UNIVERSITY OF SALERNO

DEPARTMENT OF COMPUTER ENGINEERING, ELECTRICAL
ENGINEERING AND APPLIED MATHEMATICS

Master's Degree Course in Computer Engineering

Natural Language Processing and
Large Language Models

Automatic Detection of LLM-Generated Source Code

Supervisor:
Ch.mo Prof.
Nicola Capuano

Candidate::
Nunzio Del Gaudio
Mat. 0622702277

ACADEMIC YEAR 2024/2025

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 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.

This thesis presents a more modest, yet equally important objective: **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 we can start in the 1943 during World War II with the invention of Colossus, one of the first digital electronic computers, developed in order 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 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 recent development at all: it dates back to 1957 with Fortran. Fortran is both a programming language and a 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, Visual Basic 5 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 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). We must remember that the first LLMs were typically designed to analyse and generate just natural language text, so not all LLMs were capable of generating code, or at least not of generating syntactically correct or functional code (no free lunch).

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

In the same year, several 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, 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 the AlphaCode 2 framework, reached performance within the top 15% of coding competition participants.

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..."* [39] in witch is demonstrated that a widespread use of LLMs code generator has negatively affects over students' capability.

Figure 1.1 illustrates possible correlation between final grade and the use of LLM. We can observe 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?"* [40]. 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, as demonstrated in the paper: *"Do Foundation Models Know Copyright?"* [41].

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 we might run into 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 Disappearing Data Problem..."* [42]. 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.

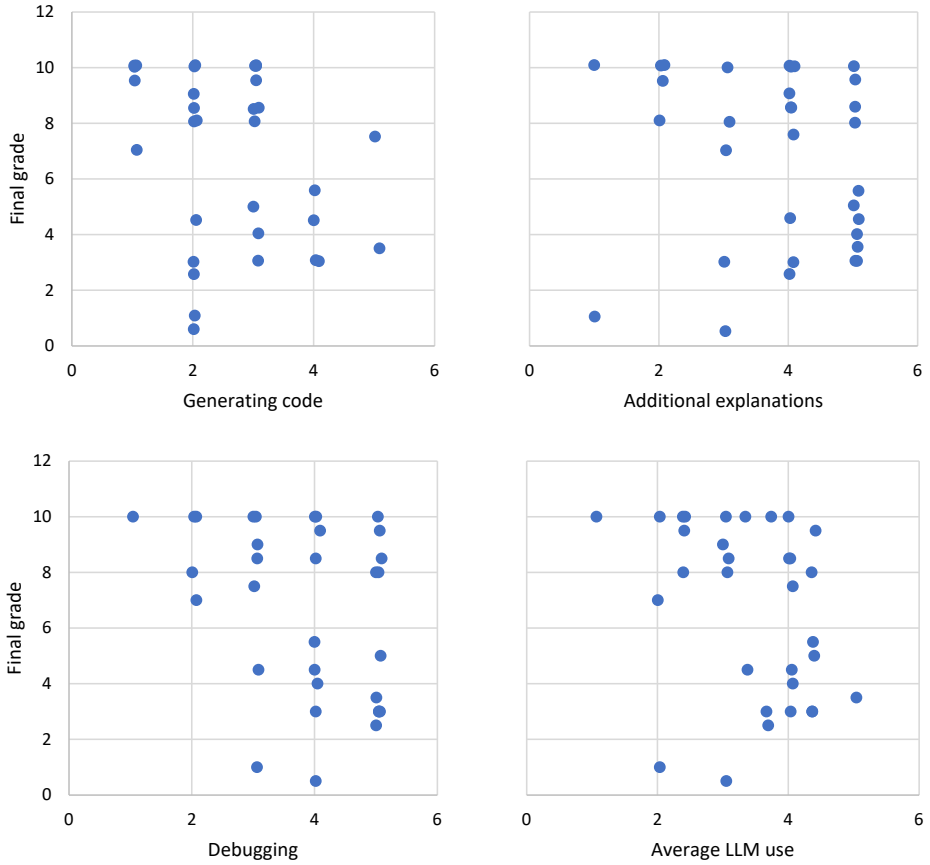

Figure 1.1: Scatter plot from the paper *The Impact of Large Language Models on Programming Education and Student Learning Outcomes*, illustrating the relationship between LLM usage and students' final performance.

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** [43], 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 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 [44]*, **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.

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 assumptions and limitations of each model. For instance, we will examine which datasets were originally used for training, which evaluation metrics are most informative in this context, the computational requirements of each method, and the specific scenarios in which they demonstrate optimal or suboptimal performance. In this phase, it will also be possible to identify the underlying reasons why certain models outperform others.

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, there are no widely established or consolidated approaches for detecting LLM-generated code. However, this does not imply a lack of research on the topic.

A subset of existing works focuses solely on providing datasets for training and evaluating detection models. The majority, instead, aim to propose detection techniques, including zero-shot approaches or methods requiring deep neural networks.

Since this area has only started to be actively explored around 2024, a significant portion of the literature has been published on platforms such as arXiv. Although arXiv is extremely useful for identifying cutting-edge research, it primarily hosts **preprints** (papers which have not yet undergone peer review by the scientific community).

This does not mean that arXiv publications should be disregarded, but the fact that most work in this area remains unpublished in peer-reviewed venues highlights the early and still-developing nature of this research topic.

Preprints

- **CoDet-M4** [45]: Presentation of a dataset obtained by using six different LLMs.
- **AIGCodeSet** [46]: Presentation of a python dataset obtained by using three different LLMs.
- **CodeMirage** [47]: !
- **How Far Are We?** [48]: Analysis of Natural Language Methods Applied to Code, the GPTSniffer Tool, and the Proposal of Alternative Detection Approaches.
- **DetectGPT4Code** [49]: Uncovering LLM-Generated Code: A Zero-Shot Synthetic Code Detector via Code Rewriting. In Proceedings of the AAAI Conference on Artificial Intelligence.

Peer-reviewed paper

- **Uncovering LLM-Generated Code** [50]: Proposal of a Detection Method Based on Code Rewriting and a Custom-Trained Similarity Model.
- **DetectCodeGPT** [51]: Analysis of Differences Between Human-Written Code and LLM-Generated Code, and Introduction of a Detector Based on Purely Stylistic Perturbations.
- **Detecting AI-Generated Code Assignments Using Perplexity** [52]: Proposal of a Detector Based on Perturbation.

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

3.2 Datasets Proposed in the Scientific Literature

3.3 Final Dataset

Chapter 4

Detection Methods

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.

4.1 Evaluation Metrics

4.2 Evaluation of Existing Methods

4.3 Proposed Improvements

Chapter 5

User Interface

5.1 Requirements Analysis

5.2 Design Choices

5.3 Implementation and Features

Chapter 6

Conclusion

6.1 Final Evaluation and Perceived Quality

6.2 Future Work

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 and et al. "Attention is all you need". In: *Advances in Neural Information Processing Systems* 30 (2017).

[3] Tom Brown and et al. "Language Models are Few-Shot Learners". In: *NeurIPS* 33 (2020), pp. 1877–1901.

[4] OpenAI. *GPT-4 Technical Report*. https://openai.com/research/gpt-4. 2023.

[5] Aakanksha Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *arXiv preprint arXiv:2204.02311* (2022).

[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] Jianfeng Jiang and et al. "Introducing Mistral: A High-Performance Language Model". In: *Mistral AI* (2023).

[9] Robin Rombach and et al. "High-Resolution Image Synthesis with Latent Diffusion Models". In: *CVPR* (2022).

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

[11] Mark Chen and et al. "Evaluating Large Language Models Trained on Code". In: *arXiv preprint arXiv:2107.03374*. 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 and et al. "Efficient Estimation of Word Representations in Vector Space". In: *arXiv preprint arXiv:1301.3781* (2013).

[15] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *EMNLP*. 2014.

[16] Alec Radford and et al. “Language Models are Unsupervised Multitask Learners”. In: *OpenAI Blog* 1.8 (2019).

[17] Yue Wang and 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 and et al. “CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis”. In: *arXiv preprint arXiv:2203.13474* (2022).

[19] Guolin Zeng and et al. “CodeGeeX: A Pre-trained Model for Code Generation with Cross-Programming-Language Evaluation”. In: *arXiv preprint arXiv:2203.13474* (2022).

[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: *NAACL* (2019).

[29] Alec Radford and et al. “Improving Language Understanding by Generative Pre-Training”. In: *OpenAI blog* (2018).

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

[31] Hamel Husain and et al. “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search”. In: *arXiv preprint arXiv:1909.09436* (2019).

[32] Zhangyin Feng and et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *EMNLP* (2020).

[33] Jacob Austin and et al. “Program Synthesis with Large Language Models”. In: *arXiv preprint arXiv:2108.07732* (2021).

[34] Dan Hendrycks and et al. “Measuring Coding Challenge Competence With APPS”. In: *arXiv preprint arXiv:2105.09938* (2021).

[35] Shuo Lu and et al. “CodeXGLUE: A Benchmark Dataset and Open Challenge for Code Intelligence”. In: *arXiv preprint arXiv:2102.04664* (2021).

[36] Yujia Li and et al. “Competition-level Code Generation with AlphaCode”. In: *arXiv preprint arXiv:2203.07814* (2022).

[37] Alexander Xu and et al. “PolyCoder: An Open-Source Programming Language Model”. In: *arXiv preprint arXiv:2202.13169* (2022).

[38] Google DeepMind. “Gemini: Unlocking Multimodal Understanding and Reasoning”. In: (2024).

[39] 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), p. 4115. DOI: 10.3390/app14104115. URL: https://www.mdpi.com/2076-3417/14/10/4115.

[40] Alex Perry et al. “Do Users Write More Insecure Code with AI Assistants?” In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 931–948.

[41] Xudong Zhang, Inioluwa Deborah Raji, and et al. “Do Foundation Models Know Copyright?” In: *arXiv preprint arXiv:2305.02407* (2023).

[42] Saurav Krishna et al. “The Disappearing Data Problem in ML Training”. In: *arXiv preprint arXiv:2305.00118* (2023).

[43] Eric Mitchell et al. “DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature”. In: *arXiv preprint arXiv:2301.11305* (2023).

[44] Shuyan Ye and et al. “Uncovering LLM-Generated Code via Behavioral and Semantic Analysis”. In: *arXiv preprint arXiv:2307.05734* (2023).

[45] 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).

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

[47] 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).

[48] 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).

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

[50] 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.

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

[52] Zhenyu Xu and Victor S Sheng. “Detecting AI-generated code assignments using perplexity of large language models”. In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 38. 21. 2024, pp. 23155–23162.