DETECTING AI-GENERATED PYTHON CODE VIA MACHINE LEARNING

PROJECT REPORT FOR AML COURSE S25 INNOPOLIS UNIVERSITY

Dmitry Beresnev
AIDS-MS1, Innopolis University
d.beresnev@innopolis.university

Vsevolod Klyushev
AIDS-MS1, Innopolis University
v.klyushev@innopolis.university

Nikita Yaneev
AIDS-MS1, Innopolis University
n.yaneev@innopolis.university

1 Motivation

Rapid advancement of artificial intelligence led to the widespread use of large language models in code generation, particularly in programming competitions. Although AI-generated code could help participants, it also raised concerns about fairness and originality in contests where human problem solving skills were evaluated. Detecting AI-generated code had to become an important task in ensuring the integrity of programming competitions.

Traditional methods of plagiarism detection were often insufficient for identifying AI-generated code, as LLMs could produce highly varied and syntactically correct solutions that differed from human-written code in subtle but detectable ways. Machine learning (ML) approaches offered a promising solution by analyzing patterns in code structure, style, and other latent features that distinguished machine-generated code from human-written code.

In this project, we explored different ML approaches to detect AI-generated Python code, because this was the most popular programming language. We compared two main strategies:

- Pre-trained LLM-based detection — leveraging a powerful but computationally heavy model to classify code based on deep semantic features
- Tree-based models — utilizing lightweight decision trees or ensemble methods for faster inference and light weight, which is crucial in time-sensitive evaluation scenarios.

Our goal is to evaluate the trade-offs between accuracy and computational efficiency, ensuring that the solution is both reliable and practical for real-world programming competitions conducted on Accept [1]. Using these approaches, our aim is to develop a robust detection system that can help maintain fairness in coding contests while optimizing performance for large-scale use.

2 Related work

The detection of AI-generated code has gained significant attention with the recent rise of large language models (LLMs) like ChatGPT, Gemini, and GitHub Copilot. Previous studies have explored the detection of machine-generated text using stylometric and statistical techniques, often focusing on natural language (e.g. [6]). Suh et al. proposed GPTSniffer, a fine-tuned CodeBERT model for Java code detection, but its ability to generalize to other languages such as Python remained limited [7]. Additional work by Choi et al. investigated the attribution of authorship of source code through LLMs [2]. However, most of these studies have focused on general-purpose detection without specialization for Python or the nuances of specific model requirements. Our work extends this line of research by developing a targeted machine learning pipeline to identify generated Python code, considering the constraints on time consumption.

3 Data generation

First of all, the dataset of both human-written and AI-generated codes should be created. The final dataset include 12427 labeled code snippets.

3.1 Human-written entities

As human-written code snippets, anonymized Python solutions of Accept [1] platform users are used. Only valid solutions were used: for example, codes that resulted in a compilation error were omitted. In total, 5951 human-written solutions were included into dataset.

3.2 AI-generated entities

To generate AI-plagiarized code snippets, we decided to use the following LLM models:

- Evil
- Llama-3.2-3b
- BLACKBOX.AI
- DeepSeek

The tasks from Accept [1] (solutions of which resulted in human-written codes) were added to the following prompt and then passed to the LLM:

> *Write a Python solution for the following task. The code should look like it was written by an intermediate student: practical but not overly optimized or perfect. Follow these guidelines:*
> *1. Use not overly long names (e.g., res instead of result or final_output_value)*
> *2. Do not include comments or explanations*
> *3. Avoid using functions, prefer straightforward logic*
> *4. Apply small stylistic deviations, like mixing single/double quotes, occasional redundant logic, inconsistent spacing, etc*
> *5. No error handling*
> *6. Do not print to output anything except answer to the problem without any annotations*
> *Finally, return just pure python code*

Therefore, in total there are 6477 AI-generated code snippets.

4 LLM approach

We decided to check several LLM models to solve our problem - DeBERTaV3 and CodeBERT. To use them, we added a linear layer with 1 output and sigmoid activation function to predict the probability of text to be AI generated. We used Adam optimizer with learning rate 2e-5 and trained them for all models within 2 epochs achieving decent performance.

4.1 DeBERTaV3

DeBERTa [4] improves the BERT and RoBERTa models using disentangled attention and improved mask decoder. With those two improvements, DeBERTa outperforms RoBERTa in most natural language understanding (NLU) tasks.

4.2 CodeBERT

CodeBERT [3] — bimodal pre-trained model for programming language (PL) and natural language (NL). CodeBERT learns general-purpose representations that support downstream NL-PL applications such as natural language code search, code documentation generation, etc.

5 AST approach

An Abstract Syntax Tree (AST) is a finite labeled oriented tree in which the inner vertices are mapped to programming language operators and the leaves to their corresponding operands.

To distinguish between human-written and AI-generated Python code, we employ an abstract syntax tree (AST)-based representation that captures the structural and syntactic patterns of the code. Unlike raw text-based or token-based methods, ASTs provide a more robust and hierarchical representation of code, making them suitable for detecting subtle differences in coding styles between humans and AI models.

The first step in our approach is to convert the Python source code into its AST representation. This is done using Tree-setter library, which parses the code into a tree structure where:

- Nodes represent syntactic elements: functions, loops, conditionals, etc.
- Edges define the relationships between these elements

For example, for the code in Listing 1, Abstract Syntax Tree is depicted in Figure 1.

```
a = 10
b = 5
if a < b:
    c = a
```

Listing 1: Python code snippet

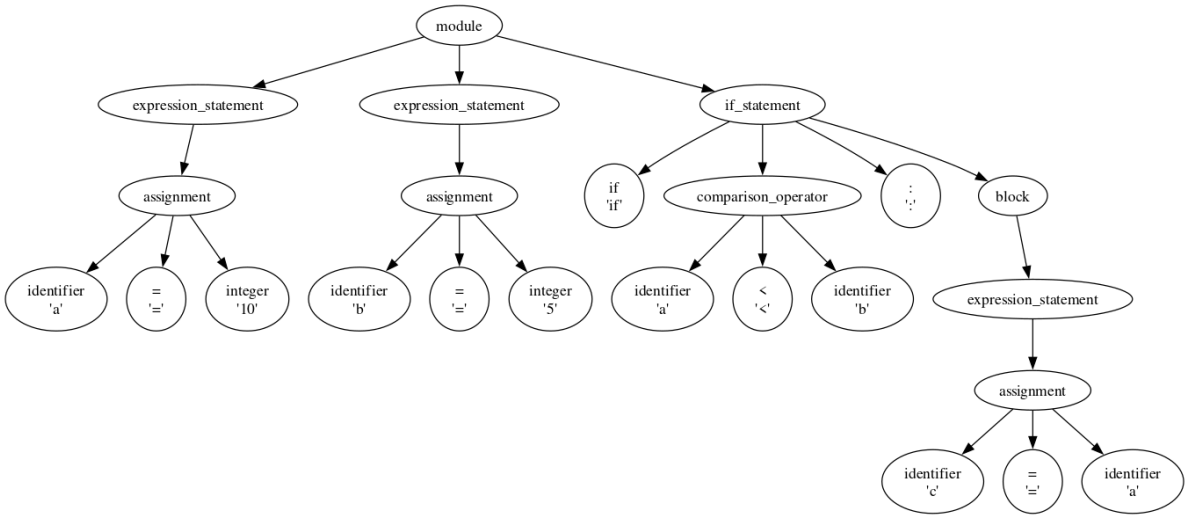

Figure 1: Example of AST

5.1 Decision Tree and Random Forest

To complement the computationally expensive LLM-based approach, we explored lightweight yet effective tree-based models — Decision Trees and Random Forest — trained on Abstract Syntax Tree (AST) features. These models provide a strong balance between performance and efficiency, making them particularly suitable for deployment in programming competitions where fast inference is critical.

The key advantage of tree-based models lies in their interpretability and speed. Unlike deep learning models, which require significant computational resources, Decision Trees and Random Forests process structured data efficiently, making them ideal for real-time detection systems. By encoding Python code into AST-based features, we enable these models to learn distinguishing patterns between human-written and AI-generated code.

However, a notable limitation of tree-based approaches is their reliance on static feature representations. As AI code generation models evolve—with new architectures, fine-tuning techniques, and prompt engineering strategies—the statistical patterns in generated code may change over time. This means that tree-based models, while highly performance in deployment, require periodic retraining on updated datasets to maintain detection accuracy. Without regular updates, their effectiveness may degrade as newer AI models produce code that diverges from previously learned patterns.

Despite this constraint, Decision Trees and Random Forests remain a practical choice for scenarios where low-latency inference is essential. Their ability to deliver high accuracy with minimal computational overhead makes them a

compelling alternative to heavy-weight LLM-based detectors, especially in resource-constrained environments such as automated classification systems or programming competitions. For long-term robustness, a hybrid approach — combining periodic retraining of tree-based models with occasional LLM-based verification — could offer the best trade-off between speed and adaptability.

5.2 Multi-Layer Perceptron

To complement tree-based methods like Decision Trees and Random Forests, we also explored a Multi-Layer Perceptron (MLP) approach for detecting AI-generated Python code. The MLP model consists of four linear layers with LeakyReLU activation functions and dropout layers for regularization. The input to the network consists of embeddings derived from the Abstract Syntax Tree (AST) representation of the code, and the output is a probability score indicating whether the code was AI-generated. This neural network approach provides a more sophisticated, non-linear alternative capable of capturing complex patterns in the data while maintaining robustness through dropout regularization. The MLP serves as an additional method in our comparative analysis of different techniques for AI-generated code detection.

6 Evaluation and comparison

6.1 Comparison of models

The results of the proposed approaches are shown on Table 1. As you can notice, the CodeBERT-base model demonstrated the best performance in terms of all metrics: F1 score, Roc/Auc, Precision, Recall and Accuracy. However, at the same time it is the second biggest and slowest model. The AST-based Random Forest is the fastest, though its metrics are still descent.

Table 1: Comparison of models

Model	F1	ROC/AUC	Precision	Recall	Accuracy	Time (s)	Memory (MB)
DeBERTa-v3-xsmall	0.899	0.890	0.870	0.930	0.891	0.07	269
DeBERTa-v3-base	0.903	0.899	0.902	0.904	0.899	0.13	701
CodeBERT-base	**0.959**	**0.959**	**0.978**	**0.941**	**0.958**	0.07	475
Random Forest AST	0.841	0.835	0.834	0.847	0.835	**0.002**	5.8
MLP AST	0.787	0.785	0.809	0.767	0.784	0.004	**0.06**

6.2 Interpretability

In order to interpret decisions of our models, we decided to add Local Interpretable Model-agnostic Explanations (LIME) [5] to our pipeline.

For example, for the code in Listing 2, the LIME explanations for AST Random Forest and CodeBERT models are shown in Figures 2 and 3 respectively.

```
n = int(input())
a = sorted(map(int, input().split()))
b = sorted(map(int, input().split()))
s = 0
for i in range(n-1, -1,-1):
    s += abs(a[i]-b[i])
print(s)
```

Listing 2: Python code snippet

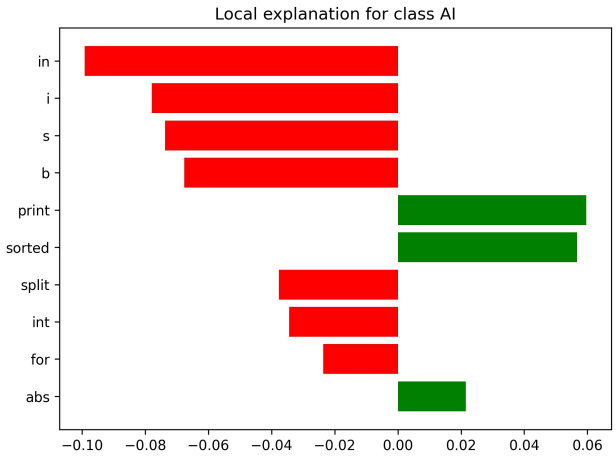

Figure 2: Lime explanation for AST RF

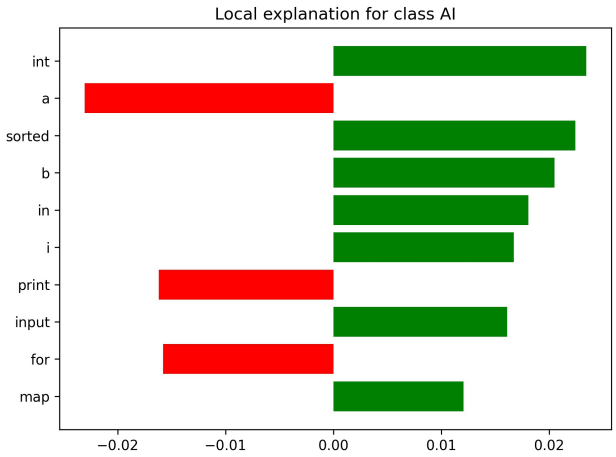

Figure 3: Lime explanation for CodeBERT

The explanation provided by LIME for both models might seem to be unsatisfactory due to applied time constraints to keep fast inference. Also, the principle of LIME work includes generating perturbations, which can result in strange and not existing code embeddings, hence the explanations can be possibly limited for interpretation.

7 Deployment

We decided to deploy our CodeBERT and AST Random Forest models via a Telegram bot with help of `aiogram` library. We choose such models, since they show best performance for each approach. You can access bot here[1].

8 GitHub

You can check GitHub of this project here.

9 Results and Discussion

We successfully implemented models for AI code detection, which might be used inside Accept [1] system. Both the AST-based and LLM-based approaches showed excellent performance. While CodeBERT wins in terms of metrics,

[1] If the bot is down, write Vsevolod Klyushev

AST-based Random Forest Classifier is much faster and less memory consuming, so it can be considered as the best solution in terms of proposed constraints: throughput and memory limitations. Also, the LIME was used to explain the decision of selected models. Finally, the Telegram bot was deployed for demonstration purposes.

References

[1] Dmitry Beresnev and Rostislav Melnikov. Accept. https://accept-school.ru/.

[2] Soohyeon Choi, Yong Kiam Tan, Mark Huasong Meng, Mohamed Ragab, Soumik Mondal, David Mohaisen, and Khin Mi Mi Aung. I can find you in seconds! leveraging large language models for code authorship attribution, 2025.

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[4] Pengcheng He, Jianfeng Gao, and Weizhu Chen. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing, 2021.

[5] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier, 2016.

[6] Shang-Yu Su, Chao-Wei Huang, and Yun-Nung Chen. Towards unsupervised language understanding and generation by joint dual learning. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, page 671–680. Association for Computational Linguistics, 2020.

[7] Hyunjae Suh, Mahan Tafreshipour, Jiawei Li, Adithya Bhattiprolu, and Iftekhar Ahmed. An empirical study on automatically detecting ai-generated source code: How far are we?, 2024.