AI Code Detector - Comprehensive Usage Guide

Table of Contents

Getting Started

Prerequisites

- Python 3.7 or higher
- No external dependencies required

Installation

```
# Navigate to the tool directory
cd /home/ubuntu/ai_code_detector

# Verify Python version
python --version

# Test the tool
python ai_code_detector.py --help
```

Basic Usage Examples

Example 1: Analyze a Single File

```
python ai_code_detector.py my_script.py
```

Output:

```
================================================================================
File: my_script.py
================================================================================
AI Probability:    65.3%
Human Probability: 34.7%
Confidence:        MEDIUM
Verdict:           POSSIBLY AI-ASSISTED

Key Indicators:
  • Verbose Naming: True
  • High Documentation: True
```

Example 2: Analyze Multiple Files

```
python ai_code_detector.py file1.py file2.js file3.java
```

This will analyze all three files sequentially and display results for each.

Example 3: Analyze an Entire Directory

```
python ai_code_detector.py --directory ./src
```

This recursively analyzes all supported code files in the `./src` directory.

Example 4: Get Detailed Analysis

```
python ai_code_detector.py script.py --format detailed
```

Output includes:
- Overall AI/Human probabilities
- Confidence level
- Verdict
- Detailed scores for all 8 dimensions
- Specific metrics for each dimension

Example 5: Export Results to JSON

```
python ai_code_detector.py --directory ./src --output results.json
```

Creates a machine-readable JSON file with all analysis results.

Example 6: Custom File Extensions

```
python ai_code_detector.py --directory ./project --extensions .py,.js,.ts,.jsx
```

Analyzes only files with specified extensions.

Advanced Usage

Batch Processing with Shell Scripts

Create a script to analyze multiple projects:

```
#!/bin/bash
# analyze_projects.sh

for project in project1 project2 project3; do
    echo "Analyzing $project..."
    python ai_code_detector.py \
        --directory ./$project/src \
        --output ${project}_results.json \
        --format detailed
done
```

Integration with Git Hooks

Create a pre-commit hook to analyze staged files:

```
#!/bin/bash
# .git/hooks/pre-commit

STAGED_FILES=$(git diff --cached --name-only --diff-filter=ACM | grep -E '\.(py|js|
java)$')

if [ -n "$STAGED_FILES" ]; then
    python /path/to/ai_code_detector.py $STAGED_FILES --format summary

    # Optional: Fail commit if AI probability is too high
    # Add logic here to parse results and exit 1 if needed
fi
```

Continuous Integration (CI/CD)

GitHub Actions Example

```yaml
name: AI Code Detection

on: [pull_request]

jobs:
  detect-ai-code:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.9'

      - name: Run AI Code Detector
        run: |
          python ai_code_detector.py \
            --directory ./src \
            --output ai_detection_results.json

      - name: Upload Results
        uses: actions/upload-artifact@v2
        with:
          name: ai-detection-results
          path: ai_detection_results.json
```

Programmatic Usage (Python API)

```python
from ai_code_detector import AICodeDetector

# Initialize detector
detector = AICodeDetector()

# Analyze a file
result = detector.analyze_file('script.py')

# Access results
print(f"AI Probability: {result.ai_probability}%")
print(f"Verdict: {result.verdict}")
print(f"Confidence: {result.confidence}")

# Access detailed scores
for dimension, scores in result.detailed_scores.items():
    print(f"{dimension}: {scores['ai_indicators']}")

# Check specific indicators
if result.indicators.get('verbose_naming'):
    print("Detected verbose naming patterns")
```

Interpreting Results

Understanding AI Probability

The AI probability represents the likelihood that the code was generated by an AI system:

```
AI Probability = (Sum of 8 dimension scores) / 8 × 100%
```

Probability Ranges

Range	Meaning	Action
0-20%	Very likely human	No action needed
20-35%	Likely human	Minimal review
35-55%	Uncertain	Contextual review required
55-75%	Possibly AI-assisted	Detailed review recommended
75-90%	Likely AI-generated	Thorough investigation needed
90-100%	Very likely AI-generated	Strong evidence of AI generation

Understanding Confidence Levels

Confidence indicates how consistent the signals are across all dimensions:

HIGH Confidence

- All 8 dimensions agree on the verdict
- Low variance in scores (< 0.05)
- **Interpretation**: The verdict is highly reliable

Example: All dimensions score between 0.75-0.85 → HIGH confidence in “AI-generated” verdict

MEDIUM Confidence

- Moderate variation across dimensions
- Variance between 0.05-0.15
- **Interpretation**: The verdict is probable but not certain

Example: Some dimensions score 0.6, others 0.8 → MEDIUM confidence

LOW Confidence

- High variation across dimensions
- Variance ≥ 0.15
- **Interpretation**: Mixed signals, manual review essential

Example: Some dimensions score 0.3, others 0.9 → LOW confidence, inconclusive

Key Indicators Explained

Verbose Naming

- **Trigger**: Average identifier length > 10 characters
- **Example**: `user_authentication_manager` vs. `auth_mgr`
- **Significance**: AI tends to use descriptive, self-documenting names

High Documentation

- **Trigger**: >70% of functions/classes have docstrings
- **Example**: Every function has a comprehensive docstring
- **Significance**: AI consistently generates formal documentation

Perfect Formatting

- **Trigger**: >90% consistency in indentation and spacing
- **Example**: Uniform spacing around all operators
- **Significance**: AI maintains perfect consistency, humans vary

Comprehensive Error Handling

- **Trigger**: >8% of lines are error handling constructs
- **Example**: Try-catch blocks for most operations
- **Significance**: AI implements defensive programming extensively

Modern Syntax Heavy

- **Trigger**: >80% modern language features
- **Example**: Type hints, f-strings, async/await
- **Significance**: AI trained on recent, high-quality code

Integration Workflows

Workflow 1: Academic Integrity Checking

Scenario: Computer science instructor reviewing student submissions

```
# Step 1: Collect all student submissions
mkdir student_submissions
# (students submit their code here)

# Step 2: Run batch analysis
python ai_code_detector.py \
    --directory student_submissions \
    --output class_analysis.json \
    --format detailed

# Step 3: Review results
# - Focus on submissions with >70% AI probability
# - Check HIGH confidence cases first
# - Manually review MEDIUM confidence cases

# Step 4: Follow up
# - Discuss findings with students showing high AI probability
# - Use as teaching moment about proper AI usage and attribution
```

Workflow 2: Code Review in Pull Requests

Scenario: Development team reviewing contributions

```
# Step 1: Checkout the PR branch
git checkout pr-branch

# Step 2: Analyze changed files
git diff --name-only main...pr-branch | \
    grep -E '\.(py|js|java)$' | \
    xargs python ai_code_detector.py --format summary

# Step 3: Review results
# - AI probability >60%: Request explanation from contributor
# - Check if AI usage aligns with team policy
# - Ensure proper testing and review of AI-generated code

# Step 4: Document findings
# - Add comment to PR with detection results
# - Request attribution if AI was used
# - Approve or request changes based on policy
```

Workflow 3: Repository Audit

Scenario: Analyzing an entire codebase for AI-generated content

```
# Step 1: Clone repository
git clone https://github.com/org/repo.git
cd repo

# Step 2: Run comprehensive analysis
python ai_code_detector.py \
    --directory ./src \
    --output repo_audit.json \
    --format detailed

# Step 3: Generate summary report
python -c "
import json
with open('repo_audit.json') as f:
    results = json.load(f)

total = len(results)
high_ai = sum(1 for r in results if r['ai_probability'] > 70)
medium_ai = sum(1 for r in results if 40 < r['ai_probability'] <= 70)
low_ai = sum(1 for r in results if r['ai_probability'] <= 40)

print(f'Total files: {total}')
print(f'High AI probability (>70%): {high_ai} ({high_ai/total*100:.1f}%)')
print(f'Medium AI probability (40-70%): {medium_ai} ({medium_ai/total*100:.1f}%)')
print(f'Low AI probability (<40%): {low_ai} ({low_ai/total*100:.1f}%)')
"

# Step 4: Investigate high-probability files
# - Review files with >70% AI probability
# - Check commit history for attribution
# - Verify code quality and testing
```

Workflow 4: Monitoring AI Adoption Over Time

Scenario: Tracking AI usage trends in a project

```
#!/bin/bash
# monitor_ai_usage.sh

# Analyze current state
python ai_code_detector.py \
    --directory ./src \
    --output "analysis_$(date +%Y%m%d).json"

# Compare with previous analysis
python -c "
import json
from datetime import datetime

# Load current and previous results
with open('analysis_$(date +%Y%m%d).json') as f:
    current = json.load(f)

# Calculate average AI probability
avg_ai = sum(r['ai_probability'] for r in current) / len(current)

print(f'Date: {datetime.now().strftime(\"%Y-%m-%d\")}')
print(f'Files analyzed: {len(current)}')
print(f'Average AI probability: {avg_ai:.1f}%')
print(f'High AI files (>70%): {sum(1 for r in current if r[\"ai_probability\"] >
70)}')
"
```

Troubleshooting

Issue 1: “Unable to analyze” Error

Symptom: File shows “Unable to analyze” verdict

Causes:
- File encoding issues (non-UTF-8)
- Binary file mistakenly analyzed
- File permissions

Solution:

```
# Check file encoding
file -i problematic_file.py

# Convert to UTF-8 if needed
iconv -f ISO-8859-1 -t UTF-8 problematic_file.py > fixed_file.py

# Check permissions
ls -l problematic_file.py
chmod 644 problematic_file.py
```

Issue 2: Unexpected Low/High Scores

Symptom: Results don't match expectations

Causes:
- Very short code files (<50 lines)
- Mixed human/AI code
- Unusual coding style

Solution:
- Use `--format detailed` to see dimension-by-dimension breakdown
- Check confidence level (LOW confidence = inconclusive)
- Manually review the code
- Consider context (e.g., generated boilerplate is expected)

Issue 3: Performance Issues with Large Codebases

Symptom: Analysis takes too long

Solution:

```
# Analyze in batches
find ./src -name "*.py" | head -100 | xargs python ai_code_detector.py

# Use parallel processing (GNU parallel)
find ./src -name "*.py" | parallel -j 4 python ai_code_detector.py {}

# Filter by recently modified files
find ./src -name "*.py" -mtime -7 | xargs python ai_code_detector.py
```

Issue 4: JSON Output Parsing

Symptom: Difficulty processing JSON results

Solution:

```
import json

# Load results
with open('results.json') as f:
    results = json.load(f)

# Filter high AI probability files
high_ai_files = [
    r for r in results
    if r['ai_probability'] > 70 and r['confidence'] == 'HIGH'
]

# Sort by AI probability
sorted_results = sorted(results, key=lambda x: x['ai_probability'], reverse=True)

# Export to CSV
import csv
with open('results.csv', 'w', newline='') as f:
    writer = csv.DictWriter(f, fieldnames=['file_path', 'ai_probability', 'verdict', '
confidence'])
    writer.writeheader()
    for r in results:
        writer.writerow({
            'file_path': r['file_path'],
            'ai_probability': r['ai_probability'],
            'verdict': r['verdict'],
            'confidence': r['confidence']
        })
```

Best Practices

For Educators

1. Set Clear Expectations

```
Before using the tool:
- Define acceptable AI usage in your course policy
- Explain how AI detection will be used
- Focus on learning outcomes, not punishment
```

2. Use as a Conversation Starter

```
When high AI probability is detected:
- Discuss findings with student privately
- Ask about their development process
- Use as teaching moment about AI ethics and attribution
- Focus on understanding, not accusation
```

3. Combine with Other Evidence

```
Don't rely solely on detection tool:
- Review commit history
- Conduct code walkthroughs
- Ask students to explain their code
- Look for consistency across assignments
```

For Development Teams

1. Establish AI Usage Policy

```
Example Policy:
- AI assistance is permitted for boilerplate and routine tasks
- All AI-generated code must be reviewed and tested by humans
- AI usage must be disclosed in commit messages or PR descriptions
- Critical security and business logic should be human-authored
```

2. Integrate into Code Review

```
Code review checklist:
□ Run AI detection on changed files
□ If AI probability >60%, verify:
  □ Proper testing coverage
  □ Security review completed
  □ Performance validated
  □ Documentation updated
□ AI usage disclosed in PR description
```

3. Monitor Trends, Not Individuals

```
Focus on:
- Overall team AI adoption trends
- Code quality metrics over time
- Areas where AI assistance is most beneficial
- Training needs for effective AI tool usage
```

For Researchers

1. Collect Baseline Data

```
# Establish baseline before AI tool adoption
python ai_code_detector.py \
    --directory ./project \
    --output baseline_2023.json

# Compare after AI tool introduction
python ai_code_detector.py \
    --directory ./project \
    --output post_ai_2024.json
```

2. Control for Confounding Variables

```
Consider:
- Team composition changes
- Coding standard updates
- Language version upgrades
- Project complexity evolution
```

3. Validate Results

```
Validation methods:
- Manual review of sample files
- Cross-reference with known AI/human code
- Compare with other detection tools
- Analyze false positive/negative rates
```

Advanced Tips

Tip 1: Custom Thresholds

Modify the tool to use custom thresholds for your context:

```python
# In ai_code_detector.py, modify _determine_verdict method:

def _determine_verdict(self, ai_score: float, confidence: str) -> str:
    # Custom thresholds for strict academic setting
    if confidence == "LOW":
        return "INCONCLUSIVE - Manual review required"

    if ai_score > 0.65:  # Lowered from 0.75
        return "LIKELY AI-GENERATED"
    elif ai_score > 0.45:  # Lowered from 0.55
        return "POSSIBLY AI-ASSISTED"
    elif ai_score > 0.30:  # Lowered from 0.35
        return "MIXED INDICATORS"
    else:
        return "LIKELY HUMAN-WRITTEN"
```

Tip 2: Language-Specific Analysis

Focus on specific languages:

```bash
# Python only
python ai_code_detector.py --directory ./src --extensions .py

# JavaScript/TypeScript only
python ai_code_detector.py --directory ./src --extensions .js,.ts,.jsx,.tsx

# Backend languages
python ai_code_detector.py --directory ./src --extensions .py,.java,.go
```

Tip 3: Differential Analysis

Compare two versions of the same codebase:

```
# Analyze version 1
git checkout v1.0
python ai_code_detector.py --directory ./src --output v1_results.json

# Analyze version 2
git checkout v2.0
python ai_code_detector.py --directory ./src --output v2_results.json

# Compare results
python -c "
import json

with open('v1_results.json') as f:
    v1 = {r['file_path']: r['ai_probability'] for r in json.load(f)}

with open('v2_results.json') as f:
    v2 = {r['file_path']: r['ai_probability'] for r in json.load(f)}

# Find files with increased AI probability
for file in v2:
    if file in v1:
        diff = v2[file] - v1[file]
        if diff > 20:  # 20% increase
            print(f'{file}: +{diff:.1f}% (v1: {v1[file]:.1f}%, v2: {v2[file]:.1f}%)')
"
```

Conclusion

This tool is designed to support transparency, quality, and proper attribution in software development. Use it as a decision support system, not a definitive judgment mechanism. Always combine automated detection with human review and contextual understanding.

For technical details, see METHODOLOGY.md (METHODOLOGY.md).
For general information, see README.md (README.md).

Last Updated: February 2026
Version: 1.0