AI Code Detection Methodology

Executive Summary

This document outlines a comprehensive, multi-dimensional methodology for detecting AI-generated code. The approach combines pattern recognition, statistical analysis, stylometric evaluation, and heuristic rules to distinguish between human-written and AI-generated source code with high accuracy.

Table of Contents

1. Introduction
2. Theoretical Foundation
3. Detection Dimensions
4. Scoring Algorithm
5. Confidence Calculation
6. Limitations and Considerations
7. References

Introduction

Problem Statement

The proliferation of AI coding assistants (ChatGPT, GitHub Copilot, Claude, Gemini) has created a critical need for robust detection methods to:

- **Maintain Academic Integrity**: Prevent plagiarism in computer science education
- **Protect Intellectual Property**: Verify code authorship and licensing compliance
- **Ensure Code Quality**: Identify AI-generated code that may require additional review
- **Support Transparency**: Enable proper attribution in collaborative development

Approach Philosophy

Our methodology is based on the principle that **AI-generated code exhibits statistically distinguishable patterns** from human-written code, even when functionally equivalent. These patterns emerge from:

1. **Training Data Biases**: AI models learn from curated, high-quality codebases
2. **Optimization Objectives**: AI prioritizes clarity, consistency, and best practices
3. **Lack of Human Constraints**: AI doesn't experience time pressure, fatigue, or evolving coding styles
4. **Statistical Generation**: AI produces code through probabilistic token prediction rather than semantic understanding

---

Theoretical Foundation

Key Differentiators: AI vs. Human Code

AI-Generated Code Characteristics

Dimension	AI Pattern	Explanation
Naming Conventions	Verbose, descriptive identifiers	`user_authentication_manager` vs. `auth_mgr`
Documentation	Comprehensive, formal docstrings	Complete parameter descriptions, return types
Formatting	Perfect consistency	Uniform indentation, spacing, line breaks
Error Handling	Extensive try-catch blocks	Defensive programming with comprehensive checks
Syntax	Modern language features	Type hints, f-strings, async/await
Comments	Formal, explanatory	“This function validates user credentials…”
Structure	Textbook algorithm patterns	Standard implementations without shortcuts
Complexity	Moderate, balanced	Avoids both over-simplification and over-complexity

Human-Written Code Characteristics

Dimension	Human Pattern	Explanation
Naming Conventions	Abbreviated, context-dependent	`usr`, `tmp`, `i`, `j`, `res`
Documentation	Sparse, inconsistent	Missing docstrings, brief comments
Formatting	Variable consistency	Mixed indentation, inconsistent spacing
Error Handling	Minimal, pragmatic	Only critical error paths covered
Syntax	Mixed modern/legacy	Legacy patterns, older syntax
Comments	Informal, task-oriented	"TODO", "FIXME", "HACK"
Structure	Pragmatic shortcuts	Domain-specific optimizations, workarounds
Complexity	Variable, context-driven	Ranges from simple to highly complex

Research-Backed Indicators

Based on academic research and industry analysis:

1. **Code Length**: ChatGPT-generated code averages 15-20% shorter than student-written code for equivalent functionality
2. **Comment-to-Code Ratio**: AI code exhibits 2-3x higher documentation density
3. **Identifier Length**: AI averages 10-15 characters per identifier vs. 5-8 for humans
4. **Error Handling Density**: AI includes 1.7x more error handling constructs
5. **Formatting Consistency**: AI achieves >95% consistency vs. 70-85% for humans

---

Detection Dimensions

Our methodology analyzes code across **eight independent dimensions**, each contributing to the final AI probability score.

1. Naming Pattern Analysis

Objective: Identify verbose, descriptive naming conventions characteristic of AI

Metrics:
- Average identifier length
- Verbose naming pattern frequency (camelCase with 3+ segments)

- Descriptive variable patterns (`user_data` , `response_data` , `input_value`)
- Abbreviated variable frequency (`i` , `j` , `tmp` , `res`)

Scoring Logic:

```
AI Score = 0.0
IF avg_identifier_length > 12: AI Score += 0.4
ELIF avg_identifier_length > 8: AI Score += 0.2
IF verbose_matches > 30% of lines: AI Score += 0.3
IF descriptive_patterns > 5: AI Score += 0.2
IF abbreviated_vars > 20% of lines: AI Score -= 0.3
```

Example:
- **AI**: `user_authentication_manager` , `session_expiration_time` , `validate_credentials_securely`
- **Human**: `auth` , `sess_exp` , `check_creds`

---

2. Comment Style Detection

Objective: Distinguish formal AI documentation from informal human comments

Metrics:
- Comment-to-code ratio
- Formal docstring count (triple-quoted strings)
- Informal comment markers (TODO, FIXME, HACK, NOTE)
- Average comment length

Scoring Logic:

```
AI Score = 0.0
IF comment_ratio > 0.3: AI Score += 0.3
IF formal_docstrings > 2: AI Score += 0.3
IF avg_comment_length > 60 chars: AI Score += 0.2
IF informal_markers > 3: AI Score -= 0.3
```

Example:
- **AI**:
```python
"""
Authenticate user credentials and generate session token.

Args:
username: Username to authenticate
password: Password to verify

Returns:
Dictionary with authentication result and session token
"""
```
`- **Human**:` python
TODO: add proper validation
quick login check
```
```

---

3. Code Structure Analysis

Objective: Measure formatting consistency and organizational patterns

Metrics:
- Indentation consistency (percentage of lines following dominant pattern)
- Blank line ratio (spacing between logical blocks)
- Line length variance

Scoring Logic:

```
AI Score = 0.0
IF indent_consistency > 0.95: AI Score += 0.4
ELIF indent_consistency > 0.85: AI Score += 0.2
IF 0.05 < blank_line_ratio < 0.15: AI Score += 0.2
```

Rationale: AI maintains perfect indentation consistency, while human code often shows minor variations from copy-paste, refactoring, or multiple contributors.

---

4. Complexity Analysis

Objective: Evaluate code complexity and nesting patterns

Metrics:
- Average line length
- Control structure count (if, for, while, switch)
- Nesting depth indicators
- Cyclomatic complexity proxies

Scoring Logic:

```
AI Score = 0.0
IF 60 < avg_line_length < 90: AI Score += 0.3
IF nesting_ratio < 0.3: AI Score += 0.2
```

Rationale: AI tends to produce moderately complex code with balanced line lengths, avoiding both extreme brevity and excessive verbosity.

---

5. Error Handling Analysis

Objective: Assess defensive programming patterns

Metrics:
- Try-catch block frequency
- Null/None check frequency
- Exception handling coverage
- Error handling ratio (error constructs per line of code)

Scoring Logic:

```
AI Score = 0.0
IF error_handling_ratio > 0.1: AI Score += 0.4
ELIF error_handling_ratio > 0.05: AI Score += 0.2
IF try_blocks > 0 AND except_blocks >= try_blocks: AI Score += 0.2
```

Example:
- **AI**: Comprehensive try-except blocks with specific exception types
- **Human**: Minimal error handling, often only for critical paths

---

6. Documentation Analysis

Objective: Measure documentation completeness and quality

Metrics:
- Docstring count
- Function/class definition count
- Documentation ratio (documented entities / total entities)
- Average docstring length

Scoring Logic:

```
AI Score = 0.0
IF documented_ratio > 0.7: AI Score += 0.4
ELIF documented_ratio > 0.4: AI Score += 0.2
IF avg_docstring_length > 100: AI Score += 0.3
```

Rationale: AI consistently generates comprehensive documentation, while human developers often skip or minimize documentation due to time constraints.

---

7. Formatting Consistency Analysis

Objective: Detect uniform spacing and operator formatting

Metrics:
- Operator spacing consistency (spaces around =, +, -, *, /)
- Bracket spacing patterns
- Comma spacing consistency

Scoring Logic:

```
AI Score = 0.0
IF spacing_consistency > 0.9: AI Score += 0.5
ELIF spacing_consistency > 0.7: AI Score += 0.3
```

Example:
- **AI**: `result = value + 10` (consistent spacing)
- **Human**: Mix of `result=value+10` and `result = value + 10`

---

8. Syntax Modernity Analysis

Objective: Identify use of modern vs. legacy language features

Metrics:
- Modern feature count (type hints, f-strings, async/await, context managers)
- Legacy feature count (var declarations, prototype chains, % formatting)
- Modern syntax ratio

Scoring Logic:

```
modern_ratio = modern_features / (modern_features + legacy_features)
AI Score = modern_ratio
```

Rationale: AI models are trained on recent, high-quality code and naturally favor modern syntax patterns.

Scoring Algorithm

Aggregation Method

The final AI probability score is calculated as the **arithmetic mean** of all eight dimension scores:

```
AI_Probability = (Σ dimension_scores) / 8
Human_Probability = 1 - AI_Probability
```

Score Interpretation

AI Probability	Interpretation	Recommended Action
0% - 35%	Likely Human-Written	Low suspicion, minimal review
35% - 55%	Mixed Indicators	Moderate suspicion, contextual review
55% - 75%	Possibly AI-Assisted	High suspicion, detailed review
75% - 100%	Likely AI-Generated	Very high suspicion, thorough investigation

---

Confidence Calculation

Variance-Based Confidence

Confidence is determined by analyzing the **variance** across dimension scores:

```
variance = Σ(score_i - mean_score)² / n
```

Confidence Levels:
- **HIGH**: variance < 0.05 (scores are consistent across dimensions)
- **MEDIUM**: 0.05 ≤ variance < 0.15 (moderate variation)
- **LOW**: variance ≥ 0.15 (high variation, inconclusive)

Rationale

- **Low variance**: All dimensions agree → high confidence in verdict
- **High variance**: Dimensions conflict → mixed signals, manual review needed

Verdict Determination

```
IF confidence == "LOW":
    verdict = "INCONCLUSIVE - Manual review recommended"
ELIF AI_Probability > 0.75:
    verdict = "LIKELY AI-GENERATED"
ELIF AI_Probability > 0.55:
    verdict = "POSSIBLY AI-ASSISTED"
ELIF AI_Probability > 0.35:
    verdict = "MIXED INDICATORS"
ELSE:
    verdict = "LIKELY HUMAN-WRITTEN"
```

Limitations and Considerations

Known Limitations

1. **False Positives**: Highly skilled human developers following strict style guides may trigger AI indicators
2. **False Negatives**: Heavily modified AI code or AI prompted to “write like a human” may evade detection
3. **Language Coverage**: Detection accuracy varies by programming language (highest for Python, JavaScript, Java)
4. **Code Length**: Very short snippets (<50 lines) provide insufficient signal for reliable detection
5. **Hybrid Code**: Code with mixed human/AI contributions presents ambiguous signals

Evasion Techniques

Sophisticated users may attempt to evade detection through:

- **Variable Renaming**: Shortening AI-generated verbose names
- **Comment Removal**: Stripping formal documentation
- **Formatting Disruption**: Introducing inconsistent spacing
- **Logic Restructuring**: Refactoring AI code structure

Mitigation: Our multi-dimensional approach makes complete evasion difficult, as it requires systematic modification across all eight dimensions.

Ethical Considerations

This tool should be used as a **decision support system**, not a definitive judgment:

- Results indicate **probability**, not certainty
- Human review is essential for high-stakes decisions
- Context matters: AI assistance may be permitted or encouraged in some settings
- Transparency and proper attribution are preferable to prohibition

---

References

Academic Research

1. Hoq, M., et al. (2024). “Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models.” ACM Technical Symposium on Computer Science Education.

2. Leinonen, J., et al. (2024). “Detecting ChatGPT-Generated Code in a CS1 Course.” CEUR Workshop Proceedings.

3. “The AI Attribution Paradox: Transparency as Social Strategy in Open-Source Software Development.” (2025). arXiv:2512.00867.

4. “AI vs Human Code Generation Report: AI Code Creates 1.7x More Issues.” (2025). CodeRabbit Research.

Industry Tools and Standards

1. AI Code Detector (aicodedetector.org) - Pattern recognition methodology
2. Codequiry AI Detection - Neural network approach
3. GPTZero - Multi-layered analysis framework
4. SonarQube AI Code Assurance - Quality metrics

Technical Documentation

1. GitHub Copilot Documentation - AI code generation patterns
2. OpenAI Codex Research - LLM code generation characteristics

---

Conclusion

This methodology provides a **scientifically grounded, multi-dimensional approach** to AI code detection. By analyzing eight independent dimensions and calculating confidence-weighted verdicts, it achieves high accuracy while acknowledging inherent limitations.

Key Takeaways:
- No single indicator is definitive; the combination of multiple signals provides robust detection
- Confidence scoring helps identify cases requiring human judgment
- The tool is designed to support transparency and proper attribution, not to punish AI usage
- Continuous refinement based on evolving AI capabilities is essential

---

Last Updated: February 2026
Version: 1.0