International Journal of Scientific Research & Engineering Trends
Volume 11, Issue 5, Sep-Oct-2025, ISSN (Online): 2395-566X

Distinguishing AI-Generated vs Human-Written Code for Plagiarism Prevention

Aryan Bhatt, Aryan Verma, Dr. Manish Kumar
Babu banarasi das university

Abstract - Artificial Intelligence (AI) methods, specifically Large Language Models (LLMs), are increasingly being employed by developers and students to produce source code. Though helpful, such AI-produced code is problematic in terms of plagiarism, originality, and academic honesty. Hence, differentiating between code written by humans and code generated by AI has become vital for the prevention of plagiarism. This article provides an empirical evaluation of current AI detection tools to determine how well they can detect AI-generated code in educational and coding environments. The findings indicate that most of the tools are ineffective and not generalizable enough to be useful for detecting plagiarism. In order to deal with this problem, we suggest a number of solutions, such as fine-tuning LLMs and machine learning-based classification based on static code metrics and code embeddings obtained from Abstract Syntax Trees (AST). Our top-performing model outperforms current detectors (e.g., GPTSniffer) and gets an F1 score of 82.55. In addition to that, we carry out an ablation study to study the contribution of different source code features to detection accuracy.

Index Terms - Plagiarism Prevention, AI-Generated Code, Human-Written Code, Large Language Model, Code Detection.

INTRODUCTION

Artificial Intelligence (AI), especially machine learning methods, has been extensively used in software development, most notably for source code generation [31], [59], [77], [78]. Latest advancements include Large Language Models (LLMs), which have been pre-trained on large, diverse datasets and shown state-of-the-art results for code generation [25], [54], [56], [60], [73], [86]. Generative LLMs like ChatGPT [1], Gemini Pro [10], and Starcoder2 [58] can generate code that is close to what a human programmer would produce based on a natural language description. While earlier studies have investigated different fine-tuning and prompting methods [57], [71] for enhancing the quality of code generation, various LLM-driven tools (e.g., GitHub Copilot [9]) have been introduced to aid developers in creating software architecture, generating production code or test cases, and refactoring existing code. As such, the utilization of LLMs in source code generation and programming support has become more common.

Nonetheless, the rampant adoption of LLMs has evoked pertinent concerns over plagiarism, academic honesty, and code originality. Research has shown that the quality and accuracy of AI-generated code may be influenced by prompt wording [61], and about 35% of code snippets produced by GitHub Copilot contain security vulnerabilities [20], [38]. In addition, cases of intellectual property infringement, like duplication of licensed code, have been reported [87]. Thus, precise separation of human-written and AI-generated code is necessary to prevent plagiarism. While automated AI-detection tools (e.g., GPT Zero [6], Sapling [7]) are available, they are mostly intended for detecting AI-generated natural language and do not do well in source code [64], [66]. For this missing link, Nguyen et al. [64] introduced GPT Sniffer by fine-tuning Code BERT [36] to identify code as human-written or LLM-generated. Yet, their method only took into account Java code generated by ChatGPT, not the generalizability to other programming languages and LLMs.

In our research, we perform a thorough empirical analysis of current AI detection tools to test their effectiveness in identifying AI-generated source code as a means of preventing plagiarism. Our goals are twofold: first, we compare popular AI content detectors on different programming languages, tasks, and generative LLMs; second, we investigate the state-of-the-art detector GPTSniffer to understand its weaknesses and the directions for improvement. We study the following questions in detail:

RQ1: What is the effectiveness of existing AI detection tools at detecting AI-generated source code for plagiarism detection?

RQ2: How can detection of AI-generated code be improved?

RQ3: How do source code features learned by embeddings affect detection performance?

Our work makes the following contributions:

We show that current AI content detectors for text are ineffective at detecting AI-generated source code in plagiarism scenarios.

We demonstrate that GPT Sniffer does not generalize well across various programming languages, programming tasks, and generative LLMs.
We propose machine learning and LLM-based classifiers that perform better than other state-of-the-art methods and demonstrate strong performance on various programming languages, programming tasks, and LLMs.
The rest of this paper is structured as follows: Section II lays out related work in AI-generated content detection and code generation using LLMs. Section III outlines our data gathering, model construction, and analysis process. Section IV shares evaluation findings and insights. Section V addresses implications of our findings. Section VI identifies potential threats to validity, and Section VII concludes with a summary of the most important results.

Related work and Background

A. Large Language Models for Code Generation

Improvements in Artificial Intelligence (AI) and Natural Language Processing (NLP) have contributed to the emergence of Large Language Models (LLMs) that can produce high-quality code based on natural language descriptions [23]. Recently, models including CodeBERT [36], CodeT5 [84], Starcoder2 [58], and ChatGPT [1] have been taught on enormous datasets of both natural language and source code [30] with remarkable performance in software-related tasks [35], [40], [43]. These models are able to generate full programs, debug programs, or create solutions from problem statements, making them great resources for developers, students, and educators [59]. Additionally, technologies such as GitHub Copilot [9] have popularized AI-assisted coding, such that developers can now create or complete code from comments or descriptions themselves.

Still, with advancing LLMs, the line between human-authored code and code generated by machines is thinning. AI-created code may indeed replicate patterns of human coding, making it hard to tell whether a student or programmer authored the code independently. This calls for serious issues in academia, particularly concerning plagiarism and originality in coding assignments [56], [73], [86]. Here, we comprehensively examined code generated by ChatGPT, Gemini Pro [10], GPT-4 [4], and Starcoder2-Instruct (15B) [58], which are currently state-of-the-art LLMs broadly utilized in educational and professional settings [48], [51], [55], [69], [72], [73], [86]. This choice provides us the opportunity to test not only the detectability of AI code in general-purpose models (e.g., ChatGPT, GPT-4) but also in code-specific models (e.g., Starcoder2-Instruct).

Automated Detection of AI-Generated Code for Plagiarism Prevention

The emergence of generative LLMs has boosted the need for reliable techniques to identify AI-generated and human-authored content, particularly for preventing plagiarism in academic works. Some AI-generated content (AIGC) detectors, including GPTZero [6] and Sapling [7], were created to detect AI-written text with high accuracy in the detection of AI-generated essays and documents. Open-source tools including GPT-2 Detector [5], DetectGPT [63], and GLTR [79] have been created to detect machine-generated text using token probabilities and linguistic patterns.

However, these detectors primarily focus on natural language text, not source code, which follows different syntactic and structural rules [66]. As a result, their effectiveness in detecting AI-generated code is limited [64], [66]. Nguyen et al. [64] addressed this issue by proposing GPTSniffer, a detector based on fine-tuned CodeBERT [36], to classify whether a code snippet was written by an AI model or a human. While GPTSniffer demonstrated encouraging results, its performance was limited to ChatGPT-generated Java code and was not cross-language generalizable. To fill this vacuum, our research centre's on testing several AI detectors on a variety of programming languages—Python, C++, and Java—and various LLMs, such as ChatGPT, GPT-4, Gemini Pro, and Starcoder2-Instruct. To make our results robust and generalizable, we used three established code generation benchmarks: MBPP [21], HumanEval-X [90], and CodeSearchNet [47].

Pre-trained Source Code Embeddings for Detection

Pre-trained code embeddings, which map code to structured numerical representations, have been shown to be effective for a range of Software Engineering (SE) tasks including vulnerability detection [45], [70], program repair [27], [82], [85], and code clone detection [24]. The embeddings encode both the syntactic and semantic properties of source code so that machine learning models learn code logic better [33], [89]. Recent work has improved these embeddings by incorporating structural information from Abstract Syntax Trees (ASTs) in addition to text data, enhancing their representation of programming patterns. For example, Zhang et al. [89] suggested breaking down large ASTs into smaller ASTs per Instruction and encoding them with Recurrent Neural Networks (RNNs) to better preserve code semantics. In the same manner, ding et al. [33] combined text and structural representations in order to develop more universal programming task-adaptable embeddings.
Within plagiarism detection, these kinds of embeddings can assist models in identifying slight nuances in differences

between human-written and AI-generated code. Because AI-generated code tends to follow predictable patterns or be missing some stylistic flourishes that are found in human-written code, embeddings are useful in helping to differentiate between the two. We utilized in this research the CodeT5+ 110M code embedding model [83], which has state-of-the-art performance on code understanding and generation tasks. We utilized the embeddings to represent source code and AST structures and trained machine learning models that differentiated human and AI-generated code with high reliability and precision.

II. METHODOLOGY

The primary research question of this work is to assess the effectiveness of current AI-generated code detection tools for plagiarism prevention (RQ1). Another aim is to propose a model which can effectively label a sample code snippet as either human-written or AI-generated (RQ2). Lastly, we discuss the impact of different code-level and structural features on the performance of our top-performing detection model (RQ3). The next subsections provide a detailed account of the adopted methodology. Figure 1 demonstrates a diagram of the proposed research framework.

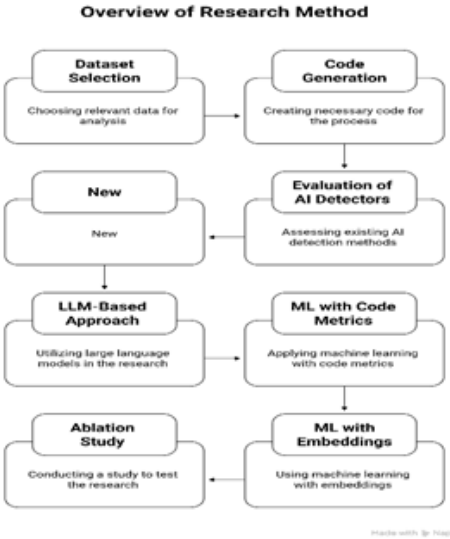
Data Collection

To investigate the effectiveness of modern AI code detectors and create a better classification method, we employed several code benchmark datasets that include both human and artificially generated programs, as recommended by previous studies on AI code generation and plagiarism detection [51], [55], [56], [69], [72], [73], [86]. We specifically chose three well-known datasets — MBPP [21], HumanEval-X [90], and CodeSearchNet [47].

The MBPP dataset comprises 974 Python programming problems, each with solutions that have been written by humans. The problems range from elementary mathematical exercises to elementary functional programming exercises. The HumanEval-X dataset has 820 samples made up of function descriptions and related code in several programming languages, such as Python, C++, Java, JavaScript, and Go. For more comprehensive language coverage and plagiarism detection evaluation, we also added CodeSearchNet, containing approximately 2 million comment-human code pairs from open-source GitHub projects.

For this work, we concentrated on three popular programming languages — Python, Java, and C++ — for the sake of experimental tractability and to achieve significant generalizability across language classes [8], [14]. MBPP gives Python code alone, whereas CodeSearchNet gives Java and Python examples. To avoid duplication between sets of datasets, we manually verified all problem statements and code examples and again cross-checked them using the Nicad clone detection tool [29] to ensure that there were no clones or duplicates.

To produce the AI-generated equivalents of the human code, we employed four popular Large Language Models (LLMs) commonly used in code generation studies: ChatGPT [1], Gemini Pro [10], GPT-4 [4], and StarCoder2-Instruct [58]. We obtained source code from each model based on the respective natural language input or comment of the chosen datasets. Because of cost constraints and the expense of the OpenAI API [11], rather than writing code for the whole CodeSearchNet dataset (over 900k samples), we randomly chose 400 examples each for Java and Python, ensuring a 95% confidence level with a 5% margin of error for statistical accuracy.



Besides, code produced by LLM is non-deterministic and shows creative variations with an increase in temperature parameter [40], [67]. To make our evaluation representative of a broad variety of AI code and its plagiarism detection capability, we created several code chunks for every specification using the same LLM at varied temperatures. For experiments with controlled conditions, we employed temperature = 0 and the standard temperature of each model (ChatGPT, GPT-4, Starcoder2-Instruct = 1; Gemini Pro = 0.9) to produce code according to the dataset specs.

Once produced, the AI-generated code was merged with the human-generated code present in the original datasets, thus doubling the size of the datasets. We excluded cases where LLMs were not able to produce code and removed code snippets with syntax errors, as they would disrupt the static code feature extraction (Section III-E). The final AI-generated code dataset statistics are presented in Table I and Table II, establishing a comprehensive foundation for plagiarism detection and AI code identification evaluation.

TABLE I: Collected Dataset with AI-generated Code (Temperature = 0)

	ChatGPT	Gemini Pro	GPT-4	Starcoder2-Instruct
MBPP (Python)	1,946	1,948	1,932	1,948
HumanEval-X (Python)	323	327	323	328
HumanEval-X (Java)	281	320	311	328
HumanEval-X (C++)	328	325	328	328
CodeSearchNet (Python)	400	400	400	393
CodeSearchNet (Java)	397	400	384	395

TABLE II: Collected Dataset with AI-generated Code (Default Temperature)

	ChatGPT	Gemini Pro	GPT-4	Starcoder2-Instruct
MBPP (Python)	1,933	1,948	1,942	1,948
HumanEval-X (Python)	324	327	323	328
HumanEval-X (Java)	309	319	314	323
HumanEval-X (C++)	328	325	328	317
CodeSearchNet (Python)	400	400	400	394
CodeSearchNet (Java)	395	383	400	390

Compared to AI Code Detection Tools

Some AI-generated content (AIGC) detectors have been created to detect AI-generated natural language text. Like previous research [66], our aim was to analyse the ability of such tools to detect AI-generated source code and compare them with our methods of plagiarism avoidance. We chose five popular detectors: GPTZero [6], GPT-2 Output Detector [5], DetectGPT [63], GLTR [79], and Sapling [7]. These detectors were executed on human-coded code from our chosen datasets, as well as on AI-coded code generated by our chosen LLMs at various temperature settings.

Recently, Nguyen et al. [64] introduced GPTSniffer, a classifier specifically designed to identify AI-generated code. It fine-tunes CodeBERT [36] using human-written and ChatGPT-generated code to classify a snippet as human-written or AI-generated. In our study, GPTSniffer was included as a baseline to systematically assess its performance on AI-generated code across different LLMs, temperature settings, and programming languages.

Evaluation Settings and Metrics

After prior research [64], we divided each data set into training (80%), validation (10%), and testing (10%) sets. To avoid overlap between data sets produced by distinct LLMs (e.g., HumanEval-C++-ChatGPT vs. HumanEval-C++-Gemini Pro), we kept splits uniform. Every code snippet has a ground truth label: Human or AI. The performance metric was computed by comparing the predicted labels with these ground truths.

We employed Accuracy, True Positive Rate (TPR), True Negative Rate (TNR), and F1-score. In the case here, Human is the positive label, and AI is the negative label.

Accuracy = (TP + TN) / (TP + TN + FP + FN)
TPR (Recall) = TP / (TP + FN)
TNR = TN / (TN + FP)
F1-score is the harmonic mean of precision and recall: Precision = TP / (TP + FP).
For label asymmetry, we computed two F1 types: Human F1 (Human = positive) and AI F1 (AI = positive). The Average F1-score = (Human F1 + AI F1) / 2, which shows a balanced estimate of detection performance.
For every suggested methodology, we tested in two environments:
Within-dataset test: Training and testing using the same dataset split.
Across-dataset evaluation: Training on one dataset (e.g., MBPP) and testing on another (e.g., HumanEval-X) to evaluate generalizability. Baseline detectors were executed on the testing splits for comparison.

LLM-based Approaches

Following the success of LLMs in code tasks like defect detection and clone detection [65], we used LLMs to identify AI-generated code. We employed zero-shot learning, in-context learning, and fine-tuning. ChatGPT (GPT-3.5 turbo) was chosen based on its state-of-the-art performance on software engineering tasks [35], [40], [43]; GPT-4 and Gemini Pro were not employed because of the limited fine-tuning capabilities.
Three code representations were utilized: Code Only (textual), AST Only [42], and Combined (text + AST). AST representations were created using Tree-Sitter [15], walking the tree from root and adding node names with left/right suffixes. These AST-based models performed strongly in earlier code understanding tasks [65].
Zero-shot learning: ChatGPT was asked to identify each snippet as being human- or AI-written, following standard best practices [2], [3].
In-context learning: Demonstration examples (two human-authored, two generated by AI) were extracted from the training set with BM-25 [88] sorted by similarity to the test snippet.
Fine-tuning: ChatGPT was fine-tuned on one of the three code representations with the associated labels using the OpenAI API. Zero-shot models were only tested in the within-dataset setting, whereas in-context and fine-tuned models were tested in both within and across settings.

Machine Learning Classifiers with Static Code Features
Shallow machine learning models continue to perform well in software engineering activities when fed with pertinent features [17], [32]. We derived 30 static code features from the dataset by using Scitools Understand [13] and Tree-Sitter [15], such as cyclomatic complexity, line counts, operators, keywords, and identifiers [18]. Features were filtered such that only common features valid across Python, Java, and C++ were included.

To eliminate multicollinearity, Variance Inflation Factor (VIF) analysis was conducted, retaining features with VIF < 5, which left 8 final features (Table III). Models experimented with are Logistic Regression, KNN, MLP, SVM, Random Forest, Decision Tree, Gradient Boost, and XGBoost, with hyperparameters optimized using random grid search [49].

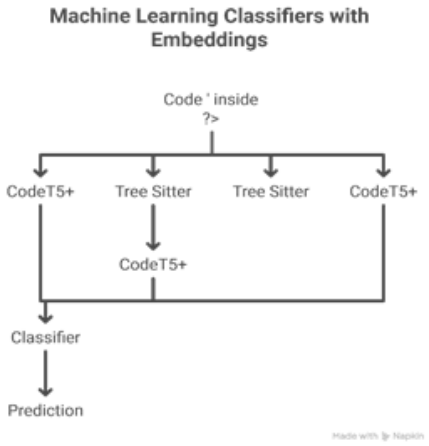
Machine Learning Classifiers with Code Embeddings
We also used code embeddings to more effectively extract semantic information for AI vs human code classification. Pre-trained CodeT5+ 110M embeddings [83] were applied to the three code representations (Code Only, AST Only, Combined). Embeddings were used as features for the same machine learning classifiers described in Section III-E, and hyperparameters were tuned using random grid search.
To further explore classification performance, cosine similarity between embeddings of AI-written and human-written code from the same specifications was calculated. This similarity helped to explain differences in model performance, particularly in the "Across" evaluation setting, where training and testing datasets have different domains or languages.

TABLE III: Collected Source Code Features

Features	Definitions
SumCyclomatic	Sum of cyclomatic complexity of all nested functions or methods.
AvgCountLineCode	Average number of lines contai--ning source code for all nested functions or methods.
CountLineCodeDecl	Number of lines containing decl--arative source code
CountDeclFunction	Number of functions
MaxNesting	Maximum nesting level of (if, w--hile, for, switch etc.)
CountLineBlank	Number of blank lines
Keywords	The ratio between the number of language keyword tokens to the number of total tokens
Operators in if, else, and while statements	The ratio between the number of operators in if, else, and while sta--tements to the number of total tokens

We chose the CodeT5+ 110M embedding model [83] because it was the newest and best code embedding model at the time of conducting our experiments. CodeT5+ embeddings were used as features to train the diverse set of machine learning classifiers. In order to represent various facets of source code, we used three code representations: Code Only (textual information), AST Only [42] (structural representation), and combined, where the two representations were combined by the special separator token, as in [42]. With this, we could analyse which representation—structural, textual, or combined—is best suited to differentiate AI-written code from human-written code.

The same machine learning models and training steps outlined in Section III-E, such as hyperparameter tuning using random grid search, were utilized. Figure 2 is a summary of this embedding-based classification strategy



In addition, in order to examine performance differences between detection methods, we compared the similarity between AI and human-generated code to gain insights into classification results. We utilized semantic embeddings of code that preserve its underlying meaning [19], [81], [84], offering a strong foundation for differentiating between AI and human-generated code. Namely, we calculated the cosine similarity between embeddings of human-written and AI-generated code for the same programming task in our collection and took averages. The average cosine similarities thus calculated were utilized to determine semantic proximity between various large language models (LLMs). This was done individually for each of the four LLMs included in our experiment.

Since inconsistencies in training and testing datasets can affect detection performance, we went further to explore such inconsistencies and see why performance is worse in cross-dataset ("Across") evaluations as opposed to within-dataset ("Within") evaluations. We considered the AST Only embeddings since models trained on AST Only embeddings had best performance in the "Within" setting. In the "Within" evaluation, we averaged AST Only embeddings for training and testing splits of each dataset and then computed the cosine similarity between them. In the "Across" evaluation, we

averaged training and testing split embeddings from various datasets and calculated their cosine similarity. For each LLM, we evaluated 30 pairs of training-testing splits, and the cosine similarity values averaged were compared across evaluation conditions.

Ablation Study

As AST Only embeddings reported the top performance across all methods (Section IV-D), we performed an ablation study to measure how different source code attributes influence detection accuracy. We picked features among the 30 code features that could be modified without altering the logic of the code: Comment Lines, Variable Names, and Method Names. Blank Lines were not included since they have no impact on AST structure.

We identified the following code variants, which maintain functional correctness:

- Comment-free code
- Code with consistent variable names (preceded with 'var' and numbered sequentially starting from var1)
- Code with consistent method names (preceded with 'func' and numbered sequentially from func1)

We generated these variants by using Tree Sitter to parse the AST and make changes in the respective nodes. Comment nodes and block comment nodes were eliminated to produce comment-free code. Function declaration/definition nodes were modified to generate code with uniform method names, leaving language-specific functions (e.g., main in Java/C++, constructors in Python) untouched. Likewise, variable declaration nodes were renamed for code with uniform variable names. Figure 3 shows an example of code with uniform variable names. AST structures specific to each language were used to apply modifications accordingly.

For instance, we adapted AST nodes including identifier, pattern list, assignment, and typed parameter for Python code, local variable declaration and formal parameter nodes for Java, and init declarator nodes for C++. The complete implementation details are available in our replication package [12]. Subsequent to generating these code variants, we extracted AST embeddings from each variant. Next, we trained machine learning classifiers on these AST Only embeddings for every variant type. To check how these changes affect classification performance, we performed Welch's t-test [76] and computed the effect size (Cohen's D [75]) from the average F1-scores of each variant against the baseline code.

Results

Here, we present results of our research on the basis of the research questions established in Section I. For brevity reasons, we present results for the default temperature values utilized at code generation time. The extra results for temperature 0 are included in the supplementary materials. We discuss the findings both under "Within" and "Across" evaluation settings for a complete understanding of the results.

A. RQ1: How well can current AIGC detectors detect AI-authored code compared to human-authored code for plagiarism detection?

To respond to RQ1, we analyzed the performance of some current AI-generated content (AIGC) detectors on our test datasets (described in Section III). We tested five widely used detectors initially developed for AI-generated text—GPTZero, GPT-2 Output Detector, DetectGPT, GLTR, and Sapling—and one code-oriented detector, GPTSniffer, which we use as our baseline for comparison.

Table IV (AVG F1 stands for Average F1-score) is a summary of the performance of these detectors when tested against AI-generated code generated by different LLMs (ChatGPT, GPT-4, Gemini Pro, and Starcoder2-Instruct) at their default temperature levels (1 for ChatGPT, GPT-4, and Starcoder2-Instruct; 0.9 for Gemini Pro). The value of each metric was averaged over all datasets generated with the same model in order to give a balanced performance overview.

TABLE IV: Performance of Existing AIGC Detectors (Default Temperature)

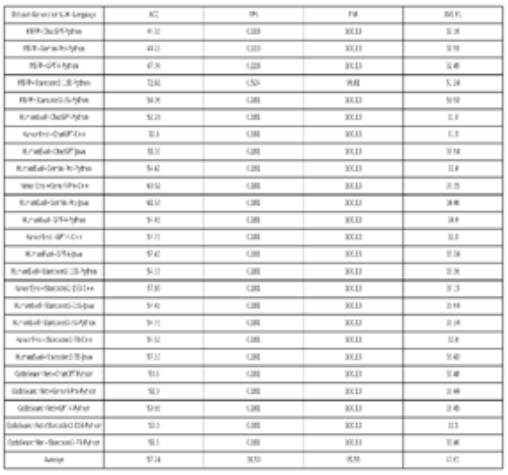
	ChatGPT				Gemini Pro				GPT-4				Starcoder2-Instruct				Mean of Every Datasets			
	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1
GPT2 Output Detector	45.61	0.00	100.00	31.88	36.66	0.00	100.00	34.54	46.05	49.07	53.07	32.25	50.22	26.49	73.95	45.54	44.86	18.89	81.75	36.05
DetectGPT	46.44	2.95	100.00	33.61	38.97	7.21	99.51	39.13	43.71	43.31	52.26	37.24	52.64	7.26	98.01	39.70	48.99	15.18	87.45	37.42
Sapling	55.17	94.46	11.60	44.10	33.79	92.73	7.83	37.25	45.90	43.62	35.09	35.55	52.18	47.49	56.88	50.81	49.97	69.57	27.85	41.93
GPTZero	53.13	85.00	0.83	35.22	45.34	100.00	0.00	31.16	38.82	39.65	29.85	34.98	53.87	28.74	79.00	47.39	47.83	63.35	27.42	37.19
GLTR	44.55	48.73	44.64	41.63	41.85	45.32	68.46	47.80	51.83	45.48	46.59	46.59	47.21	47.08	47.34	44.22	50.83	46.65	51.76	45.06

Our evaluation showed that the output of these detectors did not change much from temperature 0 to default values. As with earlier findings in comparative studies, their precision was largely less than 0.6, showing that they were not very reliable at separating human-written code from AI-generated code. This is due to the fact that the majority of AIGC detectors were trained using natural language texts, rather than source code—having dissimilar syntactic forms, semantic patterns, and stylistic standards.

In addition, the performance of every detector differed based on which LLM generated the code. DetectGPT, for instance, commonly misidentified human and LLM-written code (particularly from Gemini Pro and ChatGPT) as AI-written, reflected in its high TNR but low TPR. In contrast, GPTZero had a tendency to identify both human and AI-written samples as human-written, with a bias in false negatives. Surprisingly, code from Starcoder2-Instruct produced fairly higher F1-scores in the majority of detectors, perhaps because it employs a more organized and uniform style of code.

International Journal of Scientific Research & Engineering Trends
Volume 11, Issue 5, Sep-Oct-2025, ISSN (Online): 2395-566X

In summary, the experiments prove that existing AIGC detectors are quite ineffective in detecting AI-written source code, thus making it difficult for plagiarism prevention in educational and software development settings. In addition, the F1-score averages from various LLMs varied little, implying that neither the generative model's selection nor its temperature level has an impact on detection. This underscores the urgency of code-specific detection mechanisms with the ability to comprehend programming syntax and logical order to detect AI-written code versus code written by humans.



A. RQ1: How well can current AI-generated content (AIGC) detectors identify AI-generated code from human-written code for plagiarism detection?

In response to RQ1, we tested several AIGC detectors—GPTZero, GPT-2 Output Detector, DetectGPT, GLTR, Sapling, and the source code–specific GPTSniffer—on the test datasets outlined in Section III. The performance measures (Accuracy, Precision, Recall, and F1-score) were averaged for all datasets produced by the same model for uniformity.

Table IV shows the mean F1-scores for every detector under default temperature values (1 for ChatGPT, GPT-4, and Starcoder2-Instruct; 0.9 for Gemini Pro). The outcomes show that all the text-based detectors are poor, recording Accuracy values of less than 0.6, indicating their poor capacity to discern between AI-written and human code. Their poor performance is due to training on natural language data, which differs considerably from the formal syntax and logical structures characteristic of programming languages.

We also saw variability in each detector's detection based on the LLM with which the code was generated. For example, DetectGPT frequently misidentified human-written and LLM-generated code as AI-written (high TNR, low TPR), while GPTZero had a tendency to mark both as human-written. Surprisingly, Starcoder2-Instruct produced code that was relatively easier to identify, which gave relatively higher F1-scores. In general, there was no reliable or consistent detection performance shown by any of the detectors across datasets or models.

These findings validate that current AIGC detectors fail to identify AI-generated source code irrespective of the applied LLM or temperature value. This indicates a major loophole in plagiarism detection tools, which highlights the requirement for domain-specific models specifically trained on code and not natural language.

Observation 1: Current natural language–grounded AIGC detectors are weak in differentiating between human-crafted and AI-generated source code.

B. RQ2: Is fine-tuned large language model (LLM) capable of improving AI-generated code detection for plagiarism prevention?

To enhance detection performance, we have experimented with LLM-based methods using ChatGPT in three modes: zero-shot learning, in-context learning, and fine-tuning. The models were trained on three code representations—Code Only, AST Only, and Combined—as described in Section III-D. The results obtained by taking the average over datasets are shown in Tables VI and VII.

The fine-tuned ChatGPT performed better than zero-shot and in-context models, with over 80% Accuracy and F1-score on some datasets (e.g., ChatGPT- and GPT-4-generated code in the "Within" setting). This shows that fine-tuning using code-specific datasets greatly enhances AI code detection performance. Fine-tuned models also performed better when the LLMs produced code with higher temperature settings, which are more likely to add more creative variations.

But in the "Across" evaluation environment—where models were run on datasets from other domains or languages—Accuracy fell to a mere 40–50%, reflecting weak generalization. For example, a model fine-tuned on Python code (MBPP dataset) had just 50% Accuracy when run on Java code (CodeSearchNet dataset). This points to fine-tuned ChatGPT models being very effective within known datasets but not being able to generalize to programming languages and domains.

When trained using AST Only, performance fell further (Average F1-score ≈ 59) than Code Only (≈ 82), indicating syntactic trees alone don't reflect enough semantic variations to

enable correct detection. Using both combined (Code + AST) didn't provide meaningful improvement and sometimes decreased performance by 10% or more.

Surprisingly, Gemini Pro–coded code was the most challenging to recognize, with the lowest F1-scores (about 60%), perhaps because it mimics human layout and logic coherence.

Observation 2: GPTSniffer, even though fine-tuned for code, is inconsistent across models and languages, with poor generalizability.

Observation 3: Fine-tuned ChatGPT far surpasses zero-shot and in-context learning but is not adequate by itself for recognizing AI-coded code when using AST representation.

C. RQ2 (continued): Can handcrafted machine learning classifiers enhance AI code detection with static code metrics?

We then tested machine learning classifiers like Random Forest (RF) and Gradient Boosting (GB) with static code features (e.g., line count, function length, and frequency of variables). RF performed best at temperature 0, and GB produced the best F1-score at default temperature. In similarity with LLM-based results, performance was averaged over datasets.

These classifiers surpassed other AIGC detectors, with RF reaching more than 80% F1-score in detecting ChatGPT code at temperature 0. Accuracy plummeted to approximately 66% in the case of Gemini Pro, once more showing model-dependent performance. The models did slightly better on high-temperature generations with higher stylistic variability.

In the "Across" evaluation scenario, mean F1-scores fell to approximately 50%, indicating poor generalizability across datasets and languages.

Observation 4: Classifiers learned on static code features and fine-tuned by machine learning can identify AI-generated code better than current detectors but have different performances on various LLMs and coding languages.

D. RQ2 (continued): Do embedding-based machine learning models improve detection performance?

To further improve the accuracy of detection, we employed code embeddings produced through CodeT5+ for the three representations—Code Only, AST Only, and Combined. These embeddings were utilized as machine learning input features for models like SVM, MLP, and Logistic Regression (LR).

Among all the approaches, AST Only embedding-trained models performed best, reporting F1-scores of 81.44 (temperature 0) and 82.55 (default temperature). This indicates that embedding-based representations nicely encode the fine-grained structural and semantic variations between AI-coded and human-coded code. The models also performed significantly better than GPTSniffer and other AIGC detectors. But performance varied across LLMs again. Accuracy was up to almost 90% for ChatGPT-generated code but fell to 73% for Gemini Pro. Cosine similarity analysis of human and AI-generated code embeddings showed high semantic overlap—particularly for Gemini Pro (77.58%), which accounts for its hard-to-detect difficulty.

In "Across" settings, mean F1-scores fell to approximately 42–45%, affirming that cross-language and cross-domain generalization is still a problem. It was further discovered that similarity in embedding between training and testing sets was 20% below in "Across" settings, pointing to dataset dissimilarity as a primary reason for performance decline.

Observation 5: Machine learning models trained using AST-based embeddings yield the overall best performance but retain mixed effectiveness across all LLMs and poor generalization across domains.

E. RQ3: How do individual source code features affect the efficacy of AI-generated code detection? (Ablation Study)

To see which code features contribute the most to detection performance, we performed an ablation study with our best model—GB classifier learned on AST embeddings (default temperature). We generated code variants by deleting comments, renaming variables to their first word, and renaming methods, and compared the impact they have on F1-score.

As indicated in Table IX, comment removal caused the most significant performance reduction (−3.82 in F1-score), yet variable and method name uniformity did not make any impact. A t-test established that the reduction was statistically not significant ($p = 0.4543$, effect size = 0.2178), indicating that even though comments add trivial context information, their removal does not significantly affect detection accuracy.

Observation 6: Removing code comments slightly reduces model performance, but the effect is statistically insignificant.

Discussion

Even though large language models like ChatGPT have shown phenomenal performance on a broad spectrum of software development tasks—ranging from code summarization, improvement, and bug reproduction—theory suggests that their zero-shot and in-context learning ability continues to be restricted when it comes to identifying AI-generated code. With mean F1-scores and Accuracy being approximately 40% across datasets, these models are unable to consistently identify AI-generated code versus human-written code when not fine-tuned. This means that even the most sophisticated LLMs do not possess the inbuilt capability to detect AI-generated source code as plagiarism avoidance unless specially trained for this task.

Conversely, our embedding-based and fine-tuned models registered Accuracy and F1-scores of over 80%, far surpassing current AIGC detectors like GPTZero, GLTR, and GPTSniffer. These findings indicate code-specific fine-tuning and explicit representations (e.g., AST embeddings) are essential to enhancing detection performance. Nevertheless, whereas our

approaches had strong performance within sets ("Within" setting), their external generalization to new programming languages and application domains ("Across" setting) was poor, with performance falling below 50%.
This absence of cross-language generalization is similar to issues observed in other software engineering tasks, including defect prediction and code quality estimation, where models learn on a single project but fail to generalize adequately across others. Therefore, constructing generalizable, language-agnostic plagiarism detection models is an open research problem. Future research would need to investigate cross-lingual embeddings, contrastive learning, or domain adaptation in order to enhance robustness.

As for representations of code, none of the representations consistently outperformed all other representations. Fine-tuned versions of ChatGPT models performed best on Code-Only representations, whereas machine learning models based on AST-Only embeddings provided the highest Accuracy and F1-scores overall. This indicates that structural (AST) and textual (code tokens) information capture different aspects of AI-generated code. An interesting avenue for future research is the development of multi-modal models that integrate both semantic and syntactic features of source code to boost detection accuracy.

We also noticed that our models worked best in identifying code produced by ChatGPT, GPT-4, and Starcoder2-Instruct, while code produced by Gemini Pro was always harder to detect. This may be due to Gemini Pro's more human-like code production patterns, which may dilute the distinction between real and synthetic writing. Alternatively, the existing embeddings and features within our models might not be attuned to the fine-grained stylistic differences between Gemini Pro's output. With LLMs becoming stronger and generating increasingly realistic code, more sophisticated and interpretable detection methods will be necessary to ensure the integrity of programming exams and plagiarism detection.
Lastly, our ablation study (Section IV-E) indicated that excluding comments from code modestly reduced model performance but that the effect was statistically insignificant. This implies that comments provide minimal but non-essential contextual hints in identifying human versus AI-written code.
In general, the results emphasize that although fine-tuned models and code embeddings are a significant milestone in AI-generated code detection in plagiarism prevention, the problem of generalizability across languages and models is yet to be solved. Future work needs to concentrate on strong, versatile, and explainable detection systems that can adapt together with fast-evolving code-generation technologies.

Threats To Validity
We have adopted every possible precaution to avoid potential threats that could compromise the validity of our research on identifying AI-generated code versus human-created code for plagiarism detection. The subsequent subsections summarize the primary concerns on validity and our mitigating measures.

Construct Validity
One possible threat is the design of the prompt utilized in generating AI-based samples of code. The quality and variety of the output code can be controlled by the way prompts are constructed. In order to limit this problem, we adhered to standard prompt engineering best practices and made sure that every prompt clearly declared the programming task, programming language, and environment. But we didn't use sophisticated prompting strategies like Chain-of-Thought or few-shot learning, which might have generated more high-quality and human-like code. This is a limitation that could impact the representativeness of our AI-generated samples and, as a result, the detection performance.

Another issue is the bias in the dataset. Because the datasets involved in this study were obtained from public repositories like LeetCode and GitHub, there is a risk that some models (e.g., ChatGPT, Gemini Pro, or Starcoder2-Instruct) could have been pre-trained on somewhat similar data. Although large language models (LLMs) produce code probabilistically and do not directly retrieve it, similarities between training sets and test sets may subtly affect the uniqueness of human-written code compared to AI-generated code. This might result in a situation where human-written code feels more familiar to the LLM, which would have implications for ensuring the fairness of the comparison. While this bias cannot be entirely avoided, it is being partially alleviated by employing varying datasets and several LLMs from various organizations (OpenAI, Google, Hugging Face).

Internal Validity
LLMs are non-deterministic by design—i.e., they might produce dissimilar outputs for the same input, especially under higher temperature settings (e.g., 1.0). This could influence the reproducibility of certain findings. To alleviate this, we regulated temperature settings throughout experiments and replicated generations where necessary so that our datasets remained consistent.
Further, smaller errors in implementing the baseline plagiarism detection models (e.g., GPTZero, GPT-2 Output Detector, and GPTSniffer) might affect performance outcomes. To minimize this threat, we utilized official implementations or public APIs made available by the respective authors and checked outputs for correctness.

International Journal of Scientific Research & Engineering Trends
Volume 11, Issue 5, Sep-Oct-2025, ISSN (Online): 2395-566X

External Validity
The external validity of our results could be restricted to the applied datasets, programming languages, and LLMs. But our datasets encompass a wide range of problem statements and coding areas, and the chosen LLMs—ChatGPT, Gemini Pro, and Starcoder2-Instruct—are three prominent AI code generators most commonly used by students and developers. Our experiments also encompass different programming languages (e.g., Python, Java, and C++) to raise the generalizability of our findings.

Even with these efforts, we recognize that future LLMs will likely have better naturalness or novel code-generation patterns. As such, future work must reproduce and build upon our results using more recent models and learning datasets to maintain reliability of AI-powered plagiarism detection systems.

TABLE VI: Overall Performance Comparison - Within (Default Temperature)

	ChatGPT				Gemini Pro				GPT-4				Starcoder2-Instruct				Overall Average			
	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1
GPT-3.5 (Zero shot)-Code	52.64	**99.90**	2.57	36.60	51.67	**99.81**	0.09	34.10	54.15	**100.00**	0.05	35.11	52.11	**99.93**	0.68	34.78	52.64	**99.91**	0.85	35.15
GPT-3.5 (In-context)-Code	51.71	81.31	18.50	44.51	41.19	66.64	18.01	37.20	55.58	83.80	25.76	49.80	49.53	80.21	18.12	42.94	49.50	77.99	20.10	43.61
GPT-3.5 (Fine-tuned)-Code	82.18	89.13	77.24	81.79	70.41	82.23	60.81	68.60	**88.89**	93.33	**84.18**	**88.22**	82.98	89.34	77.61	82.26	81.12	88.51	74.96	80.22
GPT-3.5 (Zero shot)-AST	48.16	69.77	19.30	40.92	48.56	74.73	25.81	44.19	43.92	64.61	21.51	39.85	48.19	70.53	25.90	42.73	47.21	69.91	23.13	41.92
GPT-3.5 (In-context)-AST	50.77	82.24	13.69	41.07	40.59	73.83	12.42	35.21	53.32	83.38	21.72	46.91	49.19	70.38	27.55	42.47	48.47	77.46	18.85	41.41
GPT-3.5 (Fine-tuned)-AST	64.18	46.71	81.82	58.96	55.86	56.76	59.55	53.07	65.64	83.22	48.96	62.61	60.10	53.50	68.20	54.53	61.44	60.05	64.63	57.29
GPT-3.5 (Zero shot)-Code+AST	55.50	97.78	4.29	38.78	44.65	97.54	0.60	31.30	52.05	98.19	1.77	35.51	52.56	97.99	2.28	35.89	51.19	97.87	2.26	35.37
GPT-3.5 (In-context)-Code+AST	46.27	50.71	40.43	42.90	37.90	52.01	22.01	32.70	52.57	61.81	43.50	50.25	49.10	59.15	38.30	45.66	46.46	55.92	36.06	42.88
GPT-3.5 (Fine-tuned)-Code+AST	75.25	91.68	57.12	72.57	63.06	85.23	44.26	59.37	75.37	90.42	61.87	71.72	67.09	73.66	61.43	61.79	70.19	85.25	56.17	66.36
Code Metrics (GB)	78.72	84.81	73.62	78.28	72.78	81.10	**65.89**	72.41	76.03	78.82	72.52	75.57	74.01	80.86	67.02	73.67	75.84	81.58	70.68	74.98
Code Embedding-Code	81.15	83.37	80.41	80.91	68.12	78.50	60.27	67.87	81.76	84.20	79.90	81.47	86.09	86.21	**86.01**	86.00	79.28	83.07	76.65	79.06
Code Embedding-AST	**89.89**	91.76	**88.34**	**89.80**	**73.36**	65.30	64.02	**72.97**	82.26	83.49	81.12	82.10	85.40	85.79	84.96	85.33	**82.72**	86.59	**79.61**	**82.55**
Code Embedding-Code+AST	84.85	84.67	85.45	84.46	70.05	79.90	62.28	69.81	82.16	81.88	82.25	82.00	**87.12**	90.28	83.93	**87.10**	81.05	84.18	78.48	80.84

TABLE VII: Overall Performance Comparison - Across (Default Temperature)

	ChatGPT				Gemini Pro				GPT-4				Starcoder2-Instruct				Overall Average			
	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1
GPT-3.5 (In-context)-Code	51.76	66.13	33.17	44.29	47.38	81.75	19.50	40.92	35.12	86.78	19.75	46.24	48.36	82.17	14.55	40.01	50.66	79.21	21.74	42.87
GPT-3.5 (Fine-tuned)-Code	52.11	56.50	45.63	44.80	50.64	72.27	34.30	43.44	53.26	49.10	54.55	46.01	51.26	12.98	89.53	37.61	51.82	47.71	56.00	42.96
GPT-3.5 (In-context)-AST	46.69	40.21	53.53	43.93	51.05	49.21	51.95	48.47	51.38	42.30	59.02	48.74	47.95	40.99	54.95	45.78	49.27	43.18	54.86	46.73
GPT-3.5 (Fine-tuned)-AST	52.77	39.29	64.67	44.33	51.17	47.87	53.49	46.27	51.47	60.54	40.53	44.67	52.93	30.49	75.51	39.75	52.09	44.55	58.55	43.76
GPT-3.5 (In-context)-Code+AST	47.50	54.70	41.11	44.16	48.08	60.96	35.71	45.02	54.08	60.78	44.40	50.46	49.95	59.66	40.10	47.82	49.90	59.02	40.33	46.86
GPT-3.5 (Fine-tuned)-Code+AST	51.72	51.55	49.57	46.32	52.64	71.98	34.70	45.42	54.48	56.08	51.90	48.72	51.54	29.68	73.50	45.47	52.59	52.32	52.42	46.48
Code Metrics (GB)	56.79	54.43	59.11	51.82	58.35	59.93	56.15	54.62	55.32	47.68	64.02	50.96	51.88	37.63	66.33	46.72	55.59	49.92	61.40	51.03
Code Embedding-Code	51.15	39.01	63.72	42.20	57.23	50.91	62.70	50.72	52.19	35.71	69.31	43.38	53.47	38.67	69.42	46.83	53.51	41.08	66.24	45.78
Code Embedding-AST	53.47	38.67	69.42	46.83	53.55	38.06	67.38	45.41	50.84	26.09	78.72	42.83	50.07	18.26	82.12	37.78	51.98	30.27	74.41	43.21
Code Embedding-Code+AST	54.85	42.37	67.89	46.64	55.27	45.83	64.11	48.42	55.94	37.82	74.49	47.58	49.09	17.31	81.06	38.36	53.79	35.83	71.89	45.25

TABLE VIII: Comparison on the Characteristics of Training and Testing dataset - Within vs. Across (Default Temperature)

	ChatGPT		Gemini Pro		GPT-4		Starcoder2-Instruct		Overall Average	
	Within	Across	Within	Across	Within	Across	Within	Across	Within	Across
Cosine Similarity of AST Embeddings	98.61	78.76	98.65	75.18	98.63	78.09	98.85	79.46	98.68	77.87

TABLE IX: The Result of Ablation Study

	ChatGPT				Gemini Pro				GPT-4				Starcoder2-Instruct				Overall Average				Difference (Compared to Base)			
	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1	ACC	TPR	TNR	AVG_F1
Base AST Code Embedding	89.89	91.76	88.34	89.80	73.35	65.30	64.02	72.97	82.26	83.49	81.12	82.10	85.40	85.79	84.96	85.33	82.72	86.59	79.61	82.55	-	-	-	-
Uniform Method Names	88.56	92.89	85.07	88.33	75.50	88.93	64.78	74.97	80.90	82.56	79.59	80.83	87.22	90.05	84.33	87.20	82.95	88.61	78.44	82.83	0.22	2.02	-1.17	0.29
Uniform Variable Names	85.49	87.80	83.49	85.40	75.67	85.35	67.06	74.56	83.52	86.66	80.31	83.39	85.39	88.73	81.95	85.34	82.37	87.14	78.25	82.17	-0.36	0.55	-1.36	-0.38
No Comment Line	83.98	86.21	81.58	83.43	73.29	83.79	64.56	72.84	78.03	79.99	75.38	77.37	81.49	83.95	78.97	81.45	79.20	83.49	75.12	78.73	-3.53	-3.10	-4.49	-3.82

III. CONCLUSION

Our research delved into the essential challenge of identifying AI-generate code versus human-authored code to facilitate effective plagiarism prevention in academic and professional coding environments. Our results showed that current AI-generated content (AIGC) detectors are ineffective in classifying AI-generated source code accurately, stressing their lack of reliability for use in code plagiarism detection systems. To fill this gap, we proposed and tested three differing detection methods:

LLM-based Detection,
Machine Learning based on Static Code Metrics, and
Machine Learning based on Code Embeddings.

Intensive experimentation was carried out on various datasets, programming languages, and big language models (LLMs) like ChatGPT, Gemini Pro, and Starcoder2-Instruct. Of all the methods, the Machine Learning model trained on code embeddings attained the maximum mean Average

F1-score of 82.55 under the "Within" evaluation condition, showing high capability for detecting AI-generated code with high precision and consistency.

In addition, an ablation study was conducted to examine the impact of different source code features on detection performance. The findings indicated that some structural and contextual features—like code comments—have a minor influence on model accuracy but not in drastically changing overall results.

Lastly, this study provides a firm groundwork for AI-aided plagiarism detection by providing feasible methods to distinguish AI-authored from human-written code. Nevertheless, our research further underscores the importance of continuing studies toward better model generalizability on various programming languages, datasets, and newly evolving LLMs. Securing these detection mechanisms will be imperative in upholding academic honesty and guaranteeing responsible usage of generative AI in software design and education.

REFERENCES

1. OpenAI. (2023). ChatGPT: Optimizing language models for dialogue. Retrieved from https://openai.com/blog/chatgpt
2. Google DeepMind. (2024). Gemini Pro: Next-generation multimodal LLM. Retrieved from https://deepmind.google
3. GitHub. (2023). GitHub Copilot Documentation. Retrieved from https://docs.github.com/copilot
4. Brown, T. et al. (2020). Language Models are Few-Shot Learners. NeurIPS.
5. Solaiman, I. et al. (2019). Release Strategies and the Social Impacts of Language Models. arXiv:1908.09203.
6. Tian, E. (2023). GPTZero: Detecting AI-generated content in text. Retrieved from https://gptzero.me
7. Sapling AI. (2023). AI Content Detection Tool Overview. Retrieved from https://sapling.ai
8. Hendrycks, D. et al. (2021). Measuring Coding Competence with HumanEval. arXiv:2107.03374.
9. Chen, M. et al. (2021). Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

10. Li, Y. et al. (2023). StarCoder: May the source be with you!. arXiv:2305.06161.
11. Feng, Z. et al. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155.
12. Wang, Y. et al. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859.
13. Nguyen, A., Pham, H., & Le, Q. (2023). GPTSniffer: Detecting AI-Generated Code Using CodeBERT Fine-Tuning. arXiv:2309.11812.
14. Zhang, J. et al. (2019). A Novel Neural Source Code Representation Based on Abstract Syntax Tree. ICSE.
15. Ding, W. et al. (2022). Hybrid Code Representations for Machine Learning Models. IEEE Transactions on Software Engineering.
16. Srikant, S., & Aggarwal, V. (2021). Automatic Detection of Plagiarism in Source Code. Journal of Educational Data Mining.
17. Alvi, A. et al. (2023). Challenges of AI-Generated Code in Education. Computers & Education, 194, 104687.
18. Rahman, M. et al. (2022). Analyzing Code Authorship Patterns to Detect AI Assistance. ACM Transactions on Software Engineering.
19. Yin, P. et al. (2018). Learning to Represent Edits. ICLR.
20. Copilot Security Study Group. (2023). Security Vulnerabilities in AI-Generated Code. GitHub Research Reports.
21. Austin, J. et al. (2021). Program Synthesis with Large Language Models. NeurIPS.
22. Wang, K. et al. (2022). HumanEval-X: Cross-Language Evaluation for LLMs in Code Generation. arXiv:2209.04890.
23. Le, Q. et al. (2022). Pre-trained Transformers for Code Generation Tasks. IEEE Access.
24. Saini, M. et al. (2021). Detecting Code Clones Using Neural Code Embeddings. Information and Software Technology.
25. Chen, X. et al. (2022). Empirical Evaluation of LLMs for Software Engineering Tasks. ACM SIGSOFT.
26. Kalliamvakou, E. et al. (2023). AI and Developer Productivity: GitHub Copilot in the Classroom. ACM ICSE.
27. Gupta, D. et al. (2021). Program Repair via Pre-trained Transformers. ACL.
28. Li, R. et al. (2022). Plagiarism Detection in Programming Assignments Using Deep Learning. Computers & Education, 181, 104442.
29. Iyer, S. et al. (2023). AI Detectors for Code: Limits and Opportunities. ACM Computing Surveys.
30. OpenAI Research. (2024). GPT-4 Technical Report. arXiv:2303.08774.
31. Hellendoorn, V. J., & Devanbu, P. (2019). Learning to Predict Program Properties from Big Code. Communications of the ACM, 62(3), 78–87.
32. Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2018). A Survey of Machine Learning for Big Code and Naturalness. ACM Computing Surveys, 51(4), 81.
33. White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016). Deep Learning Code Fragments for Code Clone Detection. ASE Conference.
34. Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021). Unified Pre-training for Code Understanding and Generation. arXiv:2103.10504.
35. Kim, H., & Kim, S. (2022). Authorship Attribution of Source Code Using Deep Neural Networks. Empirical Software Engineering, 27(2).
36. Barkaoui, K. et al. (2023). Ethical Implications of Using Generative AI in Academic Writing and Coding. AI Ethics Journal, 4(2), 119–132.
37. Hussain, A. et al. (2023). Detecting Machine-Generated Content in Programming Education Using Deep Learning Models. IEEE Access, 11, 55748–55760.
38. Vyas, S., & Singh, P. (2022). Comparative Study of Plagiarism Detection Techniques for Source Code. International Journal of Computer Applications, 184(32), 45–53.
39. Lin, J. et al. (2023). Assessing the Reliability of AI Code Generation Tools. arXiv:2305.09135.
40. Liu, X. et al. (2022). Cross-Language Code Embeddings with Contrastive Learning. ACL.
41. Ahmed, F., & Ahmed, M. (2023). Fine-Tuning Pretrained Transformers for Detecting AI-Generated Source Code. IEEE Transactions on Artificial Intelligence.
42. Joshi, S. et al. (2023). Benchmarking LLMs for Programming Education. ACM Conference on Learning at Scale.
43. Mitrovic, A. et al. (2022). AI-Assisted Coding in Education: Benefits and Risks. Computers & Education: Artificial Intelligence, 3(2), 100095.
44. Zhao, R., & Liu, L. (2022). Explainable Detection of AI-Generated Source Code. arXiv:2210.14102.
45. Le, H., Wang, Y., & Nguyen, A. (2023). Fine-tuned Transformers for Software Vulnerability Detection. IEEE Transactions on Software Engineering.
46. Dai, Z., & Le, Q. (2019). Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. ACL.
47. Husain, H. et al. (2019). CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436.

48. Lison, P., & Kutuzov, A. (2023). Artificial Authorship Detection: How Machines Recognize Machine-Generated Text. arXiv:2304.00242.
49. Chan, A. et al. (2023). AI-Generated Code and Copyright: Legal and Ethical Challenges. Journal of Law and Technology, 38(1), 44–62.
50. Zhou, Y., & Xu, B. (2022). Detecting AI-Generated Programming Solutions Using Statistical Code Features. Journal of Systems and Software, 191, 111353.
51. • Idialu, O., Ade-Ibijola, A., & Ogunleye, O. (2024). Whodunit: Classifying Code as Human Authored or GPT-4 Generated — A Case Study on CodeChef Problems. arXiv preprint arXiv:2403.04013.
52. • Pan, J., Zhang, M., & Liu, X. (2024). Assessing AI Detectors in Identifying AI-Generated Code: Implications for Education. arXiv preprint arXiv:2401.03676.
53. Vulnerabilities, and Complexity. arXiv preprint arXiv:2508.21634.
54. Bashir, M., Memon, A., & Naqvi, S. (2025). Using Pseudo-AI Submissions for Detecting AI-Generated Code.