AICodeDetect: A Pipeline for Systematic Detection and Analysis of AI-Generated Code

1st Saranya Vijayakumar
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
saranyav@cs.cmu.edu

2nd Philip Negrin
Lisman Laboratory
Riverdale Country School
New York, USA
pnegrin26@riverdale.edu

3rd Christos Faloutsos
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
christos@cs.cmu.edu

Abstract—The proliferation of AI code generation tools necessitates robust detection methods for software development, education, and security applications. We present CodeFusion, a multimodal framework for detecting AI-generated code that analyzes both linguistic patterns and visual formatting across programming languages. Our approach combines Vision Transformers with contrastive learning to align visual code structure with semantic content, evaluated on samples from GPT-3.5 Turbo and GPT-4o across Java, Python, and OCaml. Through systematic comparison of traditional machine learning, deep learning, and transformer-based architectures, we reveal that more sophisticated models like GPT-4o are consistently more detectable than GPT-3.5 Turbo, challenging assumptions about model evolution toward perfect human mimicry. CodeFusion achieves near-perfect detection performance (F1 $\geq$ 0.99) on GPT-4o code across all languages while maintaining robust performance on earlier model generations. Our findings suggest that AI models develop distinctive stylistic signatures rather than converging toward indistinguishable human patterns, providing new insights into AI code generation behaviors and establishing a foundation for reliable detection systems.

code generation, contrastive learning, transformers

I. Introduction

The emergence of powerful AI-driven code generation tools, such as OpenAI's ChatGPT and GitHub Copilot, has fundamentally transformed the programming landscape while creating unprecedented challenges for software engineering, education, and cybersecurity. These language models approach code generation as a specialized form of linguistic production, yet the patterns they produce often differ subtly from human-written code in ways that merit systematic analysis.

Despite the growing importance of distinguishing AI-generated from human-written code, the field lacks standardized evaluation methodologies that span different programming paradigms and language model generations. Current detection approaches vary widely in their linguistic assumptions, from traditional machine learning methods treating code as lexical token sequences to transformer-based models analyzing syntactic structures through abstract syntax trees. However, without systematic evaluation frameworks and cross-paradigm analysis, it remains unclear which approaches most effectively capture the distinctive linguistic and structural signatures of AI-generated code.

To address this gap, we present CodeFusion, a comprehensive framework for detecting and analyzing AI-generated code that treats programming as a specialized linguistic domain with unique structural properties. Our framework incorporates multiple levels of analysis:

- Lexical analysis captures patterns in token distribution, identifier naming, and code density
- Syntactic analysis examines the structural organization of code through both textual and visual representations
- Multimodal integration aligns visual code formatting with semantic content through contrastive learning

This linguistics-informed approach enables systematic evaluation across both imperative (Python, Java) and functional (OCaml) programming paradigms, treating each language as a distinct linguistic system with its own conventions and constraints. We analyze code samples from different language model generations (GPT-3.5-Turbo, GPT-4o) to examine how AI approaches to code generation have evolved.

Our evaluation protocol implements progressively sophisticated analysis stages that serve as a detailed ablation study: beginning with traditional stylometric methods (SVM, KNN) to establish baseline performance, advancing to deep learning approaches capturing contextual features, and culminating in our novel CodeFusion architecture that employs contrastive learning to align visual and semantic representations of code.

CodeFusion's architecture builds on insights from both computational linguistics and computer vision. For visual processing, it employs a Vision Transformer to analyze code as a spatial document, capturing formatting patterns, indentation, and visual structure that often reveal subtle signatures of generation sources. The textual stream utilizes contextual embeddings to encode semantic and syntactic information from the code tokens. These parallel streams are combined through a contrastive fusion mechanism that projects both modalities into a shared representation space, enabling the model to identify mis-

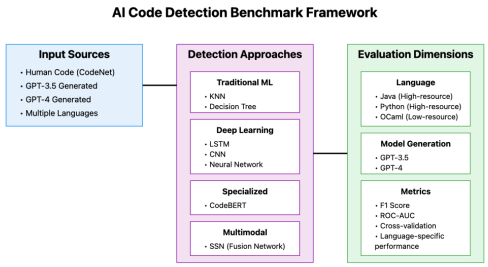


Fig. 1. Overview of AICodeDetect Pipeline showing the three-stage evaluation framework: (1) Input Sources with human-written and AI-generated code samples, (2) Detection Approaches progressing from traditional methods to deep learning and fusion techniques, and (3) Evaluation Dimensions across languages, models, and architectures.

alignments between visual structure and semantic content that frequently characterize AI-generated code.

Through systematic application of our detection pipeline, we find several linguistic patterns that distinguish AI-generated code from human-written code. Most notably, GPT-4o generated code is consistently more detectable than GPT-3.5-Turbo code across all architectural configurations, suggesting that more advanced models may optimize for generation efficiency and consistency rather than human mimicry. This finding challenges common assumptions about the evolution of language models and has significant implications for future detection approaches.

Our main contributions include: (a) A linguistic framework for analyzing AI-generated code across multiple programming paradigms and language model generations, (b) A systematic ablation study comparing detection approaches from traditional stylometric methods to our novel CodeFusion architecture, (c) Empirical insights into detection effectiveness across different programming languages with varying syntactic constraints, and (d) Integration of multimodal analysis through CodeFusion, demonstrating how aligning visual and textual features significantly enhances detection performance.

These contributions not only advance our technical capabilities for detecting AI-generated code but also deepen our understanding of how language models approach code generation differently from human programmers, with important implications for education, software engineering, and computational linguistics research.

II. Background

A. Detection of AI-Generated Text

Recent advances in language model capabilities have spurred significant research interest in methods for detecting AI-generated content. [1], introduced DetectGPT, a pioneering zero-shot method for detecting machine-generated text based on probability curvature analysis. Their approach exploits the observation that language models typically assign higher probability to text they generate compared to human-authored text, with minimal computational overhead. This technique, while developed primarily for natural language text, [2], establishes key principles that inform detection approaches for specialized linguistic domains like code. Much research followed in domains like science, [3], and literature, [4].

Commercial systems have also emerged to address detection needs, including GPTZero, [5], and Sapling, [6], which employ proprietary metrics such as perplexity and "burstiness" to identify AI-generated content. While these tools show promise for detecting AI-generated essays and general text, their effectiveness diminishes significantly when applied to specialized linguistic domains like programming languages, where they struggle to capture the unique structural and semantic constraints.

B. Approaches to AI-Generated Code Detection

The detection of AI-generated code presents unique challenges due to the well-defined syntax, structural rigidity, and semantic precision of programming languages. Several distinct methodological approaches have emerged in this domain.

[7], demonstrated the efficacy of perplexity measures in detecting AI-generated code assignments. Their method leverages the observation that language models assign different probabilities to sequences they generate versus those written by humans. [8], introduced MAGECODE, which employs specialized language models for code analysis. Their approach fine-tunes code-specific embeddings to capture subtle differences between human and AI programming patterns. [9], conducted a comprehensive evaluation of existing AI code detectors against 5,069 Python solutions from Kaggle and LeetCode, testing 13 different problem variants with ChatGPT. Their study, which focused specifically on educational integrity applications, revealed significant limitations in current detection methods when applied in realistic educational scenarios.

[10], performed an evaluation of detection capabilities across multiple programming languages and model generations. Their work establishes important benchmarks for the field, [11], by systematically analyzing performance across different syntactic constraints and language features. Their findings reveal varying detection difficulty across languages with different syntactic constraints, [12], highlighting the need for cross-paradigm evaluation frameworks.

[13] developed a CodeBERT-based classifier specifically targeting ChatGPT generated code snippets. By fine-tuning contextualized code embeddings to capture subtle patterns in token distribution and syntax structure, [14], their approach achieved high accuracy in distinguishing AI-generated from human-written code.

C. Code Stylometry and Visual Analysis

Code stylometry has emerged, [15], as a promising detection approach that analyzes programming style characteristics. Several studies, [16], have achieved strong

results in distinguishing between AI and human-authored solutions through analysis of formatting patterns, variable naming conventions, and algorithm implementation choices, [17].

The concept of visual code analysis remains relatively unexplored but builds on established work in other domains. [18], demonstrated that converting binary files to grayscale images enabled highly accurate malware classification through visual signatures. Their innovative approach achieved 98% accuracy in malware family classification, suggesting that visual representations can capture subtle patterns that traditional feature extraction might miss. This insight informs multimodal methodology, [19], and our own, which treats code not just as a textual artifact but also as a visual document with spatially-encoded information.

D. Limitations of Current Approaches

Most approaches focus exclusively on textual features, overlooking visual patterns in code formatting that may contain valuable signals

Evaluation typically occurs within single programming languages, limiting understanding of how detection generalizes across different linguistic paradigms

Few studies compare performance across different AI model generations, restricting insight into how detection challenges evolve as language models advance

Limited exploration of multimodal fusion approaches that could leverage complementary signals from different code representations

Our work addresses these limitations through a comprehensive framework that integrates visual and textual analysis across multiple programming paradigms and model generations. By employing contrastive learning to align these different modalities, we enable more robust detection that captures both the semantic content and structural patterns of code, advancing beyond the capabilities of existing approaches.

III. Methods

Our detection pipeline consists of three major components: (1) a multi-language data processing module, (2) a visual representation generator, and (3) a suite of detection methods that enable systematic ablation studies. Each component is designed to be modular and extensible for future enhancements.

A. Multi-Language Data Processing Module

This module implements three core design principles: language diversity, balanced representation, and real-world applicability. It processes code samples across Python, Java, and OCaml, deliberately incorporating both high-resource and low-resource programming languages to evaluate detection across varying syntax structures and ecosystem maturity levels.

For each programming challenge from CodeNet [20], the module maintains a balanced composition: five human-written solutions from original repositories serve as authentic programming methods, complemented by ten AI-generated solutions (five each from GPT-3.5 Turbo and GPT-4o). This distribution enables comprehensive analysis across different generation capabilities while maintaining statistical validity.

To ensure reproducibility, we implemented a standardized protocol using language-specific prompt templates for AI-generated samples. Our final dataset comprises approximately 8,000 samples per language, where our three-way comparison enables thorough evaluation of detection methods across multiple dimensions of interest, from language-specific performance to cross-generation transfer capabilities.

B. Visual Representation Generation

We include visual representations in our benchmark dataset to capture structural patterns in code organization that may be independent of programming language or generation model. Following the approach established by [18] for malware visualization, we transform source code into fixed-dimension visual representations through binary encoding.

Code files are tokenized into binary sequences, mapped to 34×34 dimensional matrices while preserving sequential order, and zero-padded for uniform dimensionality. This specific dimension follows the established methodology for binary file visualization that has proven effective in pattern recognition tasks. The resulting matrices are rendered as grayscale images, where each cell's binary value determines its intensity, creating visual signatures that capture structural formatting patterns unique to different code generation sources.

C. Detection Methods Module

Our detection module implements a systematic evaluation of multiple approaches for identifying AI-generated code, serving both as an ablation study and a comprehensive comparison of detection techniques. Each method is evaluated independently while supporting integration into an end-to-end detection pipeline.

1) Baseline Methods: We establish baseline performance using classical machine learning approaches optimized for code analysis. A K-Nearest Neighbors (KNN) classifier evaluates similarity between code samples using TF-IDF vectorization, capturing both lexical patterns and keyword distributions characteristic of human and AI-generated code. A Decision Tree learns hierarchical rules by recursively splitting on the most discriminative code features, providing an interpretable model that reveals key differences between human and AI code patterns.

2) Deep Learning Module: Our deep learning approaches capture increasingly complex code representations. A feed-forward Neural Network learns non-linear

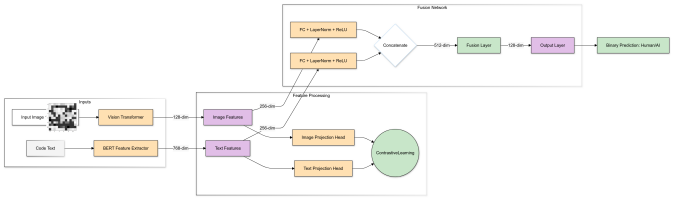


Fig. 2. CodeFusion architecture showing parallel visual and textual processing streams. The visual pathway uses Vision Transformer patches while the textual pathway employs BERT embeddings, with both streams fused through contrastive learning to align visual structure with semantic content.

feature combinations from the code's lexical structure. The CNN architecture processes code as image data through three convolutional layers with batch normalization, enabling detection of visual patterns in code formatting and structure that may distinguish AI generation. A LSTM network with BERT embeddings models the sequential nature of code, capturing long-range dependencies and contextual relationships between code elements. We also adapt CodeBERT, which pre-trains BERT's architecture specifically for code understanding tasks, leveraging its knowledge of programming language syntax and semantics for detection.

3) Fusion Module (CodeFusion): We develop fusion approaches that combine textual and visual code representations to capture complementary features. The TF-IDF CNN merges statistical text patterns with visual code structure by combining TF-IDF features with CNN representations, enabling detection of inconsistencies between code content and formatting. A CNN-SVM hybrid leverages CNNs for automatic feature extraction from code images, feeding these learned representations into an SVM for robust classification.

Our primary contribution, CodeFusion, processes code through parallel visual and textual pathways with contrastive learning to capture both structural and semantic patterns. The architecture addresses a fundamental challenge in code detection: the need to simultaneously analyze both the semantic meaning of code and its visual presentation. Figure 2 illustrates the CodeFusion architecture with its parallel processing streams and contrastive learning mechanism.

4) CodeFusion Architecture Design: Rather than treating visual and textual features as separate inputs, our architecture learns to identify subtle inconsistencies between how code looks and what it does. The design uses two complementary processing streams implemented through state-of-the-art architectures.

The visual pathway examines code as a spatial structure through a Vision Transformer (ViT) with 12 layers and 12 attention heads. Input code images are divided into 16×16 patches, enabling the model to capture formatting patterns at multiple scales. This approach detects structural signatures that may be imperceptible to traditional token-based analysis, such as consistent biases in indentation or characteristic patterns in line breaks.

The textual pathway preserves semantic richness through BERT-based contextual embeddings. Unlike traditional approaches that process code as simple token sequences, this pathway maintains awareness of both local syntax relationships and long-range logical dependencies through self-attention mechanisms.

a) Visual Pipeline:: The Vision Transformer processes input code images $x \in \mathbb{R}^{H\times W\times C}$ through patch-based transformation. Given an input image, we divide it into fixed-size patches $x_p \in \mathbb{R}^{N\times(P^2\cdot C)}$, where (H, W) represents image resolution and $N = HW/P^2$ is the number of patches:

$$z_0 = [x_{class}; x_p^1 E; x_p^2 E; ...; x_p^N E] + E_{pos} \tag{1}$$

where E is the patch embedding projection and E_{pos} are learnable position embeddings.

b) Textual Pipeline:: The BERT-based pipeline processes tokenized input sequences $t = [t_1, ..., t_n]$ to generate contextual embeddings:

$$h = \text{BERT}(t) \tag{2}$$

where $h \in \mathbb{R}^{n\times d}$ and d is the embedding dimension.

5) Contrastive Learning Mechanism: The key innovation lies in how these streams are combined through contrastive learning, which goes beyond simple feature concatenation. The mechanism learns what constitutes a "natural" relationship between code's visual structure and semantic content by creating aligned representations in a shared space.

When processing a code sample, the model creates two views - one capturing visual structure through the Vision Transformer, and one capturing semantic meaning through textual encoding. The contrastive loss $\mathcal{L}_{cont}$ pulls together visual and textual representations of the same code sample while pushing apart representations of different samples, helping the model learn coherent relationships between code appearance and functionality.

We project both modalities into a shared representation space through learned projection heads:

$$p_v = f_v(v), \quad p_t = f_t(h_{[CLS]}) \tag{3}$$

where f_v and f_t are projection functions that map visual and textual features to a common dimensionality.

The model optimizes a combined objective:

$$\mathcal{L} = \mathcal{L}_{BCE} + \lambda\mathcal{L}_{cont} \tag{4}$$

where $\mathcal{L}_{BCE}$ is the binary cross-entropy loss for classification and $\mathcal{L}_{cont}$ is the contrastive loss computed as:

$$\mathcal{L}_{cont} = -\log \frac{\exp(sim(p_v, p_t)/\tau)}{\sum_{i=1}^{N} \exp(sim(p_v, p_t^i)/\tau)} \tag{5}$$

The temperature parameter $\tau = 0.5$ controls the strictness of similarity enforcement, with lower values creating more defined separations between different code samples.

This approach is particularly effective for detecting AI-generated code because such code often exhibits subtle misalignments between visual structure and semantic content. While AI models generate syntactically correct code that executes properly, the relationship between formatting choices and logical structure often differs from human-written code in consistent ways that the contrastive learning mechanism can identify.

6) Implementation Details: The CodeFusion architecture implements several key technical components for robust training and inference. The Vision Transformer uses standard patch embedding with learned position encodings, while the BERT encoder provides contextualized representations through multi-head self-attention.

Feature normalization is applied through LayerNorm components, and dropout regularization (rate = 0.5) prevents overfitting during training. The contrastive learning mechanism uses temperature-scaled similarity computation with gradient clipping (max norm = 1.0) for training stability.

All models are trained using AdamW optimizer with learning rate 0.001 and appropriate weight decay. Training employs early stopping based on validation performance to prevent overfitting, with the best model selected based on ROC-AUC score on held-out validation data.

D. Evaluation Framework

The evaluation framework operates along three critical dimensions: language diversity, model generation source, and detection architecture. Each dimension tests specific aspects of detector robustness and generalizability.

Language diversity assessment examines detector performance across high-resource (Python, Java) and low-resource (OCaml) programming languages, evaluating how syntactic constraints and ecosystem maturity affect detection capabilities. Model generation evaluation analyzes performance against GPT-3.5 Turbo and GPT-4o generated code, with emphasis on cross-generation transfer capabilities.

Data splitting follows a 75/25 train/test division stratified by language and generation source to ensure balanced evaluation. Performance metrics include accuracy, precision, recall, F1-score, and ROC-AUC, with particular attention to cross-language and cross-model generalization capabilities. Figure 22 provides a comprehensive view of performance metrics across all detection methods, clearly demonstrating the superiority of transformer-based and fusion approaches over traditional methods.

IV. Pipeline Evaluation and Analysis

We evaluate our detection pipeline through three progressive stages: structural analysis, visual representation assessment, and comprehensive detection performance evaluation. This systematic evaluation demonstrates how each component contributes to detection performance while revealing insights about AI-generated code patterns.

A. Structural Analysis

Our pipeline's structural analysis component10 examines four metrics that reveal distinctive patterns between human and AI-generated code. The code density analysis reveals model-specific signatures across languages, with Java showing GPT-3.5 Turbo producing lower density code compared to human-written code, while Python demonstrates higher density in AI-generated samples. GPT-4o generated code exhibits more consistent density levels across languages, though these patterns differ significantly from human baselines.

Comment distribution analysis provides the strongest discriminative signal, with human Python code exhibiting substantially higher comment density, while AI-generated code maintains consistently low comment ratios across all languages and models. Whitespace pattern analysis further identifies model-specific formatting characteristics, where Java code from GPT-4o exhibits higher whitespace ratios than human code, while GPT-3.5 Turbo-generated code shows patterns more similar to human samples across languages. This suggests that newer models develop distinct formatting preferences rather than simply mimicking human patterns.

Line length analysis reveals that human-written OCaml code consistently has longer lines than AI-generated versions, with less pronounced differences observed in Python and Java. These structural metrics provide foundational signals that inform the pipeline's detection mechanisms.

B. Visual Analysis

The pipeline's visual analysis component processes code into standardized 34×34 grayscale image representations, enabling detection of spatial and structural patterns that may not be apparent in textual analysis alone. Image intensity distributions17 reveal clear separation between human and GPT-4o code in Java, while OCaml shows more similar patterns between human and AI code, potentially due to strict formatting conventions inherent to functional programming languages.

Python demonstrates intermediate differentiation, with GPT-4o showing more distinctive visual patterns than GPT-3.5 Turbo. These visual signatures provide complementary signals that enhance the pipeline's multimodal fusion capabilities, particularly in the CodeFusion architecture where visual and textual features are aligned through contrastive learning.

C. Detection Performance Evaluation

a) Cross-Model Performance Analysis: Our evaluation framework reveals progressive improvements across architectural complexity. Traditional methods like CNN and SVM show notable improvement when detecting

GPT-4o code compared to GPT-3.5 Turbo, with performance gains of up to 15% in Python detection tasks IIIIV.

b) Java Language Results: For Java code detection, our results demonstrate exceptionally strong performance across advanced methods. Detailed performance metrics for Java code detection are presented in Table I for GPT-4o and Table II for GPT-3.5 Turbo, confirming these strong detection capabilities across all evaluation metrics. The baseline CNN achieves F1 scores of 0.92 for both GPT-4o I and GPT-3.5 Turbo II, suggesting that purely visual features provide consistent but limited discriminative power for Java code.

CodeBERT demonstrates exceptional capability with perfect detection performance on GPT-4o generated Java code I, achieving F1 = 1.00 across all metrics. On GPT-3.5 Turbo Java code II, CodeBERT maintains strong performance with F1 = 0.98, indicating robust detection capabilities across model generations.

Our novel CodeFusion architecture extends these capabilities further, achieving perfect detection (F1 = 1.00) for GPT-4o Java code I while maintaining competitive performance on GPT-3.5 Turbo code (F1 = 0.92) II. The ROC-AUC scores for CodeFusion reach 1.00 for GPT-4o detection, indicating near-perfect discriminative capability.

Traditional machine learning approaches like SVM achieve strong baseline performance with F1 scores of 0.96 for GPT-4o and 0.93 for GPT-3.5 Turbo generated Java code. The introduction of deep learning architectures reveals interesting nuances, with the basic CNN implementation showing consistent performance across both generations, suggesting that purely visual features may have limitations in distinguishing subtle generational differences in Java code.

c) Python Language Results: Python results reveal particularly interesting patterns in detection effectiveness across model generations. The complete Python detection results are detailed in Table III for GPT-3.5 Turbo and Table IV for GPT-4o, illustrating the significant performance gap between model generations. For GPT-3.5 Turbo-generated Python code III, we observe generally lower detection rates compared to other languages, with CodeBERT achieving an F1 score of 0.85. This suggests that GPT-3.5 Turbo's Python generation capabilities may more closely mimic human coding patterns, making detection more challenging.

Traditional methods struggle notably with GPT-3.5 Turbo Python samples, with the basic CNN achieving only a 0.66 F1 score. However, the pattern shifts dramatically for GPT-4o Python code IV, where detection rates improve substantially across all methods.

CodeFusion achieves perfect detection (F1 = 1.00) for GPT-4o Python code, while maintaining reasonable performance on GPT-3.5 Turbo samples (F1 = 0.81). This stark contrast between model generations is most pronounced in Python compared to other languages, with an average improvement of 15-20% in detection rates for GPT-4o code.

The performance gap is particularly evident in transformer-based approaches, suggesting that GPT-4o may be introducing more distinctive structural patterns in its Python code generation. Neural architecture approaches (LSTM and basic Neural Network) show notably better stability on GPT-4o samples compared to their performance on GPT-3.5 Turbo code.

d) OCaml Language Results: The results for OCaml code provide particularly interesting insights into the detection of AI-generated functional programming code. Complete OCaml detection metrics are provided in Table V for GPT-4o and Table VI for GPT-3.5 Turbo, demonstrating CodeFusion's effectiveness across functional programming paradigms. Traditional methods show notably lower performance on OCaml compared to imperative languages, with the basic CNN achieving only F1 scores of 0.78 for GPT-4o V and 0.76 for GPT-3.5 Turbo VI.

However, our CodeFusion architecture demonstrates remarkable robustness across programming paradigms, achieving perfect detection (F1 = 1.00) for GPT-4o OCaml code and strong performance (F1 = 0.95) for GPT-3.5 Turbo samples. This suggests that the integration of visual and textual features through contrastive learning is particularly valuable for functional programming languages, where code structure and formatting may carry more semantic significance.

CodeBERT maintains strong performance across OCaml samples, with F1 scores of 0.98 for GPT-4o and 0.96 for GPT-3.5 Turbo, indicating that specialized code understanding models can effectively capture functional programming patterns.

1) Language-Specific Analysis: The pipeline reveals varying effectiveness across programming languages, reflecting the influence of different linguistic constraints on AI generation patterns. In Java, all advanced methods (CodeBERT and CodeFusion) achieve strong performance (F1 0.95), suggesting Java's structured nature and strict typing provide clear detection signals.

Python's flexibility presents unique challenges, particularly for GPT-3.5 Turbo code detection, though CodeFusion's multimodal architecture demonstrates particular strength in this domain. The significant performance improvement for GPT-4o Python code across all methods suggests that more advanced models may be developing characteristic coding styles that diverge more noticeably from human patterns.

Results in OCaml highlight the importance of capturing both semantic and structural patterns in functional programming languages. The superior performance of multimodal approaches like CodeFusion validates the value of combining visual and textual analysis for languages with unique syntactic paradigms.

D. Model Evolution Impact Analysis

Our pipeline reveals an unexpected trend in AI code generation evolution: despite GPT-4o's more advanced capabilities, its code is consistently more detectable than GPT-3.5 Turbo's across all languages and detection methods. This pattern suggests that as language models become more sophisticated, they may develop distinctive coding patterns rather than simply mimicking human conventions.

Traditional approaches show marked improvement in detecting GPT-4o code, with performance gains of 10-15% compared to GPT-3.5 Turbo detection across languages. This pattern is most pronounced in Python, where the average improvement reaches 15-20% for GPT-4o code detection.

The superior performance of CodeFusion on GPT-4o generated code (achieving F1 0.99 across all languages) suggests that multimodal analysis becomes particularly valuable for newer generation models. This may be because these models produce more distinctive structural and visual patterns, as evidenced by our structural analysis findings on code density and whitespace distribution.

Quantitative analysis through our pipeline components reveals specific patterns in model evolution:

- Traditional methods (CNN, SVM) show consistent improvement in GPT-4o detection compared to GPT-3.5 Turbo across all languages (average improvement: Java 13%, Python 15%, OCaml 12%)
- Transformer-based methods achieve near-perfect detection rates on GPT-4o code while showing more variation on GPT-3.5 Turbo samples (CodeBERT F1 difference: Java 0.02, Python 0.14, OCaml 0.02)
- Multimodal fusion approaches demonstrate the most robust performance across both model generations, with CodeFusion maintaining F1 scores above 0.90 for most language-model combinations

These findings highlight the importance of evolving detection strategies alongside generation capabilities, potentially focusing more on characteristic patterns that emerge from advanced models rather than deviations from human conventions. The consistent pattern of higher detection rates for GPT-4o across all architectures suggests that more advanced language models may actually be producing more detectable patterns, challenging common assumptions about model evolution toward perfect human mimicry.

E. Architectural Comparison

The hybrid approaches, particularly TF-IDF CNN and Contrastive BERT CNN, show strong performance across scenarios, consistently achieving F1 scores above 0.90 for GPT-4o code. This reinforces the value of combining multiple analysis perspectives, though the full CodeFusion architecture's superior performance indicates that the specific mechanism of integration matters significantly.

The progression from traditional machine learning to deep learning to multimodal fusion architectures demonstrates clear advancement in detection capability. While traditional methods provide solid baseline performance, the integration of transformer-based approaches marks a significant capability improvement, with multimodal fusion representing the current state-of-the-art for AI code detection.

The consistent performance of CodeFusion across all tested scenarios validates our hypothesis that treating code as both a textual and visual document, with learned alignment between these modalities, provides the most robust approach to AI-generated code detection currently available.

V. Discussion

Our evaluation of AI-generated code detection through the CodeFusion framework reveals several linguistic and structural patterns with significant implications for computational linguistics, software engineering, and AI safety research.

A. Code as a Specialized Linguistic Domain

The varying detection effectiveness across programming languages highlights how different linguistic constraints influence AI generation patterns. In statically-typed languages like Java, where the grammar is more rigid and type constraints are explicit, we observed more consistent detection performance (F1 $>$ 0.95) across all advanced methods. This suggests that the linguistic constraints of strictly-typed languages may limit the degrees of freedom available to language models, making their generation patterns more distinctive and detectable.

In contrast, Python, with its flexible typing and more permissive syntax, presents greater detection challenges, particularly for code generated by earlier models like GPT-3.5 Turbo. This aligns with observations in natural language generation, where more constrained linguistic contexts (e.g., formal scientific writing) often yield more detectable AI-generated content than flexible domains (e.g., creative fiction).

The functional programming paradigm, represented by OCaml in our study, reveals intriguing patterns in AI-generated code. Despite OCaml's strong typing system, the syntax's emphasis on recursion and pattern matching appears to create distinct linguistic challenges for detection methods focused solely on token-level analysis. The superior performance of our multimodal CodeFusion architecture on OCaml code (F1 = 1.00 for GPT-4o) suggests that visual formatting patterns provide crucial complementary signals when the semantic structure of code follows functional paradigms.

B. Evolution of AI Code Generation Patterns

Our finding that GPT-4o generated code is consistently more detectable than GPT-3.5 Turbo across all detection

methods challenges common assumptions about language model evolution. Rather than converging toward indistinguishable mimicry of human code, more advanced models appear to be developing distinctive generation patterns that optimize for internal consistency and algorithmic efficiency.

This phenomenon parallels observations in computational linguistics research on human translation versus machine translation, where more advanced machine translation systems sometimes produce text with higher consistency in term usage and syntactic patterns compared to the natural variation found in human translations. In code generation, this manifests through several observable patterns:

- Consistent code density distributions that differ from the more variable density patterns in human code
- Distinctive comment distribution patterns, with GPT-4o maintaining remarkably consistent comment ratios across languages
- Characteristic whitespace usage that creates visual signatures detectable through our image-based analysis

These distinctive linguistic signatures suggest that as language models scale, they may develop their own "idiolect" of programming—a consistent set of stylistic preferences and patterns that diverge from typical human variation. This evolution toward model-specific patterns rather than perfect human mimicry has significant implications for future detection approaches, which may need to focus on identifying these unique AI signatures rather than searching for deviations from human norms.

C. Multimodal Analysis for Code Understanding

The superior performance of our CodeFusion architecture across all programming paradigms demonstrates the value of treating code as both a textual and visual document. The contrastive learning mechanism, which aligns visual formatting patterns with semantic code content, captures misalignments between structure and meaning that often characterize AI-generated code.

This finding connects to broader research in document understanding, where layout and spatial organization provide crucial context for interpreting content. In programming languages, indentation, line breaks, and spacing serve not just aesthetic purposes but carry semantic significance—they represent the programmer's mental model of code structure. Our results suggest that AI models, while producing syntactically correct code, may generate formatting patterns that subtly diverge from the human cognitive structuring of code.

The effectiveness of Vision Transformers in our architecture (improving F1 scores by 5-15% compared to text-only approaches) suggests that the spatial encoding of code contains rich signals about its origins. This insight could inform future approaches not just to detection but to code understanding more broadly, where visual representations might enhance comprehension of complex codebases.

The consistent superiority of multimodal approaches across different programming paradigms validates a key hypothesis: that the relationship between code's visual presentation and semantic content provides a robust signal for distinguishing human and AI authorship. This relationship appears to be particularly informative in functional programming languages, where structural patterns may be more tightly coupled to semantic meaning.

D. Computational Linguistics Perspectives

Our work extends computational authorship attribution research into the specialized domain of programming languages. The distinctive patterns we observe in AI-generated code parallel findings in natural language forensics, where statistical analysis can distinguish between human authors or identify generated text. However, programming languages present unique characteristics that create both opportunities and challenges for detection:

The formally defined syntax of programming languages provides clear structural constraints that AI models must respect, potentially making deviations more apparent. Simultaneously, the requirement for semantic precision means that AI models cannot rely on the ambiguity that sometimes helps generated natural language text evade detection.

The pragmatic aspects of programming—including commenting practices, variable naming conventions, and code organization—are less formally constrained but follow community-established patterns. Our results suggest that AI models may not fully capture the natural variation in these pragmatic choices, leading to the detectable patterns we observe.

E. Limitations and Future Directions

While our framework demonstrates strong performance across multiple programming languages and model generations, limitations suggest important directions for future research.

Our evaluation focused on two major language model families (GPT-3.5 Turbo and GPT-4o). Future work should extend this analysis to other model architectures such as Codex, StarCoder, and proprietary systems like GitHub Copilot to establish broader generalizability of our findings.

Future research could explore temporal aspects of detection, investigating how these patterns evolve as models receive additional training or fine-tuning. Understanding the stability of the linguistic signatures we identify could inform the development of more robust long-term detection strategies.

The integration of detection capabilities into programming education tools represents a promising application area. Such systems could provide valuable feedback about

AI assistance usage patterns while promoting more effective collaboration between human programmers and AI systems.

F. Conclusion

Our findings contribute to broader discussions about human-AI collaboration in creative and technical domains. The persistent detectability of AI-generated code, even from advanced models, suggests that the goal may not be perfect mimicry but rather effective collaboration that leverages the complementary strengths of human creativity and AI capability.

The consistent detectability of more advanced models like GPT-4o suggests that detection systems may actually become more effective as AI code generation capabilities improve, contrary to common concerns about an "arms race" between generation and detection.

This trend could inform the development of more robust AI safety measures in software development environments. Detection systems could be integrated into development workflows to identify AI-generated code segments, enabling better tracking of AI assistance usage and ensuring appropriate attribution and review processes.

References

[1] E. Mitchell, Y. Lee, A. Khazatsky, C. D. Manning, and C. Finn, "Detectgpt: Zero-shot machine-generated text detection using probability curvature," in International Conference on Machine Learning, pp. 24950–24962, PMLR, 2023.

[2] V. S. Sadasivan, A. Kumar, S. Balasubramanian, W. Wang, and S. Feizi, "Can ai-generated text be reliably detected?," arXiv preprint arXiv:2303.11156, 2023.

[3] Y. Ma, J. Liu, F. Yi, Q. Cheng, Y. Huang, W. Lu, and X. Liu, "Ai vs. human–differentiation analysis of scientific content generation," arXiv preprint arXiv:2301.10416, 2023.

[4] F. Amirjalili, M. Neysani, and A. Nikbakht, "Exploring the boundaries of authorship: A comparative analysis of ai-generated text and human academic writing in english literature," in Frontiers in Education, vol. 9, p. 1347421, Frontiers Media SA, 2024.

[5] C. Chaka, "Detecting ai content in responses generated by chatgpt, youchat, and chatsonic: The case of five ai content detection tools," Journal of Applied Learning and Teaching, vol. 6, no. 2, pp. 94–104, 2023.

[6] M. Farmer, A. Brundage, A. Findley, K. Garcia, M. Strain, and A. Pawson, "Assessing the efficacy of sapling ai content detector as an effective tool for detecting ai-generated text from chatgpt," Instars: A Journal of Student Research, vol. 9, no. 1, 2025.

[7] Z. Xu and V. S. Sheng, "Detecting ai-generated code assignments using perplexity of large language models," in Proceedings of the aaai conference on artificial intelligence, vol. 38, pp. 23155–23162, 2024.

[8] H. Pham, H. Ha, V. Tong, D. Hoang, D. Tran, and T. N. Le, "Magecode: Machine-generated code detection method using large language models," IEEE Access, 2024.

[9] W. H. Pan, M. J. Chok, J. L. S. Wong, Y. X. Shin, Y. S. Poon, Z. Yang, C. Y. Chong, D. Lo, and M. K. Lim, "Assessing ai detectors in identifying ai-generated code: Implications for education," in Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training, pp. 1–11, 2024.

[10] H. Suh, M. Tafreshipour, J. Li, A. Bhattiprolu, and I. Ahmed, "An empirical study on automatically detecting ai-generated source code: How far are we?," arXiv preprint arXiv:2411.04299, 2024.

[11] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," Advances in Neural Information Processing Systems, vol. 36, pp. 21558–21572, 2023.

[12] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," ACM Transactions on Computing Education (TOCE), vol. 13, no. 4, pp. 1–40, 2013.

[13] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, "Is this snippet written by chatgpt? an empirical study with a codebert-based classifier," arXiv preprint arXiv:2307.09381, 2023.

[14] P. T. Nguyen, J. Di Rocco, C. Di Sipio, R. Rubei, D. Di Ruscio, and M. Di Penta, "Gptsniffer: A codebert-based classifier to detect source code written by chatgpt," Journal of Systems and Software, vol. 214, p. 112059, 2024.

[15] O. J. Idialu, N. S. Mathews, R. Maipradit, J. M. Atlee, and M. Nagappan, "Whodunit: Classifying code as human authored or gpt-4 generated-a case study on codechef problems," in Proceedings of the 21st International Conference on Mining Software Repositories, pp. 394–406, 2024.

[16] M. Akinwande, O. Adeliyi, and T. Yussuph, "Decoding ai and human authorship: Nuances revealed through nlp and statistical analysis," arXiv preprint arXiv:2408.00769, 2024.

[17] C. X. Liang, P. Tian, C. H. Yin, Y. Yua, W. An-Hou, L. Ming, T. Wang, Z. Bi, and M. Liu, "A comprehensive survey and guide to multimodal large language models in vision-language tasks," arXiv preprint arXiv:2411.06284, 2024.

[18] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification," in Proceedings of the 8th international symposium on visualization for cyber security, pp. 1–7, 2011.

[19] A. Gurioli, M. Gabbrielli, and S. Zacchiroli, "Is this you, llm? recognizing ai-written programs with multilingual code stylometry," in 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 394–405, IEEE, 2025.

[20] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, et al., "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," arXiv preprint arXiv:2105.12655, 2021.

VI. Appendix

Model	Accuracy	Precision	Recall	F1 Score
CodeFusion	1.00	1.00	1.00	1.00
CNN	0.92	0.96	0.88	0.92
Decision Tree	0.94	0.93	0.95	0.94
KNN	0.81	0.74	0.93	0.83
SVM	0.96	0.97	0.95	0.96
LSTM	0.94	0.95	0.93	0.94
Neural Network	0.79	0.73	0.95	0.83
CodeBERT	1.00	1.00	1.00	1.00
TFID-CNN	0.99	1.00	0.99	1.00
CNN SVM	0.86	0.88	0.86	0.86
Contrastive BERT CNN	1.00	1.00	1.00	1.00

TABLE I
Java 4o Results

Model	Accuracy	Precision	Recall	F1 Score
CodeFusion	0.91	0.89	0.95	0.92
CNN	0.92	0.91	0.92	0.92
Decision Tree	0.90	0.90	0.90	0.90
KNN	0.83	0.82	0.85	0.83
SVM	0.93	0.97	0.90	0.93
LSTM	0.90	0.88	0.92	0.90
Neural Network	0.76	0.98	0.54	0.70
CodeBERT	0.98	0.99	0.96	0.98
TFID-CNN	0.90	0.87	0.94	0.90
CNN SVM	0.88	0.88	0.88	0.88
Contrastive BERT CNN	0.91	0.93	0.88	0.91

TABLE II
Java 3.5 Turbo Results

Model	Accuracy	Precision	Recall	F1 Score
CodeFusion	0.82	0.87	0.76	0.81
CNN	0.65	0.65	0.67	0.66
CNN SVM	0.66	0.63	0.72	0.66
TFID-CNN	0.80	0.78	0.85	0.81
KNN	0.74	0.89	0.54	0.67
Decision Tree	0.77	0.80	0.72	0.76
NN	0.82	0.75	0.89	0.81
LSTM	0.80	0.76	0.87	0.81
CodeBERT	0.86	0.96	0.76	0.85
SVM	0.85	0.89	0.79	0.84
Contrastive BERT CNN	0.82	0.87	0.76	0.81

TABLE III
Python 3.5 Turbo Results

Model	Accuracy	Precision	Recall	F1 Score
CodeFusion	1.00	0.99	1.00	1.00
CNN SVM	0.75	0.74	0.76	0.75
TFID-CNN	0.96	0.94	0.99	0.96
CNN	0.79	0.77	0.82	0.79
LSTM	0.92	0.90	0.93	0.92
SVM	0.93	0.93	0.94	0.93
NN	0.90	0.92	0.86	0.89
KNN	0.82	0.78	0.90	0.84
CodeBERT	0.96	0.98	0.94	0.96
Decision Tree	0.91	0.90	0.93	0.91
Contrastive BERT CNN	1.00	1.00	0.99	1.00

TABLE IV
Model 4o Python Results

Model	Accuracy	Precision	Recall	F1 Score
CodeFusion	1.00	0.99	1.00	1.00
CNN SVM	0.76	0.76	0.76	0.76
TFID-CNN	0.97	0.97	0.97	0.97
CNN	0.80	0.82	0.76	0.78
LSTM	0.97	0.98	0.96	0.97
SVM	0.96	0.98	0.94	0.96
NN	0.88	0.88	0.73	0.84
KNN	0.82	0.76	0.95	0.84
CodeBERT	0.98	0.99	0.96	0.98
Decision Tree	0.93	0.93	0.94	0.94
Contrastive BERT CNN	1.00	1.00	1.00	1.00

TABLE V
Model 4o OCaml Results

Model	Accuracy	Precision	Recall	F1 Score
CodeFusion	0.95	0.95	0.95	0.95
CNN SVM	0.72	0.73	0.72	0.72
TFID-CNN	0.90	0.90	0.90	0.90
CNN	0.75	0.73	0.80	0.76
LSTM	0.92	0.91	0.94	0.92
SVM	0.90	0.89	0.92	0.91
NN	0.91	0.93	0.89	0.91
KNN	0.84	0.80	0.91	0.85
CodeBERT	0.96	0.98	0.94	0.96
Decision Tree	0.86	0.85	0.88	0.86
Contrastive BERT CNN	0.94	0.93	0.96	0.94

TABLE VI
Model 3.5 OCaml Results

[b]0.23

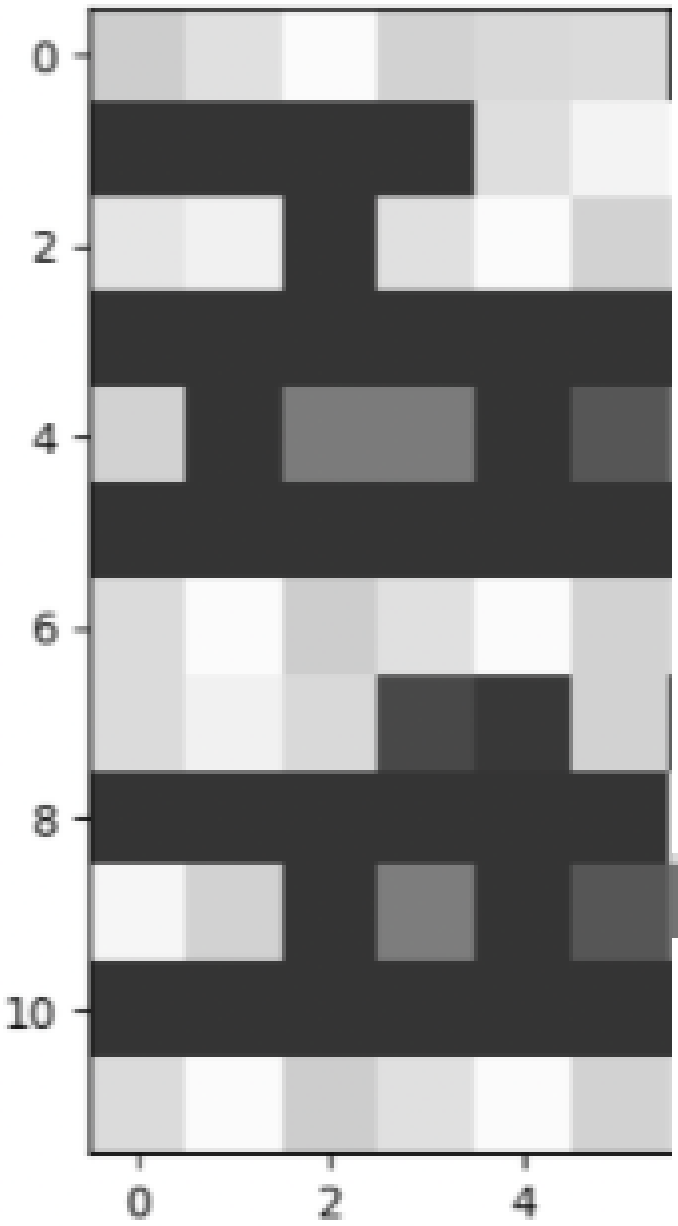
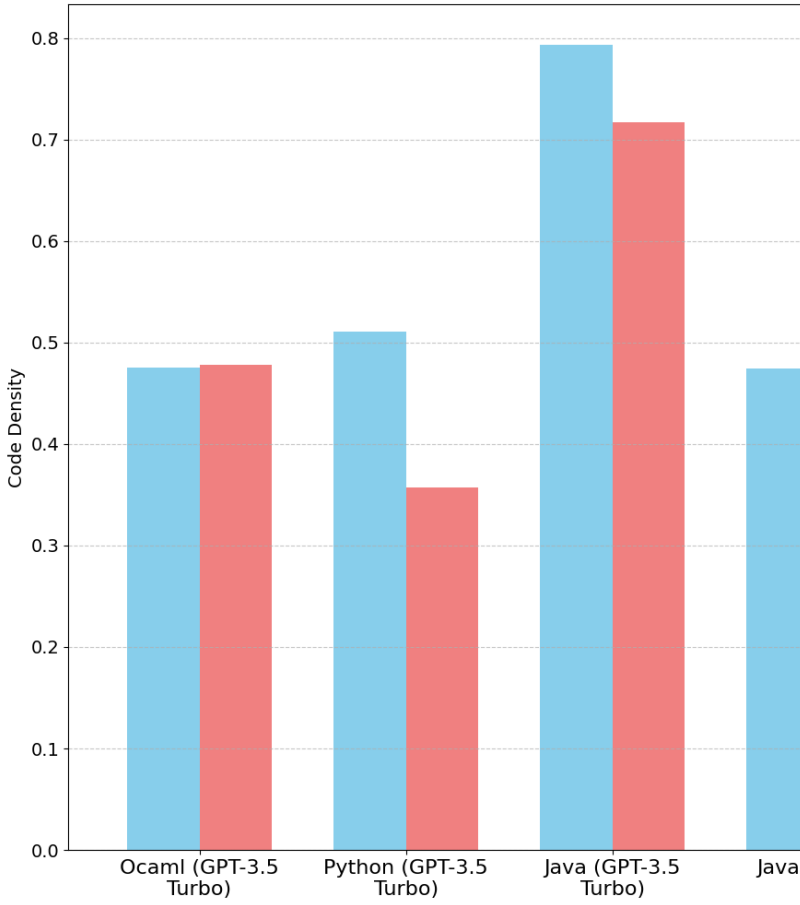


Fig. 3. Human-written code as grayscale.

[b]0.23

[t]0.23



Code density across all samples.

[t]0.23

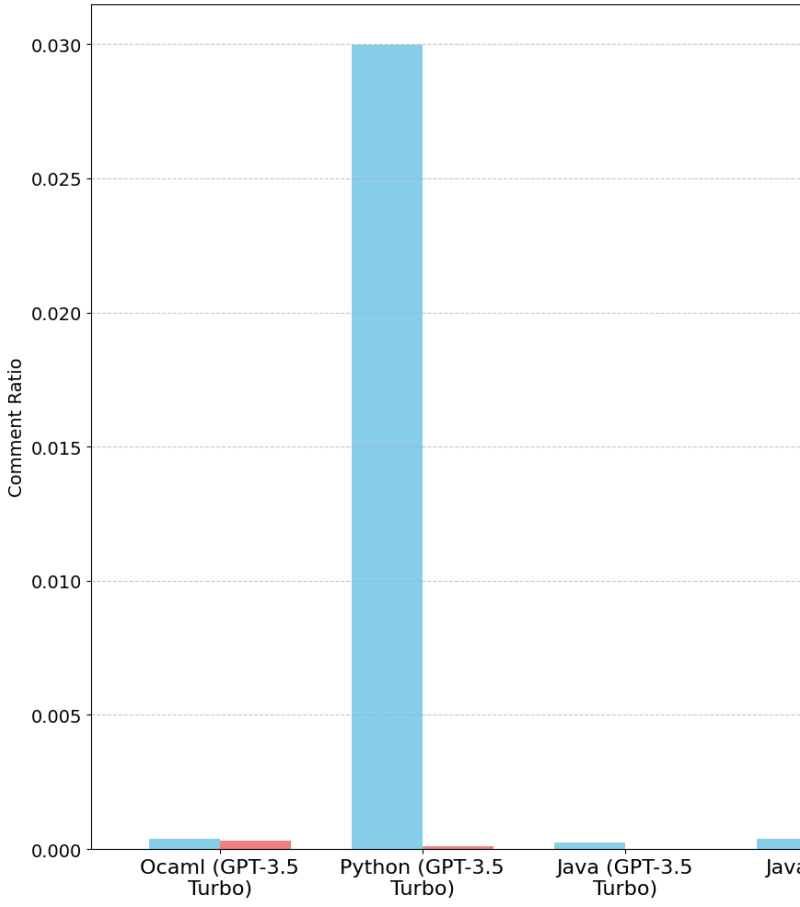


Fig. 7. Comment ratio across all samples

[t]0.32

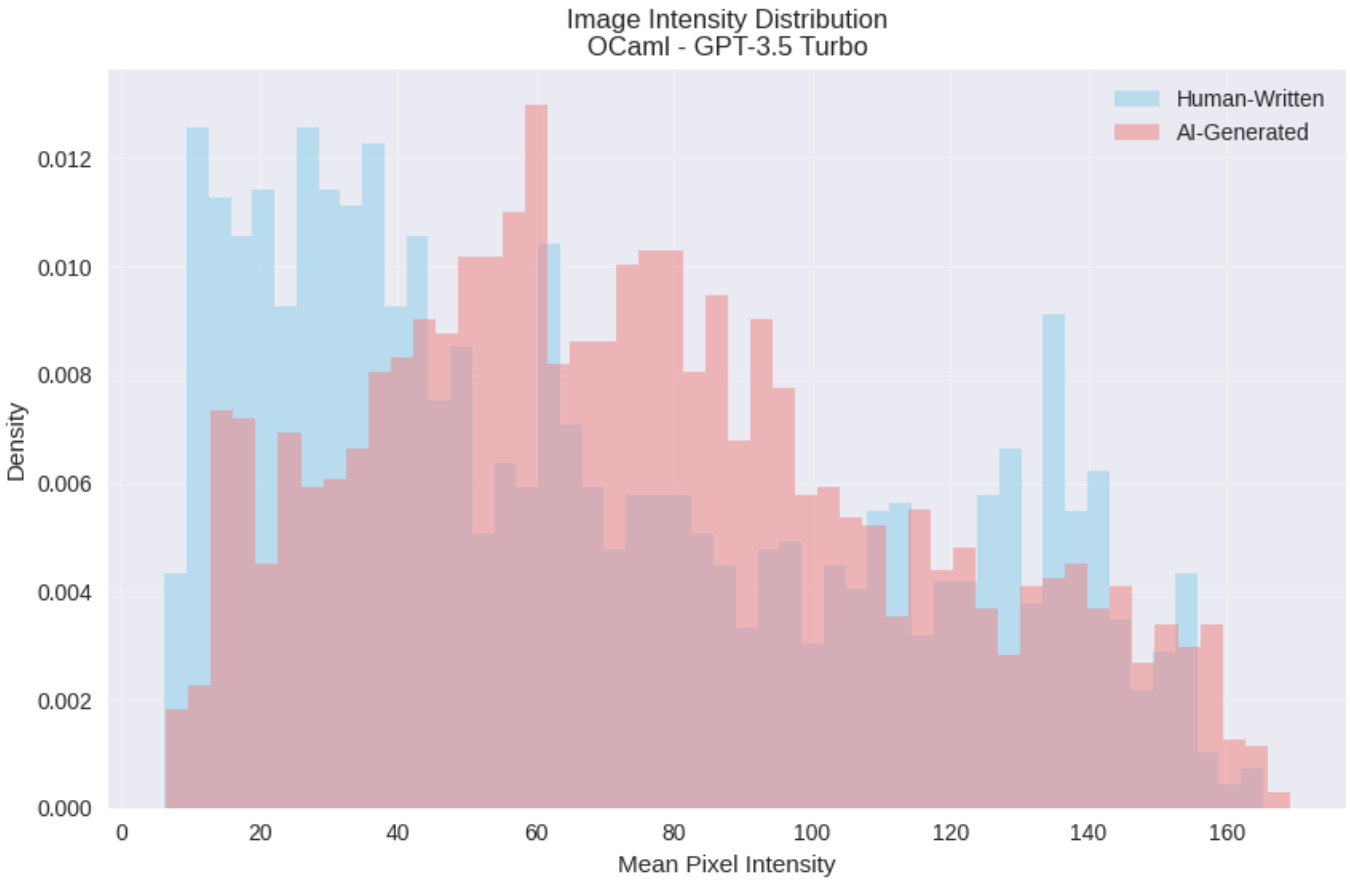


Fig. 11. OCaml GPT-3.5Turbo

[t]0.32

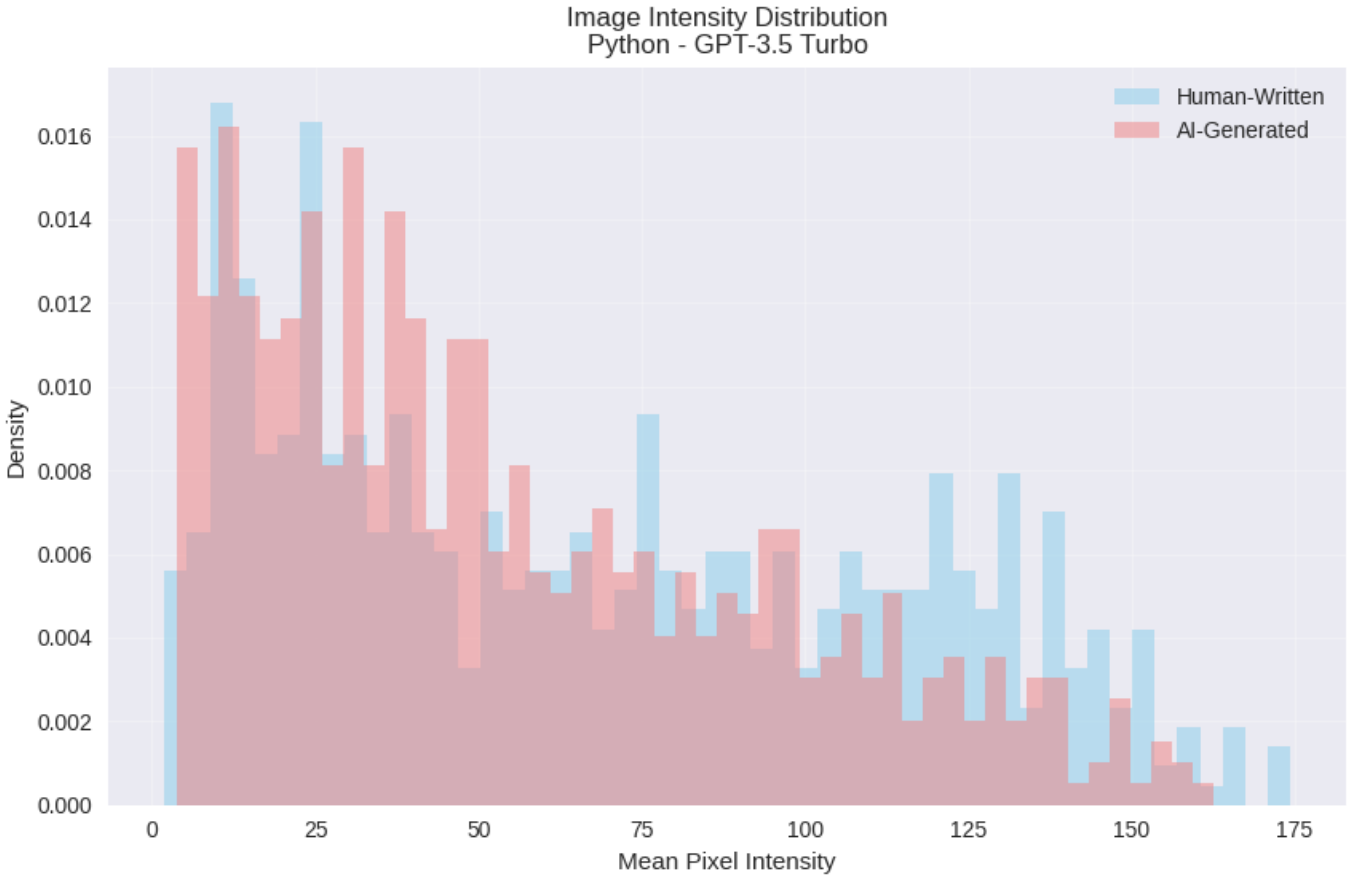


Fig. 12. Python GPT-3.5Turbo

[t]0.32

[t]0.23

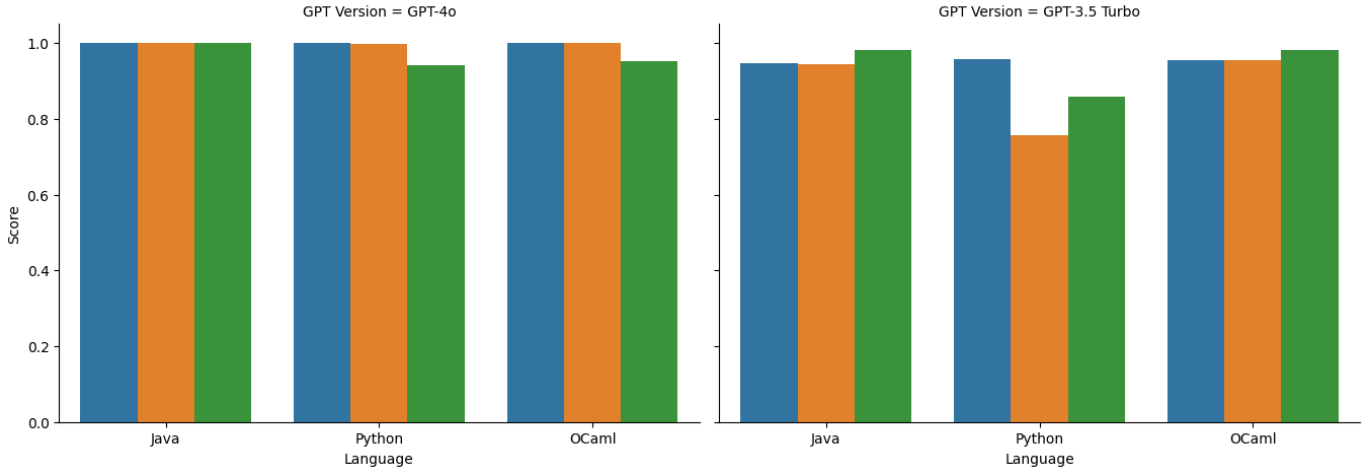


Fig. 18. Recall across models.

[t]0.23

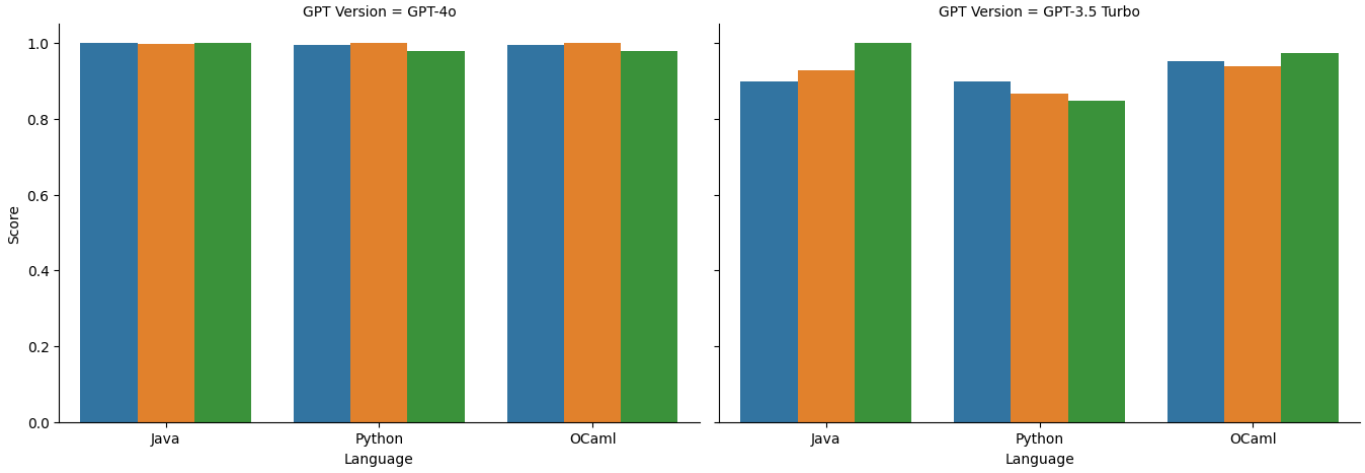


Fig. 19. Precision across models.

[t]0.23

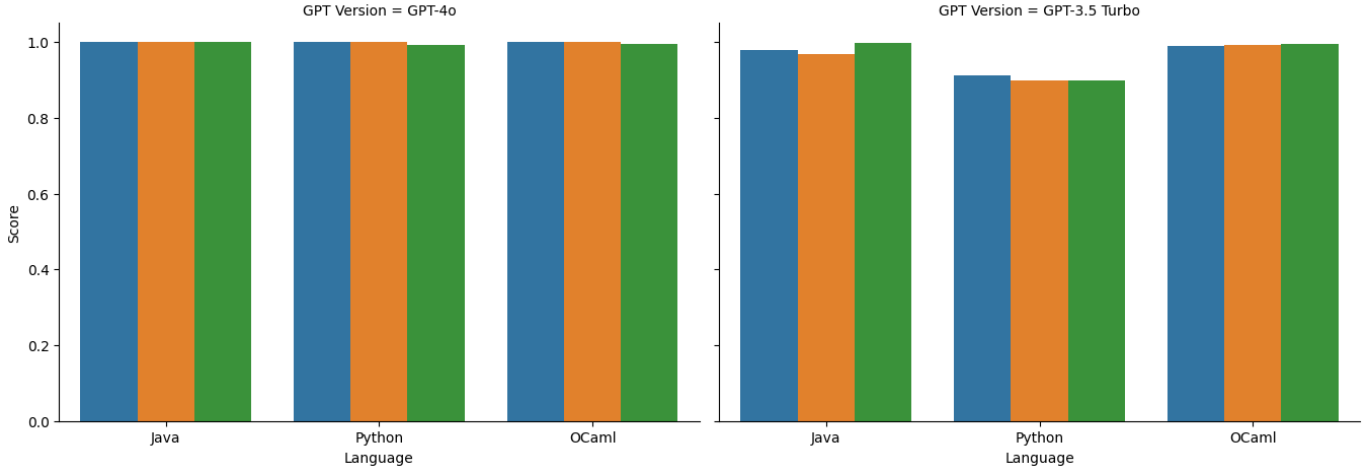


Fig. 20. ROC-AUC across models.