchers who aim to train classification models for code detection often have to construct their own datasets, with all the associated challenges in terms of bias, coverage, and quality assurance.

These issues have contributed to a significant gap in the literature: unlike DetectGPT, which is widely recognized for natural language detection, there is no single, consolidated approach for LLM-generated code detection. Instead, the field is characterized by a proliferation of parallel methods, often employing fundamentally different techniques and evaluation protocols.

Dataset	Code LLM	Detection Methods							
		log $p(x)$	Entropy	Rank	Log Rank	DetectGPT	LRR	NPR	GPTSniffer
CodeSearchNet ($T = 0.2$)	Incoder (1.3B)	0.9810	0.1102	0.8701	0.9892	0.4735	0.9693	0.8143	0.9426
	Phi-1 (1.3B)	0.7881	0.4114	0.6409	0.7513	0.7210	0.4020	0.7566	0.3855
	StarCoder (3B)	0.9105	0.2942	0.7585	0.9340	0.6949	0.9245	0.9015	0.7712
	WizardCoder (3B)	0.9079	0.2930	0.7556	0.9120	0.6450	0.7975	0.8677	0.7433
	CodeGen2 (3.7B)	0.7028	0.4411	0.7328	0.7199	0.6051	0.7997	0.6177	0.5327
	CodeLlama (7B)	0.8850	0.3174	0.7265	0.9016	0.8212	0.8332	0.5890	0.7496
CodeSearchNet ($T = 1.0$)	Incoder (1.3B)	0.7724	0.4167	0.7797	0.7876	0.6258	0.7427	0.6801	0.6761
	Phi-1 (1.3B)	0.6118	0.4588	0.5709	0.6299	0.7492	0.4528	0.7912	0.4158
	StarCoder (3B)	0.6574	0.4844	0.6987	0.6822	0.6505	**0.7050**	0.6751	0.6299
	WizardCoder (3B)	0.8319	0.3363	0.7273	0.8338	0.5972	0.6965	0.7516	0.7068
	CodeGen2 (3.7B)	0.4484	0.6263	0.6584	0.4632	0.4797	0.5530	0.5208	0.4024
	CodeLlama (7B)	0.6463	0.4855	0.6759	0.6656	0.6423	0.6768	0.6515	0.6442
The Stack ($T = 0.2$)	Incoder (1.3B)	0.9693	0.1516	0.8747	0.9712	0.6061	0.9638	0.8571	0.9291
	Phi-1 (1.3B)	0.8050	0.4318	0.6766	0.7622	0.7295	0.4022	0.8106	0.4640
	StarCoder (3B)	0.9098	0.3077	0.7843	**0.9329**	0.6824	0.9135	0.9233	0.7715
	WizardCoder (3B)	0.9026	0.3196	0.7963	0.9010	0.6385	0.7742	0.8574	0.7794
	CodeGen2 (3.7B)	0.7171	0.4051	0.7930	0.7301	0.5288	0.7604	0.5670	0.4520
	CodeLlama (7B)	0.8576	0.3565	0.7366	0.8793	0.8087	0.8358	0.5436	0.7619
The Stack ($T = 1.0$)	Incoder (1.3B)	0.7310	0.4591	0.7673	0.7555	0.6124	0.7446	0.6787	0.6846
	Phi-1 (1.3B)	0.7841	0.4205	0.6666	0.7475	0.6718	0.4106	0.7755	0.4984
	StarCoder (3B)	0.6333	0.5025	0.7010	0.6609	0.5896	0.7080	0.6638	**0.7243**
	WizardCoder (3B)	0.8293	0.3459	0.7484	0.8223	0.6377	0.6436	0.7929	0.7766
	CodeGen2 (3.7B)	0.4816	0.6046	0.5631	0.4956	0.4337	0.5740	0.5178	0.4265
	CodeLlama (7B)	0.5929	0.5260	0.6451	0.6091	0.6116	0.6365	0.6226	**0.7494**
Average	-	0.7649	0.3961	0.7228	0.7724	0.6357	0.7050	0.7178	0.6507

Figure 1.3: Performance (AUROC) of various detection methods from [47].

1.4 Thesis Objectives

This thesis aims to pursue a structured series of objectives, each forming a necessary foundation for the subsequent stage of the project.

The overarching goal is to **develop a software tool capable of reliably distinguishing source code written by humans from that generated by Large Language Models** (LLMs). Given the lack of standardization in the current literature, this project first seeks to **establish a fair and reproducible evaluation framework**.

The initial task involves **selecting** a sympathetic and representative **dataset** that can serve both for retraining detection models and for evaluating their performance consistently. A comparative analysis of current scientific literature datasets will be conducted, taking into account objective criteria such as class balance, diversity of programming languages, code length variability, and feature distribution. The selected dataset will play a central role in ensuring that all subsequent experiments are comparable and grounded on a common basis.

Once the dataset is established, the next phase consists in **collecting and systematically evaluating the various detection models suggested in recent literature**. These models will be executed on the selected dataset using consistent and comparable metrics in order to allow for a meaningful comparison. Special attention will be given to understanding the limitations of each model like the specific scenarios in which they demonstrate optimal or suboptimal performance.

Following this benchmarking effort, the most promising method will be selected for further investigation. At this stage, will be proposed enhancements aimed at improving accuracy, robustness, or efficiency. These improvements must be tailored to the nature of the selected model.

The final phase of the project will focus on the development of a usable and **user-friendly software interface**, which allows end users to apply the selected detection model with minimal technical effort. The interface may include customizable parameters such as confidence thresholds or model options, depending on the characteristics of the adopted approach. Particular care will be taken to ensure that the tool is accessible and adaptable, so that it can be employed in diverse real-world scenarios, from academic integrity enforcement to software development auditing. Ultimately, this thesis intends to contribute both a rigorous comparative study and a practical, operational tool, filling a current gap in the literature and paving the way for future research and application in this emerging area

Chapter 2

Overview of current state of the art

Before analysing the individual contributions, it is necessary to understand the current state of the art. As discussed in Section 1.1.3, even if several methods for detecting whether source code has been written by humans or by Large Language Models (LLMs) have been proposed since 2024, given the novelty of the field, the current body of work lacks the robustness and maturity observed in similar studies focused on natural language detection.

Notably, there is no consensus regarding evaluation metrics. Furthermore, these methods are rarely evaluated on the same datasets, making it difficult to perform fair comparisons.

Some researchers have attempted to provide suitable datasets for training detection models, such as CoDet-M4 [48] and AIGCodeSet [49], but these datasets have not been widely adopted by subsequent studies. "Patents and innovation: an empirical study." [50] it was made an effort to survey existing detection methods for LLM-generated code, but the overview is incomplete as it does not include more recent contributions. Only one study has attempted to test some of the SOTA methods ("CodeMirage: A Multi-Lingual Benchmark for Detecting AI-Generated and Paraphrased Source Code from Production-Level LLMs" [51]). However, this work presents several issues that will be discussed in the thesis.

The proposed methods to detect LLM generated code vary significantly in their approaches: for example, "Codevision: Detecting llm-generated code using 2d token probability maps and vision models."[52] treats source code as an image and employs convolutional neural networks (CNNs), "Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting." [53] and "Distinguishing llm-generated from human-written code by contrastive learning" [54] use models trained by contrastive learning, while "Spotting llms with binoculars: Zero-shot detection of machine-generated text" [55] and "Between lines of code: Unraveling the distinct patterns of machine and human programmers" [47] propose zero-shot techniques inspired by the well known machine generated text

detector DetectGPT[44] and "Detection of LLM-Generated Java Code Using Discretized Nested Bigrams" [56] introduces novel discretized syntactic features to classifying.

There is also variation in the scope of these works. Certain methods are limited to detecting code from a single LLM—such as "Automatic detection of llm-generated code: A case study of claude 3 haiku."[57] focusing on Claude 3, or "Chatgpt code detection: Techniques for uncovering the source of code." [58] on GPT. Additionally, some studies restrict their analysis to a single programming language (typically Python), whereas others extend it to multiple languages, commonly including C++ and Java.

Despite the variety of methods and works proposed, most of these papers are still in a pre-publication state. Some report overly optimistic results, others have not released the code or models presented in their work, and others have released only part of the work.

It is also important to note that many papers in the literature address related but distinct problems, such as watermarking (e.g., "Codeip: A grammar-guided multi-bit watermark for large language models of code"[59], "Marking Code Without Breaking It: Code Watermarking for Detecting LLM-Generated Code" [60], "Robust and secure code watermarking for large language models via ml/crypto codesign" [61], and "Detection of llm-paraphrased code and identification of the responsible llm using coding style features" [62]) or plagiarism detection where the modification is performed by an LLM (e.g., "Detection of llm-paraphrased code and identification of the responsible llm using coding style features" [62]). These lines of research, while relevant, do not directly solve the detection problem addressed in this thesis. This variety of objectives and approaches clearly reflects the dynamic and evolving nature of the field, reinforcing the need for a solid, comparative foundation on which to build a functional and user-oriented detection system.

Chapter 3

Dataset Analysis

The goal of this chapter is to identify or construct a suitable dataset to evaluate the performance of the various detection methods proposed in the literature. It is worth noting that the literature does not only include works specifically aimed at creating benchmark datasets, but in several cases, datasets are generated solely for the purpose of validating or training the proposed detection methods or models.

3.1 Dataset Evaluation Criteria

All datasets made publicly available by the scientific community will be carefully analysed and compared. The goal is to identify the most suitable dataset—either directly or through integration, for the evaluation of code detection methods under consistent conditions. Each dataset will be assessed based on the following criteria:

1. **Total number of code samples available:** a larger number of samples generally improves statistical robustness during evaluation and model training.
2. **LLMs used to generate synthetic code:** datasets that include code generated by multiple LLMs are better suited for evaluating a method's generalization capability.
   (a) **Number of LLMs:** to avoid a method from focusing too much on stylistic features of a specific LLM, it is essential to have codes generated by different LLMs.
   (b) **Actual degree of use in contemporary times:** while it is desirable for detection methods to have long-term viability, it is undeniable that a detector effective on current LLMs (such as GPT-4 [4]) is more useful than one targeting outdated models (such as GPT-3 [3]).

(c) **Actual types diversity among LLMs:** it is also necessary to analyse the diversity of the models themselves. For instance, it is more informative to include in the same dataset code produced by both an open-source LLM like CodeLlama[12] and a commercial one like GPT-3 [3], rather than, for example, code from GPT-3 [3] and Codex [11] (which is itself based on GPT-3).

3. **Code diversity:** ensuring diversity in the code is essential to understand how generalizable a method is to the overall code domain, or whether it is specialized in the stylistic patterns of specific LLMs *(a particularly relevant issue for methods relying on stylistic features of the code).*

   (a) **Different programming languages:** while Python is currently the most relevant language in the field of artificial code generation (indeed, many datasets primarily consist of Python code such as MBPP [33]), it is valuable to evaluate detection methods across as many programming languages as possible. At the very least, it is worth assessing different detection methods on different languages, possibly recommending specific approaches depending on the language being analysed.

   (b) **Different types of code:** although language differences are important, they are not sufficient. Python, for example, can be used both as an object-oriented and a procedural language. Understanding the type of code being analysed can aid both in training and in evaluating a detector.

   (c) **Code size:** code length is a key parameter, as the ease of identifying the origin of the code may vary depending on its size. It is worth noting that in certain contexts, such as homework assignments, short code snippets are extremely common.

   (d) **Code context:** it is useful to include examples of human-written code from both competitive settings (such as Leetcode dataset [63]) and open-source project environments (such as CodeSearchNet [31]), in order to evaluate detectors across different real-world contexts.

4. **Validity information:** this includes whether the dataset provides information on the validity of human-written code, the exact prompts used to generate LLM code, or whether code quality and correctness have been validated.

   (a) **Generation prompts:** it may be useful to evaluate the different prompts used to generate the code, as it is well known that varying the prompt can lead to significantly different outputs from an LLM.

   (b) **Source of human-written code:** many datasets are based on collections of human-written code that are later used to generate LLM

outputs. Knowing the source of this human code ensures the reliability and correctness of the dataset.

(c) **Code quality:** performing tests such as verifying whether both human and LLM-generated code actually works is essential. *There is little value in analysing whether non-functional code was produced by an LLM or not.*

(d) **Paper reliability:** This point is important for preprinted papers, for which reliance on the scientific community is not possible.

Although all these are evaluation parameters, it is natural that some are more relevant than others: knowing the source of the human-written code remains more important than simply including a few additional LLMs in the code generation process.

3.2 Datasets Proposed in the Scientific Literature

The works whose main contribution is the creation of a dataset are mostly preprints, and thus require careful consideration. The papers primarily dedicated to dataset generation include: CoDet-M4 [48], AIGCodeSet [49], CodeMirage [51], Pan et al [64]. Suh et al [50].

3.2.1 CoDet-M4

The main contribution of this work is the construction of a dataset derived from existing human-written code datasets, by generating synthetic code using six different LLMs. The authors then train several models on this dataset to evaluate their performance, demonstrating that their dataset can significantly improve the effectiveness of various code detection models. The authors employ the following models as code generators:

- **GPT-4o** [65] : A proprietary model developed by OpenAI, selected to represent the state-of-the-art among large-scale, closed-source LLMs.
- **CodeLlama (7B)** [12] : An open-source model by Meta, specifically trained for code-related tasks. It is one of the most popular code-oriented language models.
- **Llama3.1 (8B)** [66] : A more recent version of Meta's Llama family. Although it is a general-purpose model, it exhibits strong performance in code generation.
- **CodeQwen 1.5 (7B)** [67] : An open-source model from Alibaba Cloud, also specialized in code, and part of the Qwen series.
- **Nxcode-orpo (7B)** [68] : A fine-tuned variant of CodeQwen. The authors include it to evaluate the robustness of detectors against different fine-tuning strategies applied to the same base model (in this case, ORPO – Monolithic Preference Optimization).

Strengths

- The introduction of one of the most extensive and diverse datasets proposed for the task of LLM-generated code detection.
- The authors also construct an "out-of-domain" dataset, specifically designed to contain code with characteristics intentionally different from the main dataset (*Section 4.4:Out-of-Domain Experiment*).

- The dataset includes code in three different programming languages, unlike other works that focus solely on Python (*Section 3.1: Data Collection*).

Weaknesses

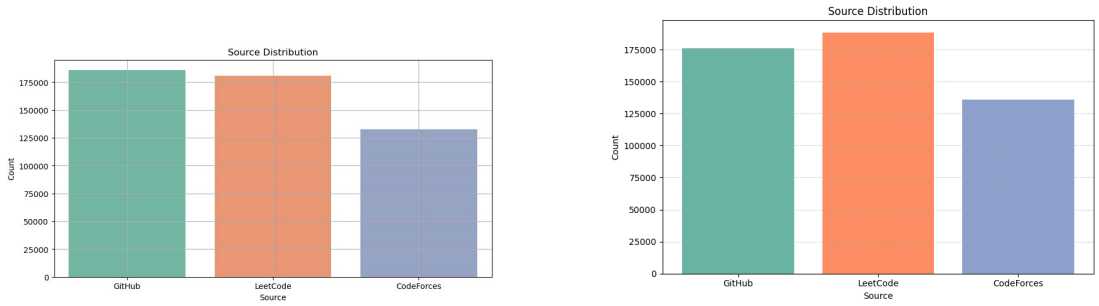
- Suspiciously high in-domain classification metrics, with F-scores exceeding 98% (*Table 2*).
- The prompts used to generate synthetic code vary depending on the source, potentially introducing an unintentional correlation between code types and the generation prompts (*Appendix F*).
- Most of the LLMs used for code generation are lightweight models, with GPT-4o being the only large-scale LLM involved (*Section 3.2: Code Generation*).
- Once the dataset is downloaded from the official portal on huggingface, several important intuitive reliability metrics are found to be missing:
  - The declared train/validation/test split described in the paper is not included; only the test set is provided. This makes **reproducing their results more difficult**.
  - Some significant **code-level features are missing**, such as whether the code sample is class-based or function-based.
  - The authors state in the paper that they plan to keep updating the dataset, which may explain small discrepancies in the number of code samples compared to what is reported. For example, there are fewer GitHub samples than those claimed in the original publication as shown in Figure 3.1 The strange think is that the number of code not only increased but also decreased.
- Focusing on the human code (*Section 3.1: Data Collection*):
  - **LeetCode corpora**: Pan et al. [64] (5,069) and others (2,800) so the total should be: 7,869 but according to the paper table we should have 13,528 codes Figure 3.1 *(we don't know the source of half of the codes).*
  - **Code-Forces corpora**: the dataset has 103,792 codes but they are all solutions of only 2,523 problems from a publicly available Kaggle dataset [69].
  - **GitHub corpora**: 135,566 codes from CodeSearchNet [31] and GitHub API in 2019 *(code not up to date)*
- Even if the dataset as a whole may appear fully balanced, clear imbalances emerge when focusing solely on the human-written code:

- There is an imbalance in the number of code samples generated by different LLMs (with relatively few generated by GPT models Figure)
- LeetCode human-written code samples are significantly fewer, meaning that many more generations were performed on LeetCode problems compared to other corpora.

Code Quality

In order to preserve quality in the LLMs code source:

- *"For LLM-generated code, we filtered irrelevant responses and extracted code from the LLM output."*
- *"After collecting the datasets, we* ***removed all comments and docstrings****."*
- *"We also filtered codes based on length, excluding those below the 5th or above the 95th percentile in token count."*
- *"Finally, we deduplicated the dataset to prevent potential code memorization."*



(a) Paper source distribution

(b) Actual source distribution

Figure 3.1: histogram differences

Final evaluation

Total number of code samples available	
(a) Total number of human code	246,221
(b) Total number of LLMs code	254,331
LLMs used to generate synthetic code	
(a) Number of LLMs	6 LLMs: *(GPT-4o, CodeLlama 7B, Llama3.1 8B, CodeQwen 1.5 7B, Nxcode-orpo 7B)*
(b) Diversity among LLMs	3 different source models: *(GPT-4o, CodeLlama, CodeQwen)*
(c) Actual degree of use in contemporary times	Average LLMs release date: 2024
Code diversity	
(a) Different programming languages	Python, Java, C++
(b) Different types of code	Functional, Object-Oriented *(missing information in the dataset)*
(c) Code size	1st percentile: 42 words, 3rd percentile: 68 words
(d) Code context	Mixed (competitive and open-source)
Validity information	
(a) Generation prompts	Provided (different between LLMs)
(b) Source of human-written code	GitHub, Codeforces [69], Pan et al. [64]
(c) Code quality	Comment removal, Duplicate code removal
(d) Paper reliability perception	Low, missing or inconsistent information

Table 3.1: Evaluation Summary: CoDetM4 Dataset

Despite being a very interesting dataset, especially due to the large amount of available code, it proves to be unreliable both because of the authors' lack of precision and due to some of its intrinsic characteristics, such as an unusual data imbalance. While it is not a dataset to be entirely discarded, its use must certainly be carefully evaluated.

3.2.2 AIGCodeSet

The main contribution of this paper is also the preparation of a dataset for evaluating detection methods. Unlike other works, this dataset includes both functional and non-functional human-written code, as it evaluates LLM code generation starting both from a plain prompt and from existing code.

- **Gemini 1.5 Flash** [70] : Proprietary, available via Google Cloud's Gemini API and Vertex AI; no public weights. Multimodal corpora including code, text, images, audio, and video; extends context handling to millions of tokens.
- **Codestral-22B** [71] : Autoregressive code modeling with fill-in-the-middle (FIM) capability; excels at cross-file completion tasks.
- **CodeLlama-34B** [12] : State-of-the-art among open models on HumanEval.

Strengths

- They evaluate LLM code generation under three different scenarios (*Section 3.2:Generating Code Snippets*):
  - textual problem descriptions taken from CodeNet (problem prompt);
  - problem prompt + human source code with a runtime error;
  - problem prompt + human source code with an output error.
- The official dataset on huggingface is well documented.
- The problems are of varying difficulty.

Weaknesses

- The dataset contains python code only.
- The dataset contain competitive code only.
- All the codes are solution of **only 300 problems** *(60 problem x 5 different difficulty)*
- GPT, probably the most widely LLM family used among students, is not included among the evaluated models.

Code Quality

In order to preserve quality in the LLMs code source they manually removed:

- *"code with failure to produce any output"*
- *"C-family languages instead of Python."*
- *"presence of meaningless characters, sentences, numbers, or dots."*
- *"Any explanations provided above or below the code. "However,* ***comments embedded within the code blocks were retained****, provided they were appropriately marked as comment lines.""*

Final evaluation

Total number of code samples available	
(a) Total number of human code	9,50
(b) Total number of LLMs code	5,650
LLMs used to generate synthetic code	
(a) Number of LLMs	3 *Gemini 1.5 Flash, Codestral-22B, CodeLlama-34B*
(b) Diversity among LLMs	3 different source models
(c) Actual degree of use in contemporary times	Average LLMs release date: 2024
Code diversity	
(a) Different programming languages	Python
(b) Different types of code	unspecified
(c) Code size	1^{st} percentile: 30 words, 3^{rd} percentile: 50 words
(d) Code context	competitive
Validity information	
(a) Generation prompts	provided
(b) Source of human-written code	CodeNet by IBM [72]
(c) Code quality	Executable check
(d) Paper reliability perception	Hight

Table 3.2: Evaluation Summary: AIGCodeSet Dataset

The dataset appears to be of high quality, but compared to other works, the variety of code is extremely limited. Moreover, GPT is not included among the

LLMs, and there is no guarantee that the LLM-generated code actually functions correctly.

3.2.3 CodeMirage

The work and dataset presented in the CodeMirage publication aim to evaluate ten LLM code detection methods. The dataset was also used to train the detectors (at least those requiring training). They employed as many as ten different LLMs for code generation.

- **GPT-4o-mini** [65] : A proprietary model developed by OpenAI, selected to represent the state-of-the-art among large-scale, closed-source LLMs.
- **GPT-o3-mini** [73] : A smaller model from OpenAI's o3 series, known for high performance in STEM and logic benchmarks despite reduced parameter count.
- **Qwen-2.5-Coder-32B** [39] : A specialized, code-centric LLM. Trained on high-quality multilingual code data, it supports over 40 programming languages and offers a 128K context window
- **Claude-3.5-Haiku** [57] : The smallest member of Anthropic's Claude 3.5 family
- **DeepSeek-R1** [74] : A reasoning-optimized model developed using reinforcement learning to enhance multi-step logical thinking.
- **DeepSeek-V3** [75] : A large-scale, general-purpose model from DeepSeek AI.
- **Gemini-2.0-Flash** [76] : Although designed for speed, it retains strong performance across code and reasoning tasks.
- **Gemini-2.0-Flash-Thinking** [76] : An experimental version of Flash that integrates internal "thinking steps" during generation.
- **Gemini-2.0-Pro** [76] : The full-sized flagship model from the Gemini 2.0 series, comparable to GPT-4 in overall performance.
- **Llama-3.3-70B** [77] : A 70-billion-parameter open-source model from Meta AI, part of the LLaMA 3.3 release. It represents one of the most capable open LLMs available to the public.

Strengths

- The entire generation process is well documented; both the Code Summarization methods (necessary to obtain the prompts to be given to the LLMs for generation) and the code generation prompts are easily accessible Figure **??**. (*Section 3:CodeMirage Framework*)
- The dataset also includes code samples with Paraphrasing, meaning simulated modifications of small stylistic elements in the code generated by LLMs. (*Section 3.1:Benchmark Construction*)
- The dataset, available on the official huggingface portal is well documented.
- The dataset is perfective balanced.

Weaknesses

- The work itself explicitly requires the LLM-generated code to match the length of the original human-written code in the generation prompt, which is unrealistic and disadvantages stylistic detection methods. (*Section 3.2:Benchmark Statistics*)
- All human-written code comes exclusively from the **CodeParrot GitHub-Code-Clean dataset** [78], meaning that the dataset covers only a single domain.
- A comment-free version of the code is not provided, which leads to some code samples being heavily commented in a distinctly human style Figure **??**. It is not specified whether comments are removed prior to the detection process; if not, many results could be significantly biased.
- Although the paper does not explicitly highlight this, for each human-written code sample, two LLM-generated versions are produced rather than just one. This means that the actual ratio between human and LLM-generated code is 1:20 Figure 3.2.

Code Quality

In order to preserve quality in the LLMs code source:

- *"consistency with the original human-written code's line count and character length"*
- *"adequate token-level divergence from the original, enforced by requiring a BLEU [79] score below 0.5 to avoid recitation."*

Final evaluation

Total number of code samples available	
(a) Total number of human code	10,000
(b) Total number of LLMs code	199,988
LLMs used to generate synthetic code	
(a) Number of LLMs	10 LLMs: *(GPT-4o-mini, GPT-o3-mini, Qwen-2.5-Coder-32B, Claude-3.5-Haiku, DeepSeek-R1, DeepSeek-V3, Gemini-2.0-Flash, Gemini-2.0-Flash-Thinking, Gemini-2.0-Pro, Llama-3.3-70B)*
(b) Diversity among LLMs	6 different source models: *(GPT, CodeQwen, Claude, DeepSeek, Gemini, Llama)*
(c) Actual degree of use in contemporary times	2024-2025
Code diversity	
(a) Different programming languages	*(HTML, JavaScript, Java, C, Python, Ruby, Go, C#, C++, PHP)*
(b) Different types of code	unspecified
(c) Code size	1st percentile: 639 words, 3rd percentile: 804 words
(d) Code context	open-source
Validity information	
(a) Generation prompts	Provided (one)
(b) Source of human-written code	GitHub
(c) Code quality	Statistical alignment
(d) Paper reliability perception	Midium, *(no precise references are provided regarding the data source)*

Table 3.3: Evaluation Summary: CodeMirage Dataset

Although the dataset appears to be well-varied and balanced, the lack of an evaluation of the actual functionality of the code may compromise the overall assessment. Furthermore, competitive-style code samples are completely absent.

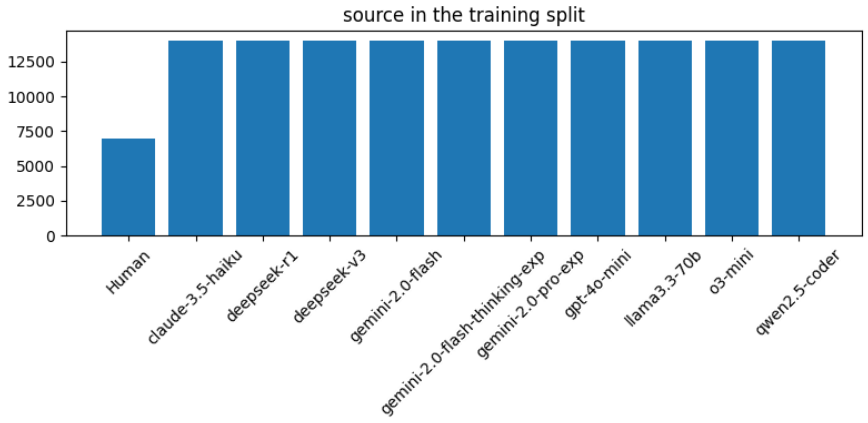


Figure 3.2: Source distribution

3.2.4 Pan et al

The main objective of this paper is to demonstrate, beyond any doubt, that detection methods commonly used for natural language are not reliable when applied to code. To achieve this, the authors construct a dedicated dataset.

- **GPT**: In the paper is not specified what gpt.

Strengths

- Generate code by different Prompt.
  - Prompt explaining the problem and asking for a solution;
  - Removal of stop words from the prompt;
  - Request to imitate a human;
  - Request to generate code without any comments;
  - Request to also generate testing code;
  - Solution with generic test cases not specifically tailored to the generated code;
  - Solution using the unfittest framework;
  - Explicit request for a long code;
  - Explicit request for a short code.
- Use different paraphrasing method.
  - Replacement of variable names;
  - Replacement of function names;
  - Replacement of both variable and function names;

  - Addition of unnecessary code to the solution.

- The source problem is specified in the dataset.

Weaknesses

- The dataset is not available on hugging face but is necessary download it from a Replication Package
- In the paper is not specified what gpt is used in order to generate code.
- Only competitive code.

Code Quality

In order to preserve quality in the LLMs code source:

- *"manual valiation to ensure the correctness of the dataset"*

Final evaluation

Total number of code samples available	
(a) Total number of human code	5,069
(b) Total number of LLMs code	65,897
LLMs used to generate synthetic code	
(a) Number of LLMs	1 *GPT*
(b) Diversity among LLMs	1
(c) Actual degree of use in contemporary times	unspecified
Code diversity	
(a) Different programming languages	Python
(b) Different types of code	unspecified
(c) Code size	1st percentile: 54 words, 3rd percentile: 83 words
(d) Code context	competitive
Validity information	
(a) Generation prompts	provided
(b) Source of human-written code	Quescol [80], Kaggle [81]
(c) Code quality	filtered by human
(d) Paper reliability perception	Peer-review paper

Table 3.4: Evaluation Summary: Pan-et-al Dataset

Despite the reliability of this work, given that it is a peer-reviewed paper, the dataset consists of only a single programming language and a single LLM (which is known to be GPT, though the specific version is not disclosed). Moreover, it is not explicitly stated whether the generated code is actually functional.

3.2.5 Suh et al

The goal of this paper is both to evaluate detectors originally designed for natural language (which, as expected, proved to be ineffective), and to propose alternative approaches, including fine-tuning LLMs, an approach that was effective only for code belonging to the same training domain. To achieve these results, the authors developed a large dataset, which should therefore be rightly considered as a separate contribution.

- **GPT (GPT-3.5)**:
- **GPT-4**:
- **Gemini Pro (Gemini 1.0)**:
- **Starcoder2-Insgruct (15B)**:

Strengths

- Use different domain dataset.
  - **MBPP**: didactic dataset
  - **humaneEval-x**: a dataset form OpenAI
  - **Code Search Net**:

Weaknesses

- It is difficult to obtain the dataset (Dataset creation is not the aim of the paper).

Code Quality

In order to preserve quality in the LLMs code source:

- *"We also removed code snippets that contain syntax errors"*

Final evaluation

Total number of code samples available	
(a) Total number of human code	3,700
(b) Total number of LLMs code	29,500
LLMs used to generate synthetic code	
(a) Number of LLMs	4 *chatgpt, gpt-4, Gemini pro, stracider2-instruct 15B*
(b) Diversity among LLMs	3 *gpt, gemini, stracider*
(c) Actual degree of use in contemporary times	Average year: 2024
Code diversity	
(a) Different programming languages	java, C++, python
(b) Different types of code	unspecified
(c) Code size	1st percentile: words, 3rd percentile: words
(d) Code context	Both
Validity information	
(a) Generation prompts	not in the paper
(b) Source of human-written code	MBPP, humaneEval-x, Code Search Net
(c) Code quality	Sintax quality
(d) Paper reliability perception	Peer-review paper

Table 3.5: Evaluation Summary: Suh Dataset

Unfortunately, the data they used is no longer available through the official portals, which, at the time of writing, are no longer accessible. As a result, this dataset is effectively unusable.

3.3 Considerations

The variety of available data is undeniable. Nevertheless, during the analysis, an issue emerged that is rarely addressed in the reviewed papers: the correctness of LLM-generated code. It is self-evident that anyone using LLM-generated code would, before committing it, most certainly run at least one test to verify that the code works. Moreover, for our purposes (Section 1.2), whether a student or candidate submits non-functional code, it makes little difference whether it was written by them or by an LLM.

A relevant concern arises from the lack of guarantee that the LLM-generated code present in the test datasets is correct. Another important question is whether there is a correlation between incorrect and correct code in the detection of LLM code. According to some analysed studies, the answer is affirmative: 'The results demonstrate that detecting correct code is more difficult than detecting incorrect code, both with our method and baselines.' [46]

Therefore, the consideration that every detection method should be evaluated specifically on **functional** LLM-generated code, rather than on LLM code in general, proves useful in preventing a significant gap between the reported metrics and the actual user experience. Such a gap may arise if detection methods exhibit a classification bias due to the incorrect behaviour of the code. It is worth noting that while many works perform syntax or runtime checks, none explicitly mention functional testing.

3.3.1 Test framework

Given the evaluations presented by the works I propose, the detection methods, it was necessary to develop a framework that could ensure the LLM code was certainly correct. The minimum result was to obtain at least a subdataset that would allow evaluating whether there was a difference between detection of functioning and non-functioning code.

So a small framework was developed and published on GitHub. This framework had important constraints:

- It had to work with Python (the most generated programming language)
- It had to work on LLMs that could run on home hardware,
- It had to support as many LLMs as possible,
- It had to allow for the interruption and resumption of test generation and the tests themselves at any time, in order to evaluate error conditions and bugs.

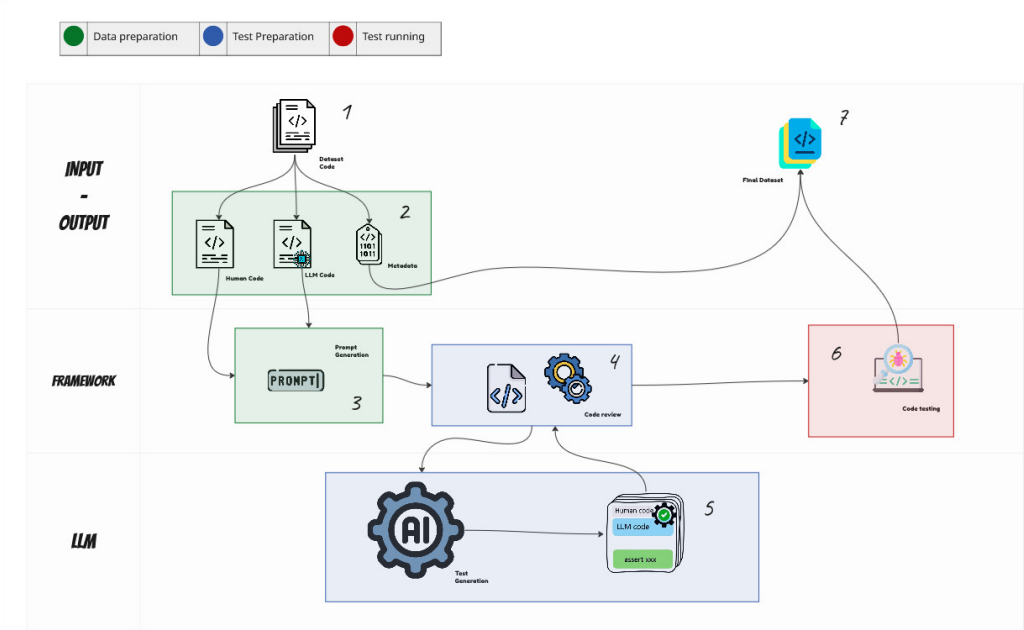


Figure 3.3: Framework overview

The framework was divided into several sections, so it could be easily modified for potential future needs that required dataset generation by an LLM.

1. **Input:**The framework is capable of processing any dataset that contains both LLM-generated code and human code that solve the same problem.

2. **Input\Data preparatin:** Any supported dataset is converted into a JSON file, specifying which columns in the dataset correspond to *"human code"* *"LLM code"*, and *"problem explanation"* (the latter being optional and included only to provide a more complete prompt to the LLM).

3. **Framework\Data Preparation:** A prompt template has been prepared for an instruct-LLM. The prompt asks the LLM to understand both the LLM code and the human code that solve the same problem, with potentially different input formats. The model is asked to generate code that evaluates whether the two implementations return the same output values given the same input values (even if provided in different formats).

4. **Framework\Test Preparation:** Since the LLMs used were "lightweight", both before passing the code to the LLM and after obtaining the final .py file, small adjustments are applied, such as: renaming classes and functions to ensure that the two solutions have different function and class names, adding missing imports (a common cause of test failures), and so on.

5. **LLM\Test Preparation:** The LLM generates 3 inputs and 3 control assets in Python.

6. **Framework\Test Running:** The framework therefore generates a folder to insert the test code, executes it, and collects the results, saving the number of tests passed for each code.

7. **Output:** The test results are collected and added to the metadata saved at the beginning of the process, with the aim of obtaining a final dataset containing information about the test execution.

Development insights

Firstly, it is important to underline a fundamental consideration: the reason for not asking the LLM to directly classify the code as correct or incorrect is supported by the findings of the paper *"Do LLMs generate test oracles that capture the actual or the expected program behaviour"* [82], which states: *"LLMs are more effective at generating test oracles rather than classifying them."*

The framework has been developed starting from the project **KodCode**[83], available on GitHub. KodCode aims to generate code from prompts in order to train LLMs on synthetic code. However, the project underwent huge changes, leaving very little of the original code (like the *.jsonl* use).

Initially, the idea was to slightly modify the `KodCode` framework so as not to require a large language model (LLM) to generate both code and tests, but only the tests. However, during the development and analysis of the framework, it became evident that much more substantial changes and additions were necessary.

First and foremost, it was necessary to develop a method for preprocessing and preparing a dataset of pre-written code. Subsequently, the entire generation phase had to be adapted to meet the new objectives.

After initial testing, it became clear that an LLM of moderate size (14B), a necessary compromise due to hardware limitations, was not sufficiently capable of understanding the problem and the code to generate effective tests. In many cases, the tests produced were incorrect and would have incorrectly flagged correct code as faulty.

As a result, it became necessary to change the approach: rather than relying solely on the LLM's test-generation capabilities, functioning reference code that solved the same problem was used. Since the dataset proposed by Pan et al. includes both LLM-generated code and corresponding human-written code (the latter being reliably correct), the analysis concentrated on that dataset.

The new strategy required generating inputs consistent with both the problem statement and the function signatures. The LLM's task was thus limited to generating such inputs and writing code to verify whether both the human-written and LLM-generated implementations would produce the same outputs for those inputs.

Several challenges emerged from this new approach, some of which are outlined below:

1. **Lack of Imports**: Human-written code in the dataset did not include any imports. Therefore, a mechanism was introduced to automatically insert the most common imports, as the LLM proved ineffective at consistently adding all necessary ones.

2. **Code Structure Variability**: Not all solutions were simple functions—some were classes, and others were standalone Python scripts. For class-based solutions, the prompts were adapted to enable the LLM to instantiate objects correctly and generate the necessary tests. For script-based code, such examples were discarded, since converting them into testable functions might have introduced errors attributable to the transformation process rather than to the code itself.

3. **Name Collisions**: In many cases, the human-written and LLM-generated code used identical class or function names. To address this, an algorithmic renaming strategy was implemented to avoid such conflicts.

Subsequently, it became necessary to develop an entire sub-framework responsible for effectively generating the test files. These files needed to contain both the human-written and LLM-generated functions, as well as the tests produced by the LLM.

The sub-framework was also tasked with executing multiple tests in parallel, reporting the result of each individual execution, and generating a new dataset

based on the previous one, but enriched with additional information regarding the outcome of the tests.

Despite the significant effort invested in developing the framework, it was decided to maintain, at least in this phase, certain limitations—potentially addressable in future work, such as compatibility with the Python programming language only.

Possible improvements

Despite the work done, there is still significant room for improvement:

- Predefined prompts depending on the LLMs
- Support for APIs for test generation
- Test execution within containers separate from the execution environment
- Improvement of library import techniques

3.4 Final Dataset

Upon analysing the available datasets, it was decided to use several of them for specific purposes.

3.4.1 Pan et al tested dataset

The analysis of the dataset by Pan et al. enabled the construction of a small dataset consisting of definitively correct and definitively incorrect code samples (where "incorrect" refers to cases in which the LLM-generated code did not produce the same outputs as the corresponding human-written code).

The process began with the subset of the dataset corresponding to the prompt *"Removal of stop words from the prompt"*, which includes 5069 samples. From this subset, has been discarded all code samples that the framework was unable to process. Tests were then executed on the remaining samples, and only those which did not produce runtime faults were retained—this was necessary to exclude failures due to incorrectly generated tests or missing import statements.

As a result, a dataset of approximately 1000 code samples was obtained, of which around 600 were classified as correct and 400 as incorrect (i.e., they failed at least one test).

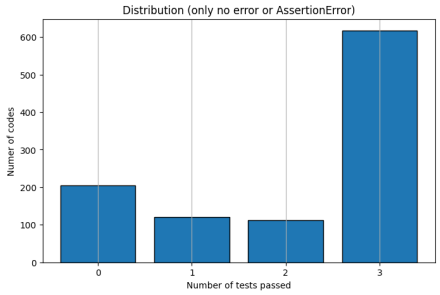


Figure 3.4: Source distribution

3.4.2 Dataset standardization

Different *"roles"* have therefore been assigned to the datasets in the testing process.

CodeMirage 3.2.1 is the primary test dataset, against which all methods will be compared. This choice is motivated by the fact that CodeMirage has already published tests on various methods using its dataset, providing a solid starting point. Additionally, the CodeMirage dataset is the most diverse and extensive: it contains both short code snippets and entire programs, includes 10 programming languages, and 10 different LLMs. This dataset is therefore useful for evaluating the generalization of results across multiple programming languages and LLMs.

AIGCodeSet 3.2.2 plays the role of evaluating, among a set of methods selected through the tests on CodeMirage, the effectiveness on competitive-type code. Competitive code is an extremely important type of code because it requires the greatest detection capabilities. This is because competitive code refers to coding exercises, such as those requested during job interviews and in educational settings. Additionally, AIGCodeSet can also be used to make secondary considerations about the detection method's capabilities on more complex code.

The code from Pan et al. tested 3.4.1 using the framework developed in 3.3.1 allows us to assess the considerations of various works regarding the increased difficulty of detecting error-free code. This test remains a consideration, and if proven true, it should encourage future works to make more intensive use of testing frameworks to improve the quality of their datasets, both for training and testing.

CoDet-M4 3.2.1 does not appear to be a high-quality dataset, but it remains a dataset of enormous size. Therefore, although it is not recommended for testing, it could be interesting to use this dataset to train an AI-based detection method and evaluate its results.

So the final role of the dataset will be:

- **CodeMirage:**
  - Language generalization
  - LLMs Generalization
- **AIGCodeSet:**
  - competitive code evaluation
  - difference between difficulty
- **Pan:**
  - correct code vs wrong code evaluation
- **CoDET-M4:**
  - Traning set for AI based detection

It is important to note that in subsequent phases it will be indicated that, for some methods, the entire available dataset will not be used, possibly to avoid training or test sessions that are too long. To alleviate this issue, a framework was developed to perfectly balance any requested sub-dataset through a convenient graphical interface. This topic will be revisited later in Chapter 5.

Chapter 4

Detection Methods

4.1 Evaluation Metrics

Let us recall that the ultimate goal is to build a binary classifier capable of determining whether a piece of source code was written by a human or generated by a Large Language Model (LLM). For binary classifiers, **accuracy** is typically the default evaluation metric, unless one classification outcome, such as true positives or false positives, is considered more critical than the other.

Given that this classifier can be used in scenarios where a programmer could be 'accused' of not having authored the code themselves, it is essential that such claims be made with a high degree of confidence. While it is certainly desirable to allow end users to configure the decision threshold, the default setting should prioritize a **low FPR** (false positive rate) to prevent unjust accusations.

> "In previous experiments, we mainly use F1 score, which is a threshold-dependent measure that balances precision and recall, but F1 can be misleading in real-world detection tasks. As it gives equal weight to false positives and false negatives [...] it often fails to reflect performance in imbalanced settings or under strict false-alarm constraints. By contrast, reporting the true positive rate at low false-positive rates directly measures how many genuine positives the model catches when false alarms must be kept to a minimum." [51].

Given these considerations, the most interesting metrics for evaluating the difference between different detection methods are the TPR (True Positive Rate) at a fixed FPR (False Positive Rate), as well as the F1-Score, which allows for a general comparison between multiple detection methods in cases where the FPR is not a critical parameter for the method's application domain.

Therefore, the following will be evaluated: F1-Score, TPR@FPR = 10%, and TPR@FPR = 1%.

4.2 Evaluation Methods

Analyzing various works that propose detection methods, several "adversarial scenarios" have emerged, which can be considered more or less challenging:

1. **Out-of-domain:** All machine learning-based methods struggle significantly when tested on datasets different from the training dataset. It is therefore interesting to evaluate the performance loss between different datasets, considering:
   (a) **Type of code** present in the dataset *(length, paradigm, language, etc)*
   (b) **LLMs** in the dataset different from the LLM used during training.
2. **Comment removal:** It would be extremely easy for a user to remove all comments within the code. The reason is clear: comments are essentially natural language, and it is likely that a detection method could exploit comments to detect LLM-generated text.
3. **Indentation removal:** Optional indentation could be another challenge, as removing or modifying the indentation might confuse detection methods that rely on structural features of the code.
4. **Paraphrasing:** It would not be difficult for a user to change variable and function names, perhaps calling them with more standard names. Changing names becomes another realistic and difficult scenario for detection methods.

The probable reason why the **Out-of-domain scenario** is so challenging may be related to the following statement: *"Every programmer develops a unique coding style, shaped by their routines and habits. Similarly, generative models, like experienced programmers, exhibit distinctive coding patterns influenced by the biases present in their training data. Therefore, LLMs can be viewed as programmers with consistent coding styles shaped by these inherent biases."* [46] Thus, it can be thought that the goal of a detection method for LLM-generated code is simply to identify a specific coding style (whether from a particular programmer or from an LLM). It can be expected that multiple LLMs share common traits, because: *"Since LLMs are trained on vast corpora of data from the Internet, their training data likely exhibit significant similarities, leading to similar behaviors across models."*[84]

That said, some tests are more complex and less realistic than others. For example, out-of-domain tests are more relevant in real-world settings than indentation removal. For these reasons, the focus will be on the most important metrics (F1-score, TPR@FPR = 10%, and TPR@FPR = 1%) on in-domain, **out-of-domain**, and **comment-removal** tests.

It is explained from now that all test plots will include these three selected metrics (F1-score, TPR@FPR = 10%, TPR@FPR = 1%). In particular, the values of F1, TPR@FPR = 10%, and TPR@FPR = 1% averaged across languages will be reported. At the same time, an error bar computed via the standard deviation on the average values will be shown. Variation across programming languages, not across LLMs, was chosen for reasons of informational utility. If the joint variability across languages and generating LLMs were evaluated, the source of uncertainty would be difficult to understand. Moreover, uncertainty due to differences between LLMs is less significant (because more tied to training data) than that due to the programming language. If variation across LLMs is of interest, it is more useful to display the performance that the model obtains on a specific LLM.

This policy was not applied solely to the metrics reported by the CodeMirage paper (figure 4.1), which appear to have been evaluated both over LLM and over language.

To allow anyone to run tests for any preferred LLM, a graphical interface was created that allows easy testing of the methods presented in this work. This part will be detailed in Section 6.

4.3 Evaluation of Existing Methods

4.3.1 Baseline Considerations

It should be recalled that the work of CodeMirage[51] was not only to provide a dataset, but also to perform tests on some of the methods present in the literature.

The methods tested by CodeMirage can be divided into 4 categories:

1. **Zero-shot detectors**: These are methods based on the computation of code-related metrics and do not require any training.
   The advantage of these methods is that they are totally independent of any dataset and therefore "immune" to the *out-of-domain effect*. This characteristic is very useful if the detection method must be per-formant in different domains.
   Unfortunately, these methods also turn out to be complicated to develop because they require a deep analysis of every type of code generated by an LLM, unless such characteristics exist and are detectable by a non-AI-based method.
   Moreover, it is essential to identify characteristics that depend intrinsically on the architecture of LLMs and not on their training, otherwise these methods would become even less reliable over time than machine-learning-based methods.

2. **Embedding-based detectors**: These methods consist of using an encoder-only transformer (or one that can be used as such), trained on code and not fine-tuned for the classification task, paired with a classifier trained on a dataset of LLM and human code.
   These methods always remain inferior to the performance achieved by the next category.

3. **Fine-tuned detectors**: Fine-tuned methods are like embedding methods, but during training they also train the encoder-only. The difference between the various methods lies only in the choice of the encoder-only model. They seem to be the models that in general achieve the best performance but are also the most affected overall by the out-of-domain effect.

4. **Pretrained LLM with downstream detector**: The last category is a middle ground between the previous ones and seems to suffer overall less from out-of-domain problems. This category statistically analyses the output of an LLM to which the code has been provided via a prompt, such as `\explain this code:{code}"`. Such statistical features are used by a classifier to learn which ones identify human code more and which ones identify LLM code.

CodeMirage problems

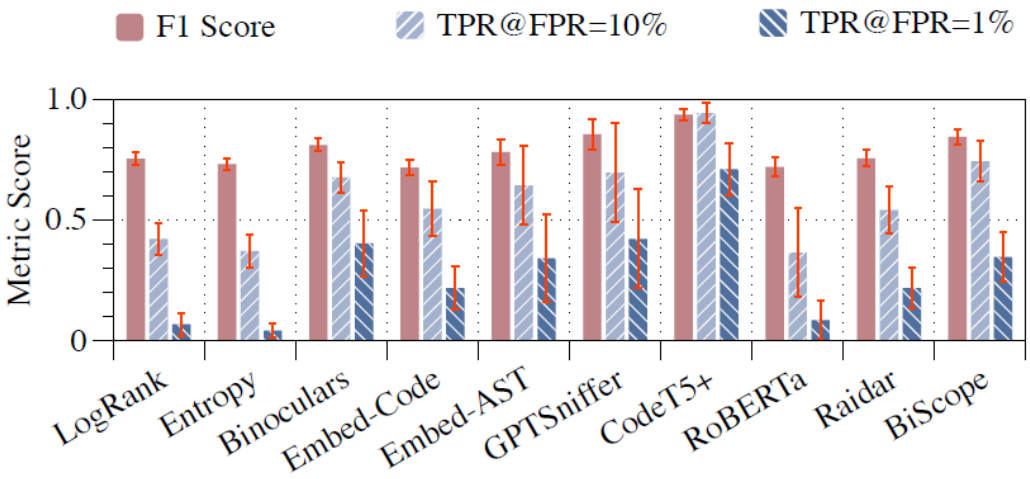


Figure 4.1: CodeMirage's tests

The presented methods therefore fall into the previously described categories: Zero-shot detectors: logRank[85], entropy[86], binoculars[55]. Embedding-based detectors: Embed-Code, Embed-AST. Fine-tuned detectors: GPTSniffer[87], CodeT5+[88], RoBERTa[89]. Pretrained LLM with downstream detector: Raidar[90], Bioscope [84].

Furthermore, these tests are not complete: recall that the CodeMirage dataset includes only code coming from GitHub and tends to be large. In addition, (more importantly), they never mention having removed comments in the code, which not infrequently (by analysing the dataset) appear to be extremely long and verbose. For this reason it is plausible that zero-shot methods designed for natural text apparently achieve good results on code as well (contrary to what many other works state: *zero-shot text detectors are ineffective in detecting code, likely due to the unique statistical properties found in code structures.* [91]).

It is important to make some clarifications: all the zero-shot methods presented were designed to analyze natural text which, as previously stated, is a process extremely different from analyzing code due to the naturally low next-token perplexity (a feature on which almost all zero-shot methods are based). The Embed-Code and Embed-AST methods are both based on CodeXEmbed-2B[92], where in the first case code is provided and in the second case the AST obtained from code. GPTSniffer, CodeT5+, and RoBERTa are three methods that rely entirely on the GPTSniffer[87] technique and differ only by the encoder used. Finally, both Raidar and Bioscope are methods designed especially for natural text, but they seem to achieve good results on code as well.

For all the above reasons, it is justified not to stop at these tests but to pursue more in-depth analyses. In particular, additional methods not considered by

CodeMirage and specifically designed for LLM code classification will be examined. Nevertheless, the CodeMirage results should not be ignored.

Key points: the method that appears to deliver the best performance is GPTSniffer with CodeT5+. BiScope also merits consideration. Rationale: CodeT5+ attains very high TPR at a 10% FPR. However, these tests were run on a split of the CodeMirage dataset. Therefore, GPTSniffer with CodeT5+ was not evaluated out of domain, which is the main weakness of fine-tuned detectors, the category to which GPTSniffer with CodeT5+ belongs.

Similarly, BiScope belongs to the pretrained LLM with downstream detector category, which has fewer difficulties adapting to out-of-domain tests. It also achieves positive TPR at an FPR of 10%. Therefore, GPTSniffer with CodeT5+ and BiScope will be considered the baselines for subsequent models.

It must be noted that CodeMirage has a major issue. In their work, it is never specified that the codes on which the methods are tested operate on clean code. By clean code it is meant code in which the natural-language sections (comments) have not been removed. This is actually fundamental for testing because it shows that the detection method does not rely on natural text but on code. Moreover, the approach would become even less dataset-independent, since performance would risk varying extremely depending on the size of a code's comment section. A very strong and perhaps obvious statement can therefore be made: **any test on any dataset to which at least comment removal has not been applied is not to be considered valid for LLM code detection**.

4.3.2 Methods to test

An extensive survey of LLM code-detection methods in the literature was conducted. Many works were available only as preprints. Several seemingly promising papers could not be tested. For example, [52] appears to leverage CNNs to analyse images of LLM- and human-written code for classification. Unfortunately, neither code nor pretrained models were accessible. Consequently, only a small set of other methods specifically designed for LLM code detection was evaluated.

BiScope

BiScope is a method designed for natural-text detection but was also tested by its authors on code detection.

	Normal Dataset					Paraphrased Dataset				Normal	Normal	Paraphrased
Method	GPT-3.5 Turbo	GPT-4 Turbo	Claude-3 Sonnet	Claude-3 Opus	Gemini 1.0-pro	GPT-3.5 Turbo	GPT-4 Turbo	Claude-3 Sonnet	Claude-3 Opus	OOD Avg.-CM	OOD Avg.-CD	OOD Avg.
BiScope	0.9665	**0.9655**	**0.8528**	0.6069	**0.7809**	**0.9659**	**0.9464**	**0.9691**	**0.9250**	0.7974	0.5895	0.8999
BiScope*	**0.9692**	0.9586	0.8526	0.6620	0.7741	0.9597	0.9435	0.9600	0.9222	0.7898	0.5855	**0.9024**

Figure 4.2: BiScope official code result from BiScope paper

Unlike methods that focus only on next-token prediction, BISCOPE evaluates text along two axes: next-token prediction and previous-token memorization.

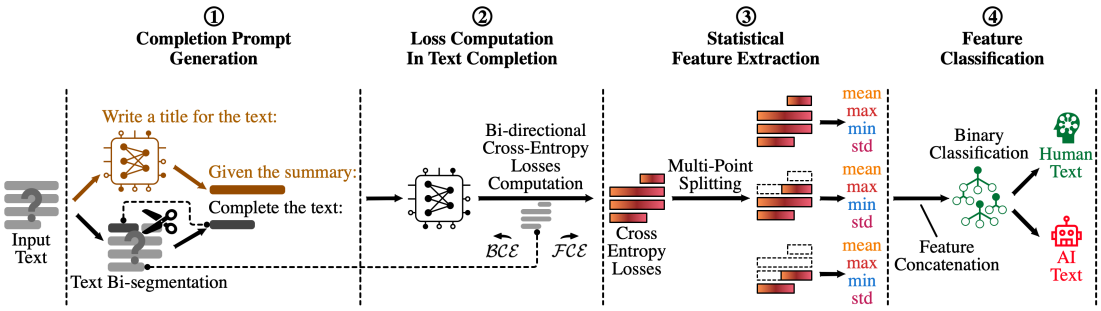


Figure 4.3: BiScope overview

1. **Completion-prompt construction:** an LLM first summarizes the text to obtain global context. The original text is then split in two (prefix and suffix). A completion request with the summary of the text and the first part (prefix).

2. **Loss computation:** Using the prompt, the surrogate model processes the suffix text and computes two per-token cross-entropy losses:

   (a) *Forward Cross-Entropy (FCE)*: the cross-entropy loss between the ground truth (suffix) next token and the LLM next token prediction given the

previous tokens.

$$\mathrm{FCE}_i \;=\; -\sum_{z=1}^{\|V\|} \tilde{P}_{i+1}^{z}\, \log\Big(P_i^{z}\Big),$$

(b) *Backward Cross-Entropy (BCE)*: the cross-entropy loss between the ground truth (suffix) last token and the LLM next token prediction given the previous tokens (last ground truth token included).

$$\mathrm{BCE}_i \;=\; -\sum_{z=1}^{\|V\|} \tilde{P}_{i}^{z}\, \log\Big(P_i^{z}\Big).$$

- V: as vocabulary; $\|V\|$ is its size.
- i: as current position in the *suffix* given the prompt+prefix (FCE valid only if i+1 exists).
- z: as index over vocabulary items $1, \ldots, \|V\|$.
- P_i^z: as softmax probability assigned by the LLM at position i to token z given all preceding tokens up to i.
- $\tilde{P}_i^z$: as one-hot encoding of the ground-truth token at position i ($\tilde{P}_i^{z^\star}$=1 if z=$z^\star$, else 0).

3. **Feature extraction:** The text is divided into segments. For each segment, statistics of FCE and BCE are collected: mean, max, min, and standard deviation.

4. **Feature classification:** All statistics are concatenated into a single feature vector used to train a binary classifier that decides whether the text is human- or AI-written *(implemented with a Random Forest)*.

Analysing the method, it is seen that many variables exist, such as: which LLM to use for completion, into how many sections to split the text, whether to give more or less importance to FCE and BCE. All these variables were not specified in the CodeMirage paper. For this reason, each hyperparameter was set to its default value.

BiScope-improved

This method is proposed for the first time in this thesis and attempts to modify BiScope specifically for code. The big changes are: erase the phase of summarizing the code and the LLM used in FCE and BCE computation.

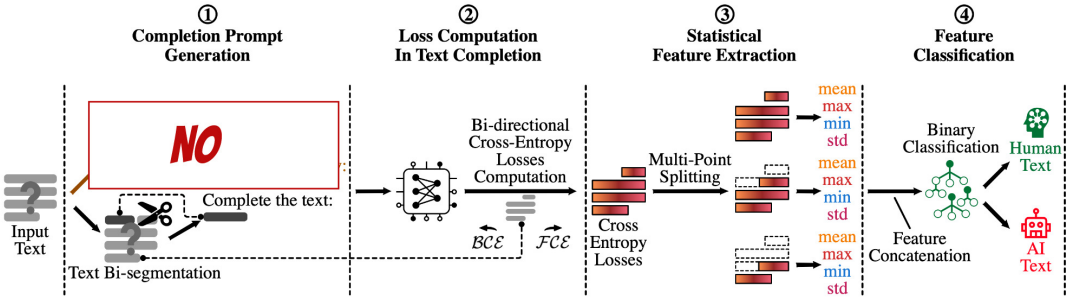


Figure 4.4: BiScope overview improved

The **phase of summarizing** is real useful for natural text, like chose a title for the text (another type of prompt recommended by the authors). The problem is that more or less every LLM can generate a good summary of natural text, but non every LLM-code-finetuned can generate a good summary (or explanation) of code. For this reason in the BiScope-improved version the summarization phase was removed, and we can demonstrate that it is not necessary for achieve good results. Erasing the summarization phase also has the advantage of speeding up the inference phase, which is already quite slow due to the need to perform double cross-entropy computation (BCE and FCE).

Regarding the **choice of the LLM**, by analysing the LLMs proposed by BISCOPE (and probably also used in the CodeMirage tests) on which to evaluate the loss values, it was evident that none of those LLMs had been fine-tuned on code:

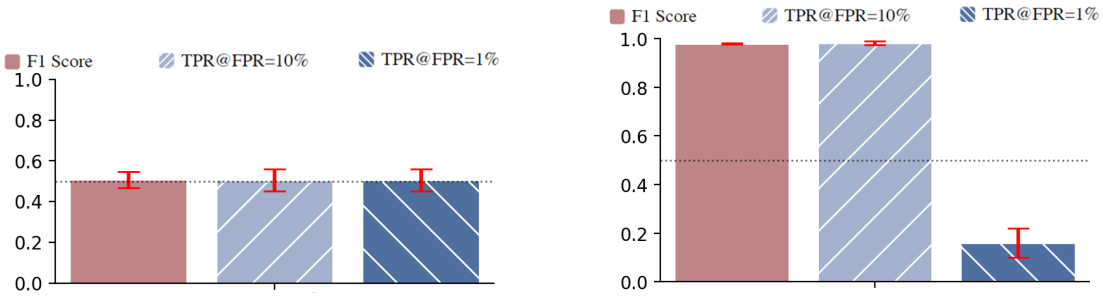
LLM proposed	**Code finetuning?**
Llama-2-7B[6],**Llama-2-13B**[6], **Llama-3-8B**[93]	No
Gemma-2B[94], **Gemma-7B**[94]	Code partly present in the training dataset
Mistral-7B[8]	Code partly present in the training dataset

Table 4.1: Model Comparison and Training on Code

For this reason, a retraining of the classification head was performed using **CodeLlama-7b-hf** [12] as the LLM. It was chosen as the LLM because: it is not among the LLMs that generated code in the CodeMirage dataset; it has been

fine-tuned on code in several languages (Python, C++, Java, PHP, TypeScript, JavaScript, C#); it has been trained to respond to completion requests (required by BiScope); and it is an instruct version, as expected by the BiScope framework.

To evaluate the improvements introduced by changing the LLM, in-domain tests were conducted on the CodeMirage dataset.



(a) BiScope standard.

(b) BiScope CodeLlama-7b-hf.

Figure 4.5: BiScope and BiScope improved CodeMirage in-domain test.

It is evident that the results are more than excellent. Inspecting the plot, it appears that with an FPR of 10% the model attains an almost perfect TPR. It can also be noted that at FPR = 1% the TPR decreases by a non-negligible amount. This means that, under these conditions, the model is usable only by accepting an error probability of about 10% or slightly less.

GPTSniffer-CodeT5+ multilingual

This method is based on another work, GPTSniffer[87], with the only difference of using a different encoder-only model. This also shows how the same methods can obtain extremely different results (see the TPR results at FPR = 1% reported by CodeMirage 4.1) without changing the methodological aspect but only by changing the transformer used. This should not be surprising, since much of the work is carried out by these networks, but it also shows how the results of the same method can vary greatly when a different transformer is used.

The GPTSniffer technique is based on two principles:

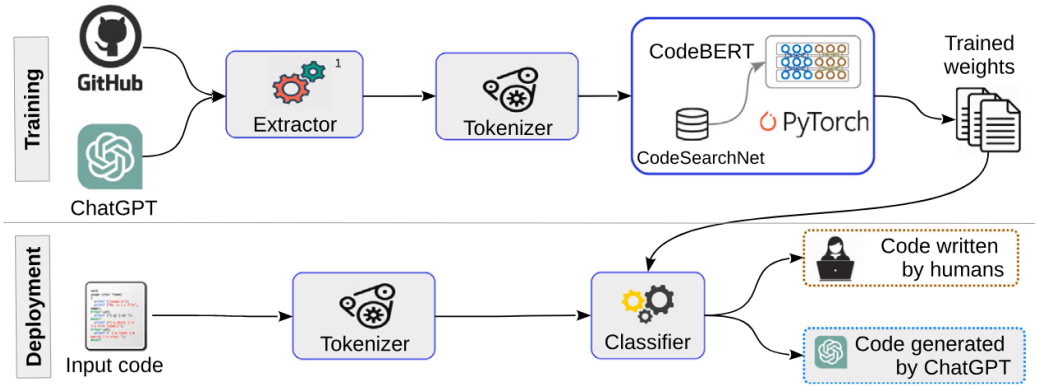


Figure 4.6: GPTSniffer system

1. Train the transformer and the classification head on code style, not on the code itself. They seek to obtain this result by applying heavy preprocessing that:

   (a) Removal of all imports

   (b) Removal of comments

   (c) Removal of formatting characters

   (d) Replacement of class name

2. Fine-tune the encoder-only module to obtain the best embedding representation to be classified by the head.

Unlike CodeMirage, to carry out these tests a **preprocessing version applicable to many programming languages** (all those present in the CodeMirage dataset) was developed. In fact, GPTSniffer focuses only on Python, which would be a problem both during training and during testing on a multilingual dataset.

To evaluate the improvements introduced by changing the LLM, in-domain tests were conducted on the CodeMirage dataset.

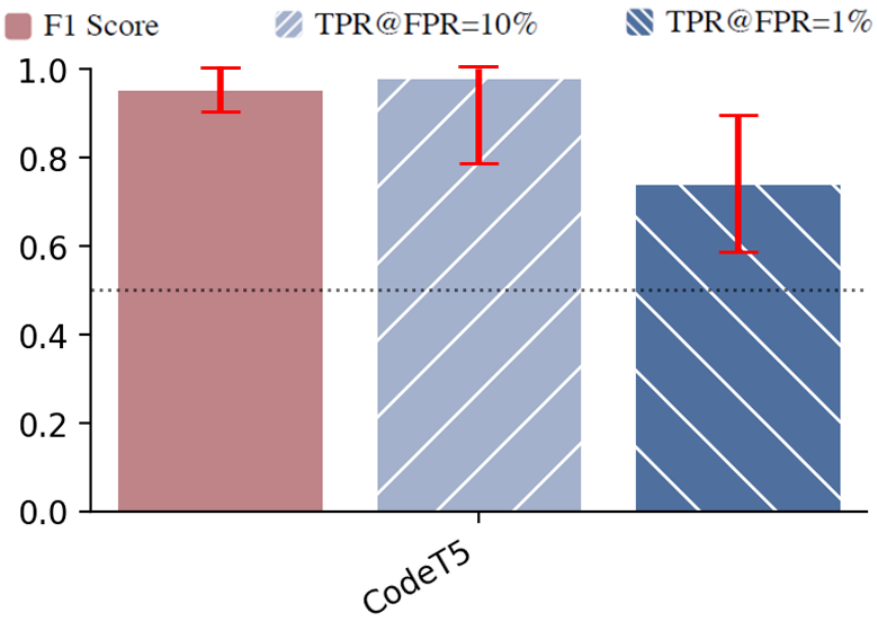


Figure 4.7: CodeT5 tests with multilingual preprocessing CodeMirage in-domain test.

Excellent performance is observed here as well. Compared with BiScope with CodeLlama, performance across languages remains much more variable (higher standard deviation). Nevertheless, very good TPR is obtained at FPR = 1%, a result that is interesting and useful for scenarios where avoiding unjust accusations is required.

UncoveringLLM

This method was one of the first proposed for detection specifically on code. After recognizing that the problem on code was profoundly more complex than on human code, a new technique was implemented. Unfortunately, it should be noted that the provided repository was wholly insufficient. For this reason, the entire technique they proposed was reproduced from scratch. The technique is based on a key observation: when an LLM rewrites code it previously generated, the differences between the two versions are minimal. Conversely, if the LLM rewrites code written by a human, the differences are much more significant. Leveraging this intuition, the detector operates in three fundamental steps:

1. **Rewrite the code:** An LLM is used to rewrite the code snippet under analysis. The model is instructed to explain the code's functionality and rewrite it, a method based on chain-of-thought prompting.

2. **Measure similarity:** A code-similarity model, such as GraphCodeBERT[95] (trained with self-supervised contrastive learning), computes a similarity

score between the original code and its rewritten version. The framework is flexible and not tied to a specific similarity model.

3. **Compute the detection score:** The rewriting process is repeated multiple times (m times), and the final similarity score is the mean of the scores from each rewrite. If the score exceeds a threshold, the code is classified as LLM-generated. According to experiments, only four rewrites are sufficient to achieve excellent results.

The method, although interesting, presents a major problem in use: at least two rewrites are required for each code snippet, greatly lengthening both the training and inference phases. For this reason, despite the work recommending 4 rewrites for higher performance, the tests will use two rewrites, which according to the original paper's tests does not reduce the F1-score by more than 0.1.

To use this method, the previously described framework 3.3.1 was therefore used (with slight modifications) to generate code rewrites on a dataset. In this way it was possible to run tests on the various available datasets.

It is important to note that, since CodeBERT was not retrained for the tests conducted on this method, the model used to rewrite the code is one of those adopted during CodeBERT's training (CodeLlama-13-Instruct[12]).

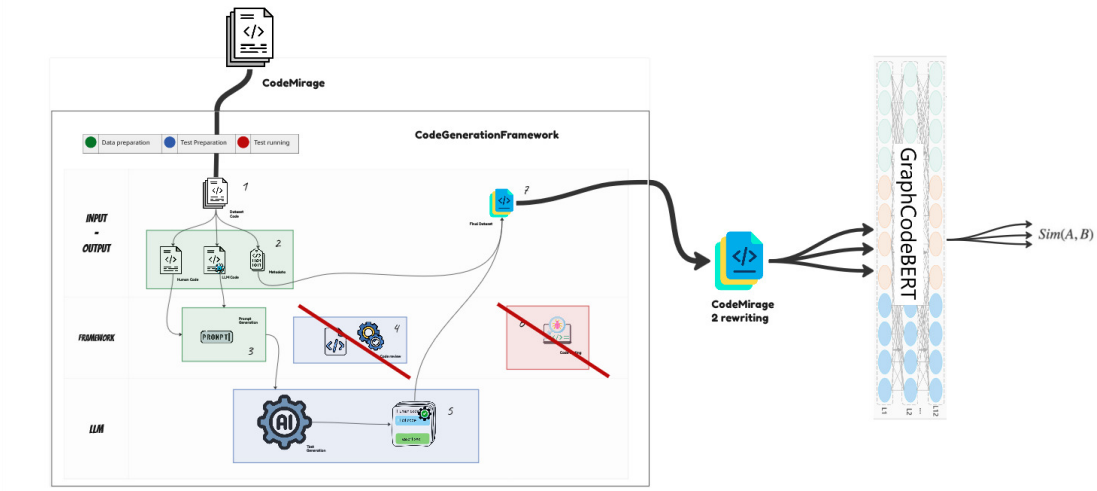


Figure 4.8: UncoveringLLM Framework

LLMPPL

The work "Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models" proposes a zero-shot detection method based on perplexity. However, from their official repository on GitHub it can be seen that only the code for computing perplexity was released, and not the code for all the subsequent evaluations described in their work. Nevertheless, it is appropriate to test what was implemented. In this work, perplexity is computed using different techniques,

including the use of a **Mamba model**. In particular, they implemented the calculation of perplexity with mamba-130m-hf The use of Mamba for perplexity calculation is very interesting, since it is known that Mamba models tend to be much more efficient than transformers (linear-time computation rather than exponential).

Quickly repeating how the perplexity score is computed in general:

1. **Tokenization**
   A text sequence is converted into token IDs, where s is the text:

$$T(s) = (x_1, \dots x_L), \quad x_i \in V = 1, \dots, n$$

2. **Autoregressive prediction**
   The model produces logits for each position i given the prefix, where $|V|$ is the vocabulary size and f_M is the LLM *(or in this case the mamba model)*:

$$Z_i = f_M(x_{0:i-1}), \quad Z_i \in \mathbb{R}^{|V|}$$

3. **Token probabilities**
   Apply the softmax function to obtain probabilities:

$$Y_i = \text{softmax}(Z_i), \quad Y_{ij} = P(v_j \mid x_{0:i-1}) = \frac{\exp(Z_{ij})}{\sum_{u \in V} \exp(Z_{iu})}.$$

4. **Shifted targets**
   For causal LM, the target at step i is the next token in the sequence:

$$\text{target at } i \text{ is } x_i \text{ given } x_{0:i-1}, \quad i = 1, \dots, L.$$

5. **Negative log-likelihood per token**

$$\text{NLL}_i = -\log Y_{i\,x_i}.$$

6. **Average loss (cross-entropy in nats)**

$$\text{loss} = \frac{1}{L} \sum_{i=1}^{L} \text{NLL}_i = -\frac{1}{L} \sum_{i=1}^{L} \log Y_{i\,x_i} = \log \text{PPL}_M(s).$$

7. **Perplexity**
   So more the perplexity is low, more the text is likely to be generated by an LLM:

$$\text{PPL}_M(s) = \exp(\text{loss}).$$

In order to **select the best perplexity threshold** in the tests, the training dataset was used to find the threshold that guaranteed an FPR = 10%.

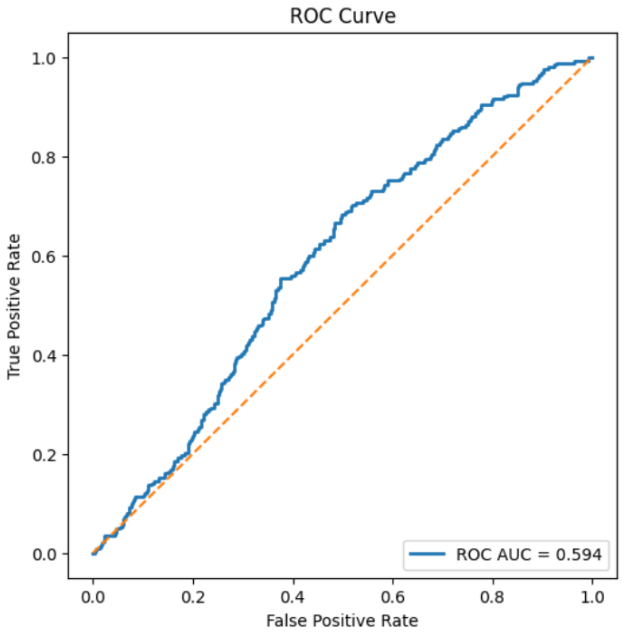


Figure 4.9: LLMPPL ROC over codemirage dataset

Binoculars

Classical detection methods rely mainly on PPL calculation. Binoculars takes a step further, pointing out that a text may have low PPL even if generated by an LLM with a particular prompt. It is well known that there is an entire field called prompt engineering that focuses precisely on obtaining certain results from an LLM by simply changing the prompt. Since the prompt is so fundamental during generation, it cannot be ignored in PPL calculation. However, it is obviously not possible to recover the prompt given only the generated text (in this case, the code). For this reason, Binoculars proposes evaluating the value B, given by the ratio between the PPL computed by a first LLM and the PPL of a second LLM on the predictions of the first LLM.

PPL over first LLM:

$$\log \mathrm{PPL}_{M_1}(s) = -\frac{1}{L}\sum_{i=1}^{L} \log p_{M_1}(x_i \mid x_{<i}).$$

PPL over second LLM of the predictions of the first LLM:

$$\log \mathrm{xPPL}_{M_1,M_2}(s) = -\frac{1}{L}\sum_{i=1}^{L}\sum_{v\in V} p_{M_2}(v \mid x_{<i}) \log p_{M_1}(v \mid x_{<i}).$$

Ratio:

$$B_{M_1,M_2}(s) = \frac{\log \mathrm{PPL}_{M_1}(s)}{\log \mathrm{xPPL}_{M_1,M_2}(s)}.$$

Deduction:

- LLM-like text: $\mathrm{PPL}_{M_1}(s) \approx \mathrm{xPPL}_{M_1,M_2}(s) \;\Rightarrow\; B$ small.
- Human-like text: $\mathrm{PPL}_{M_1}(s) \gg \mathrm{xPPL}_{M_1,M_2}(s) \;\Rightarrow\; B$ large.
- Assumption needed: M_1 and M_2 are similar (same tokenizer and comparable strength).

The calculation of PPL by the second LLM on the predictions of the first is intended to "normalize" the perplexity value due to the particularity of the generated text (in this case code) and thus the particularity of the prompt. It is easy to see that this issue is more evident in natural-language texts; nevertheless, this method was also tested using LLMs fine-tuned on code, in particular meta-llama/CodeLlama-7b-hf as the first LLM and CodeLlama-7b-Instruct-hf [12] as the second LLM. Tests were also conducted with the LLMs used in the original work: falcon-7b and falcon-7b-instruct [96]

For the standard method, the thresholds recommended by the authors of Binoculars were used, whereas for the version with different LLMs the same criterion used for LLMPPL was applied.

4.3.3 Tests

CodeMirage tests

Since the goal of this section is to test the different detection methods, all models had to be retrained on the same dataset. To this end, the dataset used by UncoveringLLM [58] to train its method was obtained. This dataset includes 6000 code samples, of which 1500 are human and 4500 are generated by: CodeLlama-13B-Instruct [12], StarChat-Alpha [97], and GPT-3.5-Turbo. Unfortunately, the dataset does not indicate which LLM generated each code sample, so it was not possible to balance the different LLMs during training. The UncoveringLLM model was not retrained, since its training dataset was used during the tests. Let recall the utilities behind CodeMirage [51] test:

- **out-domain tests:** the model is trained and tested on different datasets (the training set is collect by competitive programming datasets, while the test set is collect by github).
- **LLM generalization tests:** The test set includes 10 LLMs, while the training set only 3.
- **multilingual generalization:** the test set includes code in 7 different programming languages, the training set contains only python code.

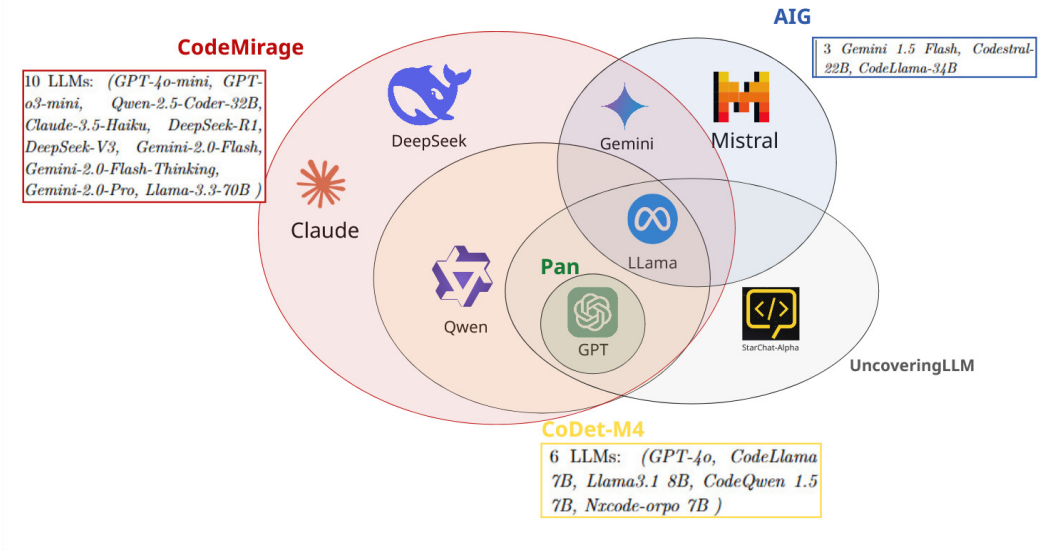
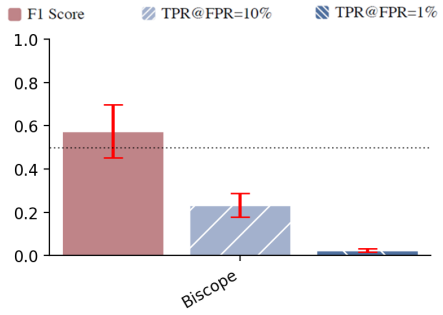


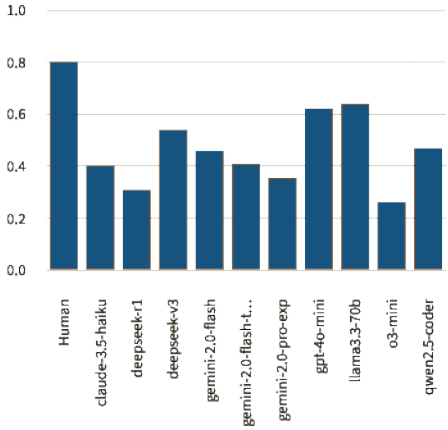
Figure 4.10: The different LLMs familys inside the datasets.

For this purpose, not only the metrics chosen as relevant for selecting the best method will be evaluated, but also the accuracies on the different LLMs or languages will be shown, as useful for assessing model generalization. The idea behind this evaluation is to determine whether a method tends to learn

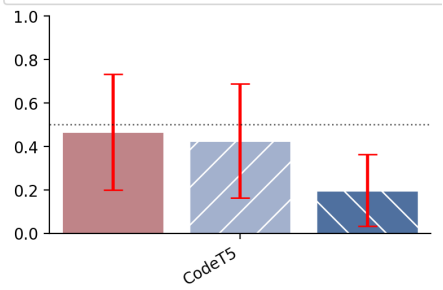
the "programming style" of a specific LLM (thus requiring training on every available LLM) or whether **it is able to capture features generalizable to all transformer-generated code** (assuming such features exist).



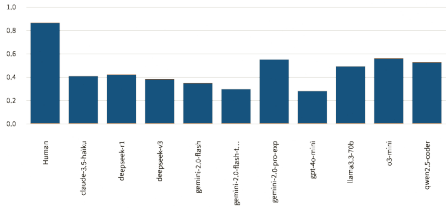
(a) BiScope-improved test Codemirage



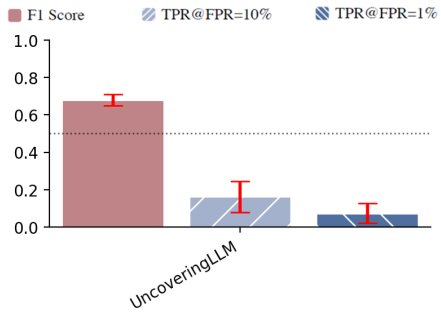
(b) BiScope-improved test Codemirage accuracy at FPR=10%



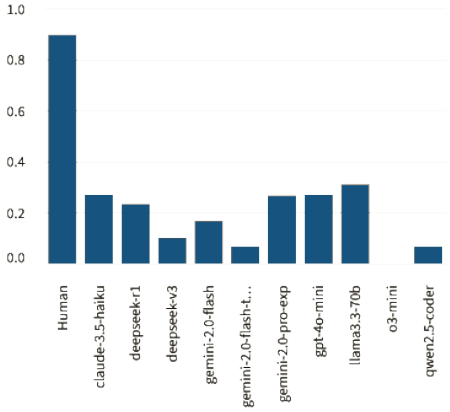
(a) CodeT5+ test Codemirage



(b) CodeT5+ test Codemirage accuracy at FPR=10%



(a) UncovernngLLM test Codemirage



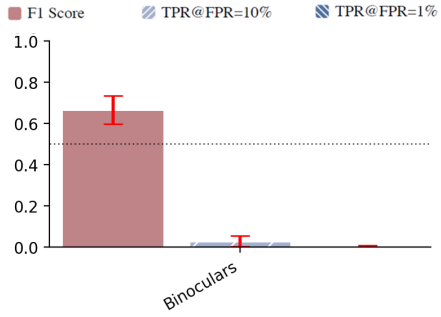
(b) UncovernngLLM test Codemirage accuracy at FPR=10%

Better results (BiScope-improved, CodeT5+, UncoveringLLM): Let's start by analyzing the most evident result. While all methods achieve poor performance (a result that was predictable given the training dataset), BiScope-improved appears to be the method that achieves the best results. This suggests that perhaps the forward and backward approach seems to be the most generalizable across LLMs and programming languages.

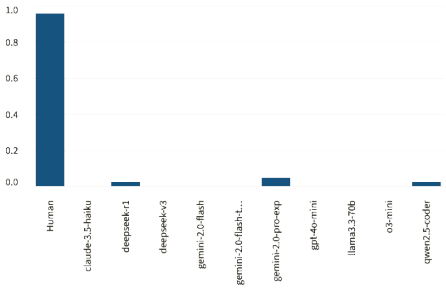
The evaluation of accuracy across different programming languages was not reported because it does not seem to yield any significant value. On the other hand, by analyzing the accuracy of different detection methods across various LLM codes, the following observations can be made:

The differences in accuracy between LLMs across the different methods are not proportional. Taking the example of o3-mini (a model not present in the dataset but represented by GPT-3.5), we can see that UncoveringLLM reports an accuracy of 0%, while GPTSniffer, CodeT5+, and BiScope-improved report an accuracy around 0.3%. Although lower, this shows a greater ability of these methods to identify general features of LLM generation.

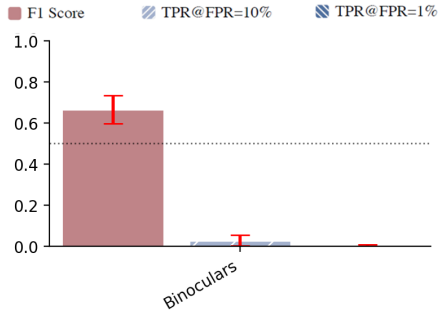
Additionally, it is observed that the accuracy reported by GPTSniffer and CodeT5+ tends to be more or less uniform across different LLMs, whereas methods like UncoveringLLM and BiScope report better results on the same LLMs. As expected, the best results tend to be associated with the LLMs present in the training dataset. However, it is worth noting that BiScope achieves positive accuracy results even for LLMs not present in the dataset, such as DeepSeek and Qwen.



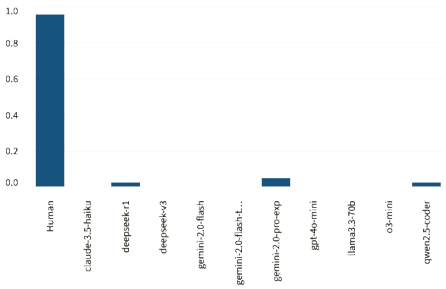
(a) Binacularis standard test Codemirage



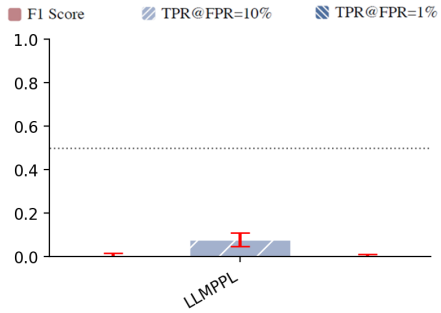
(b) Binacularis standard test Codemirage accuracy at FPR=10%



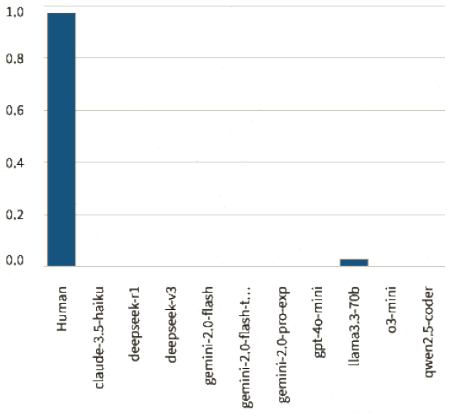
(a) Binacularis test Codemirage



(b) Binacularis test Codemirage accuracy at FPR=10%



(a) LLMPPL test Codemirage



(b) LLMPPL test Codemirage accuracy at FPR=10%

Worse results (Binoculars, LLMPPL): Furthermore, it is evident that Binoculars, whether with LLMs fine-tuned on code or the LLMs proposed by

the authors, achieves poor and nearly identical performance. Similar results are obtained with LLMPPL. These results are consistent with the previous considerations: a simple perplexity calculation on code (even if normalized on the prompt as proposed by Binoculars) is not sufficient for good code detection results. For this reason, LLMPPL and Binoculars will no longer be considered valid methods for code detection.

AIGCode tests

Rather than stopping at the initial results, it is important to conduct a broader analysis, as proposed in the previous sections. First, the datasets with different types of code (competitive code) were analysed.

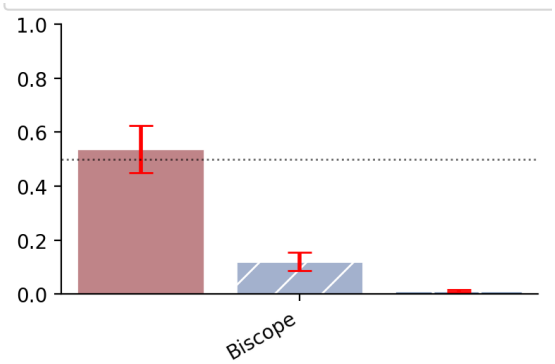


Figure 4.17: BiScope test AIGCode

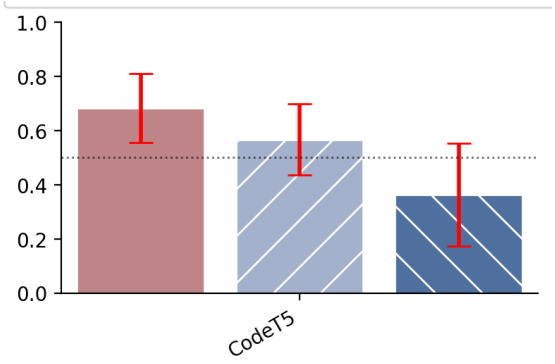


Figure 4.18: CodeT5 test AIGCode

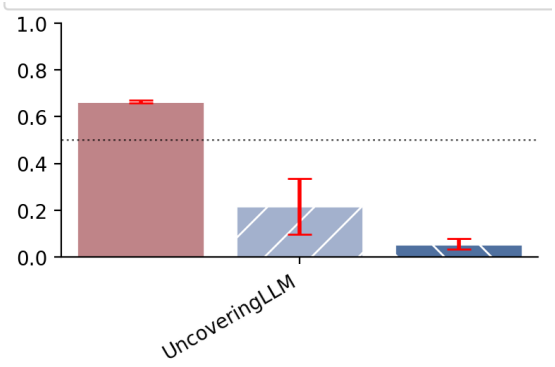


Figure 4.19: UncovernngLLM test AIGCode

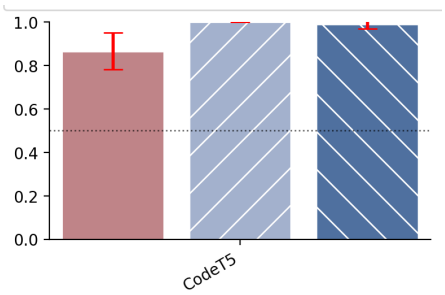
It can be noted that the only method providing a positive TPR@FPR=10% is CodeT5. This is likely due to the fact that competitive code tends to be extremely short, making it very difficult to capture relevant metrics in a few lines of code without comments, unless through a pre-trained transformer encoder-only model.

Additionally, it is significant to note that the data on which the models were trained is extremely limited due to the small size of AIGCodeste.

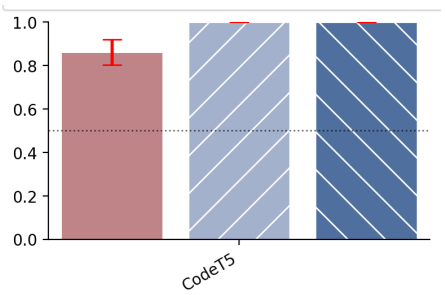
Since CodeT5+ is trained to "understand" code, it achieves better results than other methods, even for small code snippets. However, the results are still not particularly positive, even for CodeT5+. For this reason, a decision was made: to use the most promising method (BiScope), train it on more available datasets, and use Pan as the test set. Therefore, we give up the ability to detect small code snippets but aim to achieve better results with larger code samples.

Pan tests

In the end, it is useful to perform one final test: evaluating whether the detection method's results tend to differ depending on whether the code is functional or not. The reason behind this test is that if the method appears to disproportionately detect non-functional code, it could indicate that there was a correlation between functional and non-functional code in the training datasets, and that CodeT5 might have learned to detect errors in non-functional code rather than understanding the intrinsic characteristics of LLM-generated code.



(a) COdetT5+ Pan good code



(b) COdetT5+ Pan bad code

Given the tests, there does not appear to be any significant difference.

4.3.4 Proposed Improvements

Given the tests, it seems clear that the method capable of achieving the best TPR@FPR=10% is GPTSniffer/CodeT5+. However, it cannot be denied that the results are not satisfactory. It is evident that the detection method cannot be considered usable in a real-world environment. Nevertheless, it is also the most promising method, and for this reason, GPTSniffer/CodeT5+ will be the method proposed for implementation. Before doing so, some final tests and training phases were conducted: first, the entire CodeMirage dataset was used for training, and to make this training feasible, the Lora method was applied.

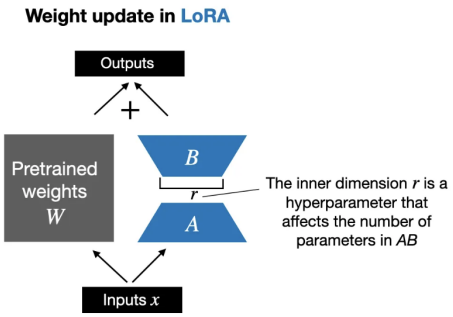


Figure 4.21: Lora learning method

LoRA operates by freezing the pre-trained model weights and injecting two low-rank matrices, $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times k}$, into each layer of the transformer model. The weight update is approximated as:

$$\Delta W = A \cdot B$$

where r is the rank of the decomposition, and d and k are the dimensions of the original weight matrix. This approach significantly reduces the number of trainable parameters, as $r \ll \min(d, k)$. By introducing low-rank matrices, LoRA reduces the number of trainable parameters, leading to lower memory usage and faster training times. The low-rank approximation may not capture all the complexities of the task, potentially limiting performance. The Lora configuration used are: r=8 and α=16

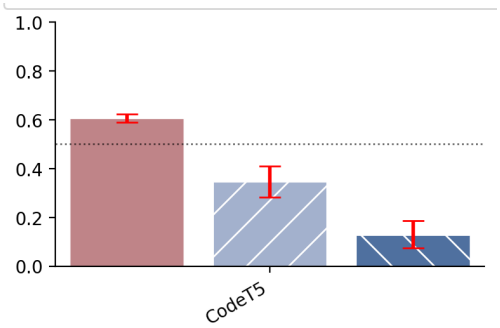
Anyway, the tests conducted between the model trained with Lora and the model trained on a smaller sub-dataset without Lora lead to the preference for the model without Lora.

A test was also conducted on inference quantization, and in that case, the TPR@FPR=10% performance was halved. For this reason, quantization is not recommended.

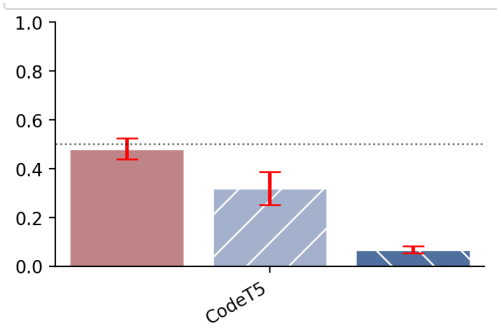
Quantization in Transformers means storing weights, activations, and the KV-cache with fewer bits to cut memory and bandwidth while keeping accuracy within acceptable limits.

At its core you map real numbers to integers using a scale (and sometimes a zero-point). During inference you multiply the small integers by the scale to recover approximate real values. This is cheap, and it lets the heavy matrix multiplies run on fast low-precision kernels.

Why it matters for Transformers: these models are bandwidth bound. Self-attention and MLP blocks move a lot of data, and the KV-cache grows linearly with sequence length. Dropping from FP16 to INT8 or INT4 often shifts the bottleneck and yields higher throughput and lower memory use without retraining.



(a) CodeT5 test AIGCode



(b) AIGCode Test with Lora

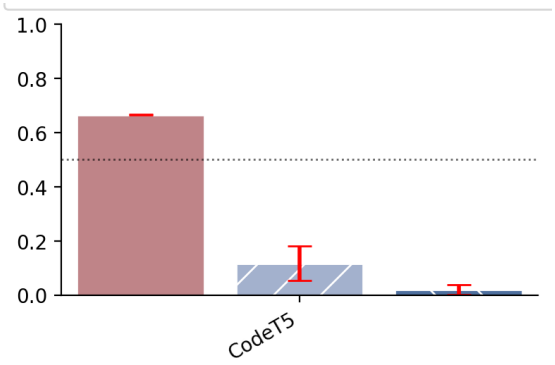


Figure 4.23: quant 8-bit test

Chapter 5

User Interface

The user interface was initially designed to resemble any typical detector interface, such as GPTZero. The original idea was simply to provide a text box where the code to be analysed could be inserted, along with a graph to visualize the probability that the code was generated by an LLM. However, during the evaluation of the methods, it became clear that an effective code detection method does not currently exist. For this reason, the project evolved to not only allow a user to evaluate a single piece of code, as originally intended, but also to enable the retrieval of datasets and the testing of methods presented in this work. The motivation behind this decision was to avoid repeating the mistakes of some published works in this field and to make it extremely simple for any user to test and replicate the results obtained.

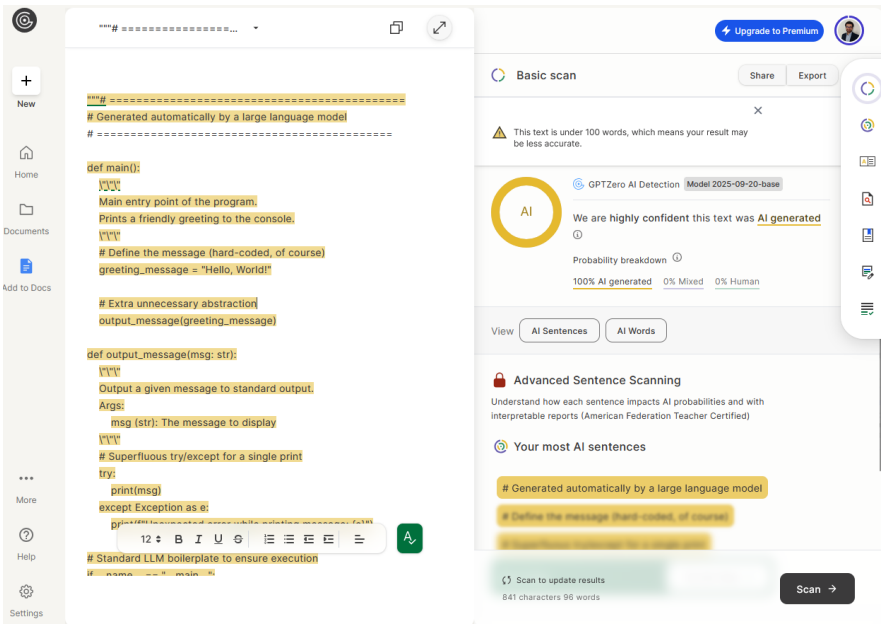


Figure 5.1: gptzero interface

5.1 Requirements Analysis

Unfortunately, it was not possible to distribute questionnaires regarding the requirements for a code detection interface. However, it wasn't difficult to find successful portals to draw inspiration from. Methods for detecting natural text on the web tend to have a clean interface, display a probability percentage of detection, and show the sections of the text that most likely indicate that the text is generated by an LLM. This last request was not feasible due to the different detection method used for code (code embedding does not allow us to understand which sections of the code lead to the conclusion that it was generated by an LLM).

A requirement set during development was the need to create a web-based interface. In this way, it would not be complicated, if desired, to make the code detector easily accessible to any user via the web by setting up an appropriately equipped server. Given the request to make this interface not only a simple detector but also a validator of the methods presented and datasets proposed, it was crucial to make the detection interface easily and immediately accessible, clearly separated from other sections.

Given the large number of datasets and methods presented, it was necessary to visually differentiate each dedicated page, using colors (for datasets) and different page layouts for the detection methods. In fact, each detection method had to provide several possible operations, such as: training a model or loading the weights of a previously trained model, easily launching training, allowing methods based on perplexity to calculate the best threshold or set it manually before launching any test. Additionally, for each dataset, an automated dataset balancing method, an automatic split for training, validation, and test sets, and a standard format for each dataset had to be provided.

5.2 Design Choices

It was decided to design the text box to recognize programming language patterns and allow the code to be displayed in a user-friendly manner according to the programming language. For this reason, a section was added to select the language of the code. The language selection does not affect the detection in any way (at least in the current method). After entering the code on the right, after a brief loading period, the probability that the code was generated by an LLM or written by a human will be displayed. The percentage bar changes colour depending on the percentage. On the same page, the reasons why an LLM code detector is useful are quickly listed.

In the dataset section, every dataset available from Hugging Face is automatically downloaded. The purposes for which the dataset is recommended for use are provided (for example, AIG is recommended for testing the detection method's ability to identify competitive code).

Users are also given the option to decide the size of the subset they wish to obtain from the dataset. A balanced subset is then automatically created based on: LLM generators, programming languages, and code correctness. When these fields are not present, they are ignored. Additionally, a graph is displayed to visually represent the average length of code without comments.

Chapter 6

Conclusion

Given the results of the various tests, we can assert that the problem of detecting code generated by LLMs is still not solved. The methods proposed in the literature are few, and even fewer are available and actually testable. Among the proposed methods, even the best can only achieve a TPR@FPR=10% that is not acceptable in a real-world context. However, this work can be an important milestone, clearly demonstrating that:

1. It is essential to conduct tests only on code without comments to avoid contaminating the results with the method's ability to detect natural text.

2. To develop a generally effective code detection method, it is necessary to rely on code encoding models, and it is not sufficient to use methods based solely on perplexity.

6.1 Final Evaluation and Perceived Quality

6.2 Future Work

There are many other methods proposed in the literature that could not be tested because their work was not available. It would be worth considering the possibility of retrieving these implementations or developing and testing them. Additionally, it would be a good initiative to make the interface more easily modifiable by any user, making it simple for anyone to add datasets or detection methods. The goal would be to encourage the community to make detection code publicly available. Perhaps this interface could be hosted on a platform to allow any user to perform detection tasks without the need for local hardware capable of running the recommended detection method.

It would be an interesting task to extend the CodeMirage work by adding competitive code as well. This would be useful because this thesis suggests that the most solid methods rely on transformer-encoders fine-tuned for the detection

task. Therefore, it is clear that having a large dataset with all types of code, languages, and as many LLMs as possible can only benefit any kind of training.

More variants of those tested in this work could be explored, such as using a more complex classification head for both BiScope and GPTSniffer/CodeT5+. Detection methods based on machine learning could also be tested with different techniques, hyperparameters, optimizers, etc.

Bibliography

[1] Kevin Warwick and Huma Shah. "Can Machines Think? A Report on Turing Test Experiments at the Royal Society". In: *Journal of Experimental & Theoretical Artificial Intelligence* 28.6 (2016), pp. 989–1007.

[2] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[3] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[4] Josh Achiam et al. *Gpt-4 technical report*. 2023.

[5] Aakanksha Chowdhery et al. "Palm: Scaling language modeling with pathways". In: *Journal of Machine Learning Research* 24.240 (2023), pp. 1–113.

[6] Hugo Touvron et al. "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023).

[7] Anthropic. *Claude: The Anthropic Language Model*. https://www.anthropic.com/index/introducing-claude. 2023.

[8] Albert Q. Jiang et al. "Mistral 7B". In: (2023). arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825.

[9] Robin Rombach et al. "High-resolution image synthesis with latent diffusion models". In: (2022), pp. 10684–10695.

[10] Google DeepMind. *Veo: Advancing Generative Video Models*. https://deepmind.google/technologies/veo. 2024.

[11] Mark Chen et al. "Evaluating large language models trained on code". In: 2021.

[12] Baptiste Roziere and et al. "Code LLaMA: Open Foundation Models for Code". In: *arXiv preprint arXiv:2308.12950* (2023).

[13] Joseph Weizenbaum. "ELIZA—a computer program for the study of natural language communication between man and machine". In: *Communications of the ACM* 9.1 (1966), pp. 36–45.

[14] Tomas Mikolov. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* 3781 (2013).

[15] Jeffrey Pennington, Richard Socher, and Christopher D Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[16] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[17] Yue Wang et al. "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation". In: *arXiv preprint arXiv:2109.00859* (2021).

[18] Erik Nijkamp et al. "Codegen: An open large language model for code with multi-turn program synthesis". In: *arXiv preprint arXiv:2203.13474* (2022).

[19] Qinkai Zheng et al. "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x". In: (2023), pp. 5673–5684.

[20] John Backus and et al. "The FORTRAN automatic coding system". In: *Proceedings of the Western Joint Computer Conference* (1957), pp. 188–198.

[21] Richard M. Stallman. *EMACS: The extensible, customizable self-documenting display editor*. https://www.gnu.org/software/emacs/. 1981.

[22] W3C. *XSL Transformations (XSLT) Version 1.0*. https://www.w3.org/TR/xslt. 1999.

[23] John M. Zelle and Raymond J. Mooney. "Learning to parse database queries using inductive logic programming". In: *AAAI/IAAI, Vol. 2*. 1996, pp. 1050–1055.

[24] Raymond J. Mooney. "Learning semantic parsers: An important but understudied application of machine learning". In: *Proceedings of the AAAI Spring Symposium on Natural Language Processing for the World Wide Web*. 1997.

[25] Pallets Projects. *Jinja2 Documentation*. https://jinja.palletsprojects.com/. 2005.

[26] Mike Bayer. *Mako Templates for Python*. https://www.makotemplates.org/. 2006.

[27] Pengcheng Yin and Graham Neubig. "A syntactic neural model for general-purpose code generation". In: *ACL*. 2017.

[28] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: (2019), pp. 4171–4186.

[29] Alec Radford et al. "Improving language understanding by generative pre-training". In: (2018).

[30] TabNine. *TabNine: Autocomplete AI*. https://www.tabnine.com/. 2019.

[31] Hamel Husain et al. “Codesearchnet challenge: Evaluating the state of semantic code search”. In: *arXiv preprint arXiv:1909.09436* (2019).

[32] Zhangyin Feng et al. “Codebert: A pre-trained model for programming and natural languages”. In: *arXiv preprint arXiv:2002.08155* (2020).

[33] Jacob Austin et al. “Program synthesis with large language models”. In: *arXiv preprint arXiv:2108.07732* (2021).

[34] Dan Hendrycks et al. “Measuring coding challenge competence with apps”. In: *arXiv preprint arXiv:2105.09938* (2021).

[35] Shuai Lu et al. “Codexglue: A machine learning benchmark dataset for code understanding and generation”. In: *arXiv preprint arXiv:2102.04664* (2021).

[36] Yujia Li et al. “Competition-level code generation with alphacode”. In: *Science* 378.6624 (2022), pp. 1092–1097.

[37] Frank F Xu et al. “A systematic evaluation of large language models of code”. In: *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*. 2022, pp. 1–10.

[38] Google DeepMind. “Gemini: A Family of Highly Capable Multimodal Models”. In: (2023).

[39] Binyuan Hui et al. “Qwen2. 5-coder technical report”. In: *arXiv preprint arXiv:2409.12186* (2024).

[40] Gregor Jošt, Viktor Taneski, and Sašo Karakatič. “The Impact of Large Language Models on Programming Education and Student Learning Outcomes”. In: *Applied Sciences* 14.10 (2024). DOI: 10.3390/app14104115. URL: https://www.mdpi.com/2076-3417/14/10/4115.

[41] Neil Perry et al. “Do users write more insecure code with AI assistants?” In: *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*. 2023, pp. 2785–2799.

[42] *J. Doe 1, et al. v. GitHub, Inc., et al.* Order Granting in Part and Denying in Part Motion to Dismiss, ECF No. 195. Jan. 3, 2024. URL: https://law.justia.com/cases/federal/district-courts/california/candce/4:2022cv06823/403220/195/ (visited on 05/21/2024).

[43] Ilia Shumailov et al. “The curse of recursion: Training on generated data makes models forget”. In: *arXiv preprint arXiv:2305.17493* (2023).

[44] Eric Mitchell et al. “Detectgpt: Zero-shot machine-generated text detection using probability curvature”. In: (2023), pp. 24950–24962.

[45] Edward Tian, Alexander Cui, and the GPTZero Team. *GPTZero’s AI Detection Technology*. Tech. rep. GPTZero, Inc., 2023. URL: https://gptzero.me/technology.

[46] Tong Ye et al. “Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting”. In: 39.1 (2025), pp. 968–976.

[47] Yuling Shi et al. “Between lines of code: Unraveling the distinct patterns of machine and human programmers”. In: *arXiv preprint arXiv:2401.06461* (2024).

[48] Daniil Orel, Dilshod Azizov, and Preslav Nakov. “CoDet-M4: Detecting Machine-Generated Code in Multi-Lingual, Multi-Generator and Multi-Domain Settings”. In: *arXiv preprint arXiv:2503.13733* (2025).

[49] Basak Demirok and Mucahid Kutlu. “AIGCodeSet: A New Annotated Dataset for AI Generated Code Detection”. In: *arXiv preprint arXiv:2412.16594* (2024).

[50] Hyunjae Suh et al. “An Empirical Study on Automatically Detecting AI-Generated Source Code: How Far Are We?” In: *arXiv preprint arXiv:2411.04299* (2024).

[51] Hanxi Guo et al. “CodeMirage: A Multi-Lingual Benchmark for Detecting AI-Generated and Paraphrased Source Code from Production-Level LLMs”. In: *arXiv preprint arXiv:2506.11059* (2025).

[52] Zhenyu Xu and Victor S Sheng. “Codevision: Detecting llm-generated code using 2d token probability maps and vision models”. In: *arXiv preprint arXiv:2501.03288* (2025).

[53] Tong Ye et al. “Uncovering llm-generated code: A zero-shot synthetic code detector via code rewriting”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 39. 1. 2025, pp. 968–976.

[54] Xiaodan Xu et al. “Distinguishing llm-generated from human-written code by contrastive learning”. In: *ACM Transactions on Software Engineering and Methodology* 34.4 (2025), pp. 1–31.

[55] Abhimanyu Hans et al. “Spotting llms with binoculars: Zero-shot detection of machine-generated text”. In: *arXiv preprint arXiv:2401.12070* (2024).

[56] Timothy Paek and Chilukuri Mohan. “Detection of LLM-Generated Java Code Using Discretized Nested Bigrams”. In: *International Conference on Computational Science and Computational Intelligence*. Springer. 2024, pp. 118–132.

[57] Sonnet Anthropic. “Model card addendum: Claude 3.5 haiku and upgraded claude 3.5 sonnet”. In: *URL https://api. semanticscholar. org/CorpusID* 273639283 (2024).

[58] Marc Oedingen et al. “Chatgpt code detection: Techniques for uncovering the source of code”. In: *arXiv preprint arXiv:2405.15512* (2024).

[59] Batu Guan et al. “Codeip: A grammar-guided multi-bit watermark for large language models of code”. In: *arXiv preprint arXiv:2404.15639* (2024).

[60] Jungin Kim, Shinwoo Park, and Yo-Sub Han. “Marking Code Without Breaking It: Code Watermarking for Detecting LLM-Generated Code”. In: *arXiv preprint arXiv:2502.18851* (2025).

[61] Ruisi Zhang et al. “Robust and secure code watermarking for large language models via ml/crypto codesign”. In: *arXiv preprint arXiv:2502.02068* (2025).

[62] Shinwoo Park et al. “Detection of llm-paraphrased code and identification of the responsible llm using coding style features”. In: *arXiv preprint arXiv:2502.17749* (2025).

[63] Yunhui Xia et al. “Leetcodedataset: A temporal dataset for robust evaluation and efficient training of code llms”. In: *arXiv preprint arXiv:2504.14655* (2025).

[64] Wei Hung Pan et al. “Assessing ai detectors in identifying ai-generated code: Implications for education”. In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*. 2024, pp. 1–11.

[65] OpenAI. *Hello GPT-4o*. https://openai.com/index/hello-gpt-4o/. Accesso effettuato il: 24-07-2024. May 2024.

[66] Meta AI. *Introducing Meta Llama 3.1: Our most capable, open models to date*. https://ai.meta.com/blog/meta-llama-3-1/. Accesso effettuato il: 24-07-2024. July 2024.

[67] Qwen Team. *CodeQwen1.5: An Open-source Large Language Model Series for Code*. https://qwenlm.github.io/blog/codeqwen1.5/. Accesso effettuato il: 24-07-2024. Feb. 2024.

[68] ntqa. *ntqa/NxCode-CQ-7B-orpo*. https://dataloop.ai/library/model/ntqai_nxcode-cq-7b-orpo/. Model page on the Dataloop AI Library. Updated on May 19, 2024. Accesso effettuato il: 24-07-2024. May 2024.

[69] Yeoh Yun Siang Geremie. *Codeforces Code Dataset*. Accessed on Kaggle. 2023. URL: https://www.kaggle.com/datasets/yeoyunsianggeremie/codeforces-code-dataset (visited on 07/07/2025).

[70] Gemini Team et al. “Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context”. In: *arXiv preprint arXiv:2403.05530* (2024).

[71] Mistral AI Team. *Codestral: First Open-Weight Generative AI Model for Code*. https://mistral.ai/news/codestral/. Released under Mistral AI Non-Production License; accessed 2025-07-09. May 2024.

[72] Ruchir Puri et al. “Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks”. In: *NeurIPS Datasets and Benchmarks*. arXiv:2105.12655. 2021. URL: https://arxiv.org/abs/2105.12655.

[73] OpenAI. *Introducing OpenAI o3-mini*. Includes benchmarks and system card; accessed 2025-07-09. Jan. 2025. URL: https://openai.com/index/openai-o3-mini/.

[74] Daya Guo et al. “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning”. In: *arXiv preprint arXiv:2501.12948* (2025).

[75] Aixin Liu et al. “Deepseek-v3 technical report”. In: *arXiv preprint arXiv:2412.19437* (2024).

[76] Google Developers Blog. *Gemini 2.0: Flash, Flash-Lite and Pro*. Accessed: 2025-07-09. May 2025. URL: https://developers.googleblog.com/en/gemini-2-family-expands/.

[77] Aaron Grattafiori et al. “The llama 3 herd of models”. In: *arXiv preprint arXiv:2407.21783* (2024).

[78] CodeParrot. *GitHub Code Clean Dataset*. https://huggingface.co/datasets/codeparrot/github-code-clean. Contains 115M code files across 32 languages; Apache-2.0 license; accessed 2025-07-09. 2022.

[79] Kishore Papineni et al. “Bleu: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[80] Quescol. *Quescol - A Platform That Provides Previous Year Questions And Answers*. https://quescol.com/. Last accessed on Dec 23, 2023. 2023.

[81] Wikipedia contributors. *Kaggle*. https://en.wikipedia.org/wiki/Kaggle. Last accessed on Dec 23, 2023. 2023.

[82] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. “Do LLMs generate test oracles that capture the actual or the expected program behaviour?” In: *arXiv preprint arXiv:2410.21136* (2024).

[83] Zhangchen Xu et al. “KodCode: A Diverse, Challenging, and Verifiable Synthetic Dataset for Coding”. In: (2025). arXiv: 2503.02951 [cs.LG]. URL: https://arxiv.org/abs/2503.02951.

[84] Hanxi Guo et al. “Biscope: Ai-generated text detection by checking memorization of preceding tokens”. In: *Advances in Neural Information Processing Systems* 37 (2024), pp. 104065–104090.

[85] Sebastian Gehrmann, Hendrik Strobelt, and Alexander M Rush. “Gltr: Statistical detection and visualization of generated text”. In: *arXiv preprint arXiv:1906.04043* (2019).

[86] Thomas Lavergne, Tanguy Urvoy, and François Yvon. “Detecting Fake Content with Relative Entropy Scoring.” In: *Pan* 8.27-31 (2008), p. 4.

[87] Phuong T Nguyen et al. “GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT”. In: *Journal of Systems and Software* 214 (2024), p. 112059.

[88] Yue Wang et al. “Codet5+: Open code large language models for code understanding and generation”. In: *arXiv preprint arXiv:2305.07922* (2023).

[89] Yinhan Liu et al. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[90] Chengzhi Mao et al. “Raidar: generative ai detection via rewriting”. In: *arXiv preprint arXiv:2401.12970* (2024).

[91] Xianjun Yang et al. “Zero-shot detection of machine-generated codes”. In: *arXiv preprint arXiv:2310.05103* (2023).

[92] Ye Liu et al. “Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval”. In: *arXiv preprint arXiv:2411.12644* (2024).

[93] Abhimanyu Dubey et al. “The llama 3 herd of models”. In: *arXiv e-prints* (2024), arXiv–2407.

[94] Gemma Team et al. “Gemma: Open models based on gemini research and technology”. In: *arXiv preprint arXiv:2403.08295* (2024).

[95] Daya Guo et al. “Graphcodebert: Pre-training code representations with data flow”. In: *arXiv preprint arXiv:2009.08366* (2020).

[96] Ebtesam Almazrouei et al. “Falcon-40B: an open large language model with state-of-the-art performance”. In: (2023).

[97] Lewis Tunstall et al. “Creating a Coding Assistant with StarCoder”. In: *Hugging Face Blog* (2023). https://huggingface.co/blog/starchat.