Detecting AI-Generated Python Code via Machine Learning

This project is about defining code that is either generated using LLM or written by a human. This structure is designed for the Accept educational platform, so we focused on the speed of the model and its minimal memory consumption

Dmitry Beresnev d.beresnev@innopolis.university

Vsevolod Klyushev v.klyushev@innopolis.university

Nikita Yaneev n.yaneev@innopolis.university

GitHub: https://github.com/dsomni/ml-s25

Introduction to AI Code Detection

Challenge

The rise of AI code generation threatens fairness in programming competitions. Traditional plagiarism detection fails, LLMs produce syntactically correct but distinct solutions.

Approaches

1. LLM-based detection (high metrics, slow)
2. AST-based Tree-based models (lightweight, fast inference)
3. AST-based MLP (lightweight, fast inference)

Goal

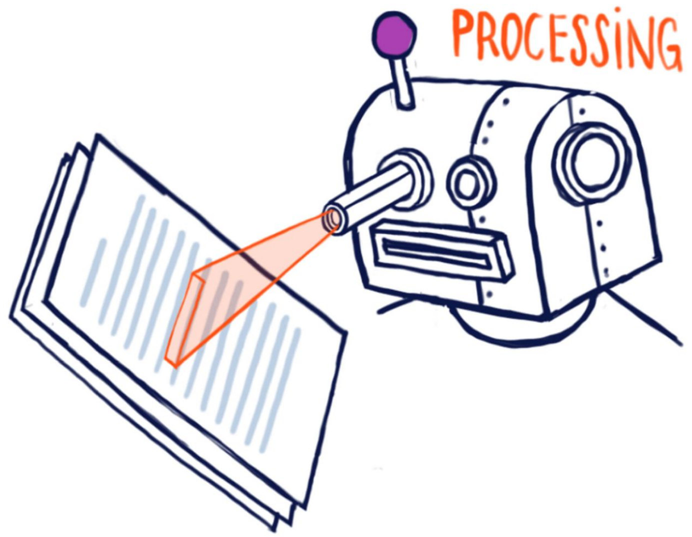
Balance detection reliability and speed for real-time competition use.

Related work

Recent studies focus on AI-generated text/code detection, but gaps remain:

• GPTSniffer (CodeBERT for Java) lacks Python generalization

• Most approaches don’t address time and memory consumption limitations



Data Generation for Detection

Human-Written Entities

As human-written code snippets, anonymized Python solutions of Accept 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 to dataset.

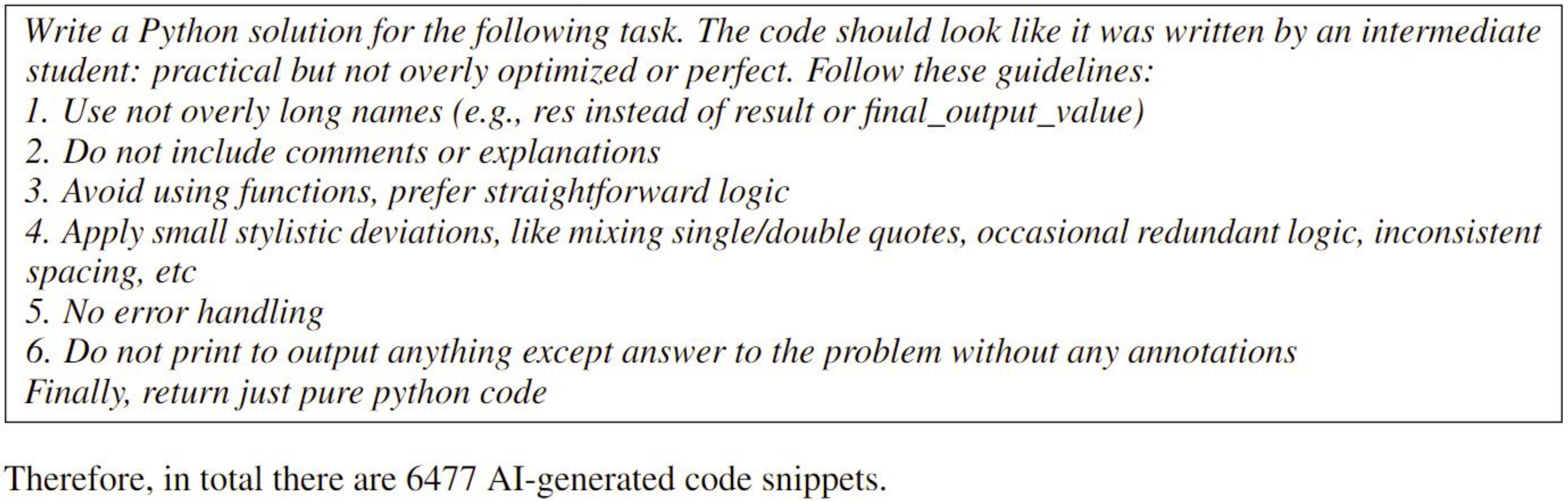
AI-Generated Entities

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

1. Evil
2. Llama-3.2-3b
3. BLACKBOX.AI
4. DeepSeek

Data Generation for Detection

To solve this, we sent the task with accept and this is the prompt:



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.

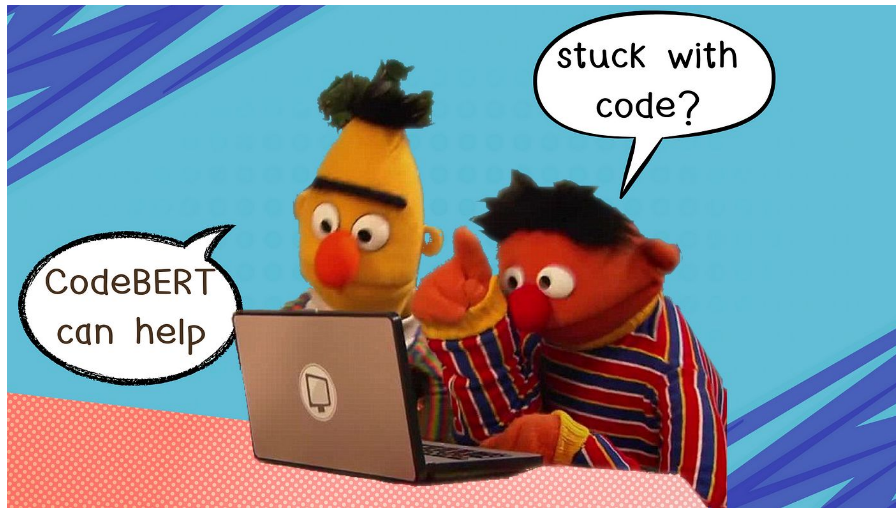
Large Language Model (LLM) Approach

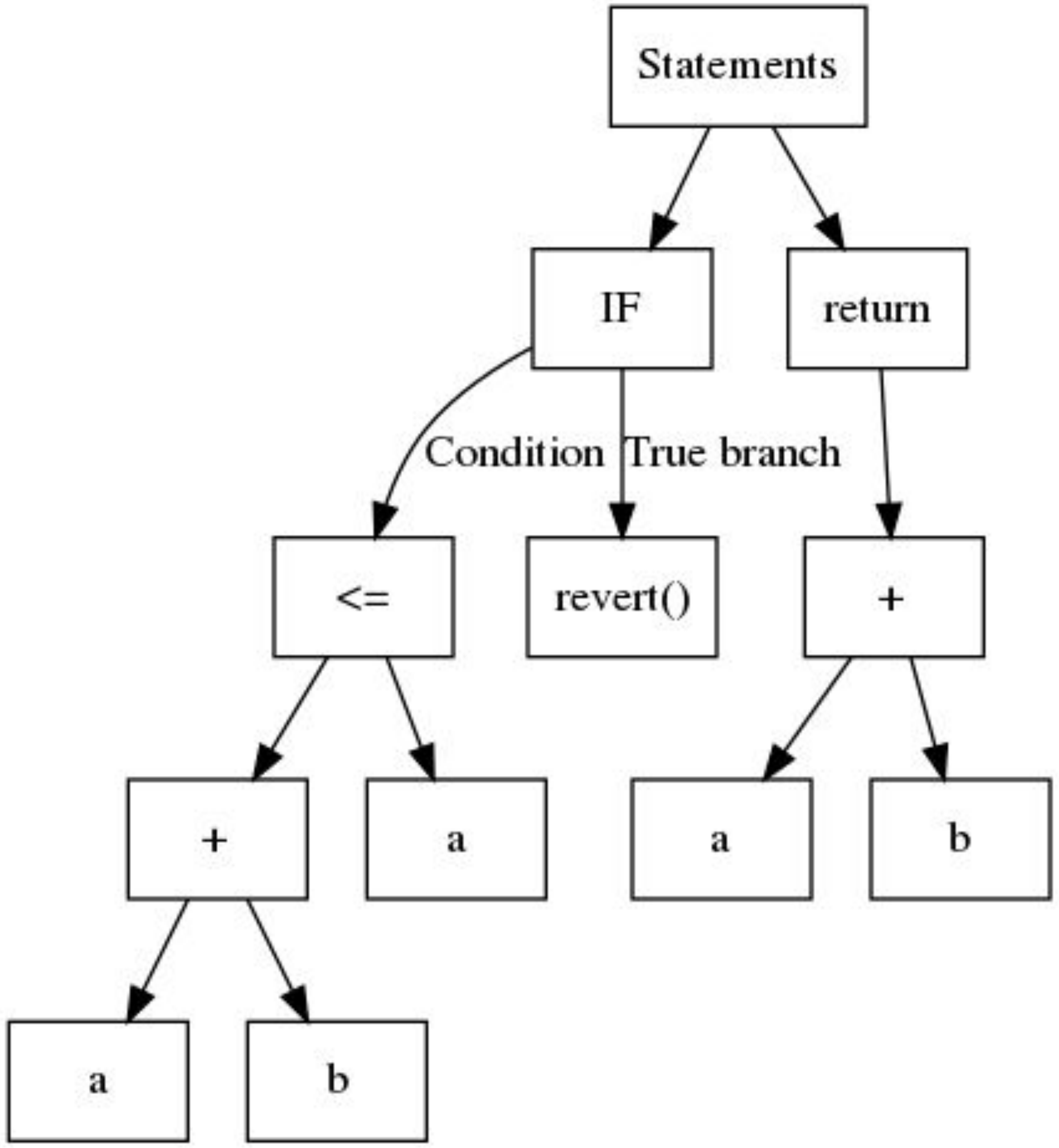
DeBERTaV3 base/small

- Improves on BERT and RoBERTa with disentangled attention and mask decoder, excelling in natural language tasks
- Trained with a linear layer and sigmoid activation to classify AI-generated code with good performance

CodeBERT

- Bimodal model trained on programming and natural language, supporting code search and documentation tasks
- Used similarly with fine-tuning to detect AI-generated Python code effectively





Abstract Syntax Tree (AST)

1

AST Representation

Converts Python code into a tree structure. This is done using Tree-sitter library

2

Tree-Based Models and MLP

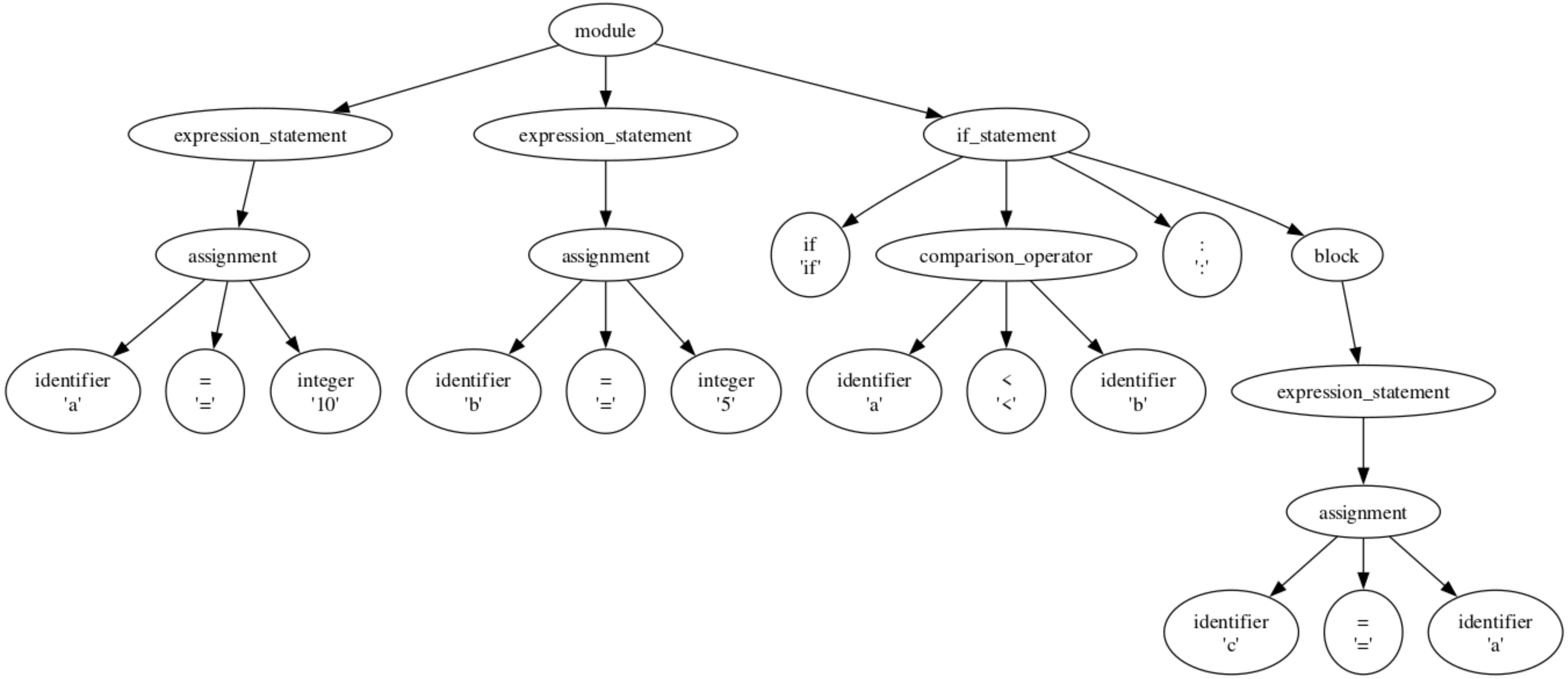
Decision Tree and Random Forest trained on AST features offer fast and interpretable detection. AST-based MLP is not such interpretable

3

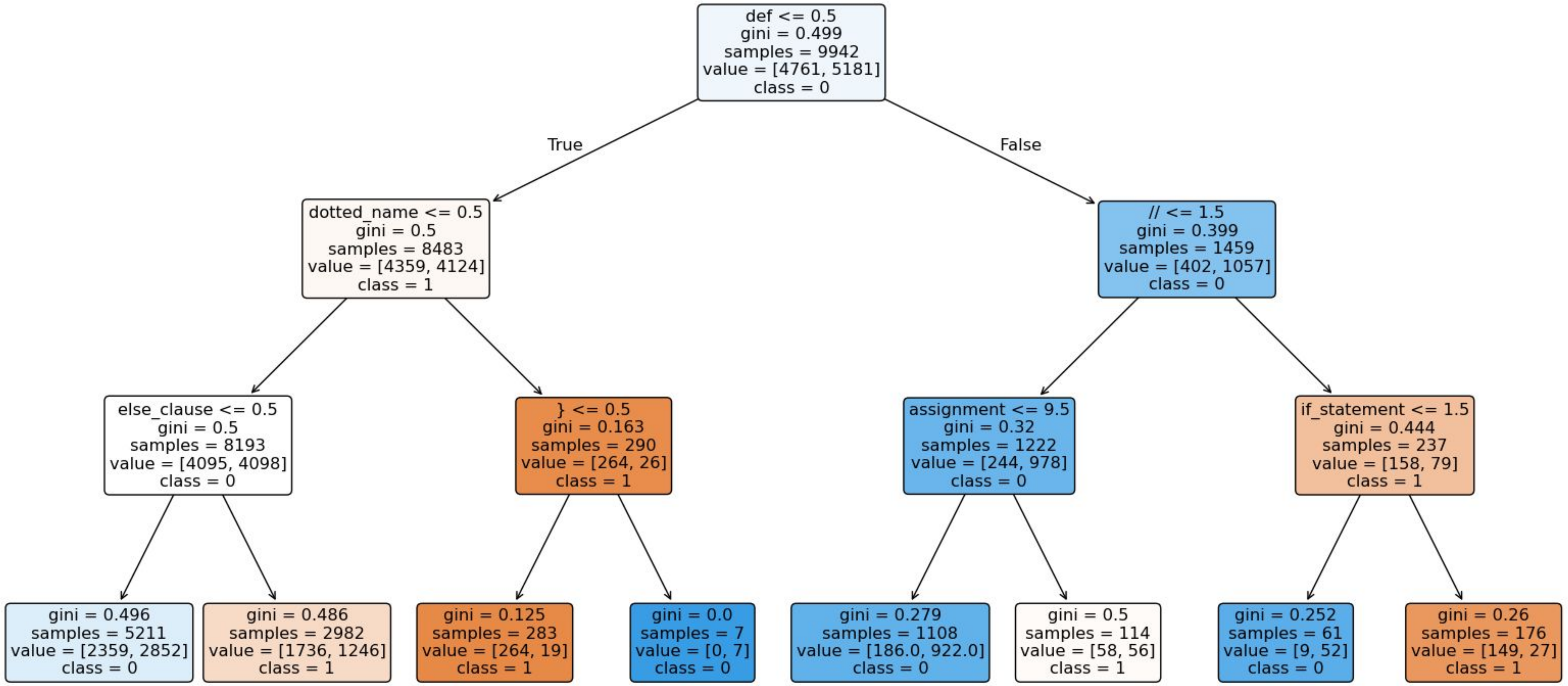
Trade-offs

Tree models are efficient but require regular retraining to adapt to evolving AI code generation, when adding new data, while MLPs can be fine-tuned.

Example of AST



Example of Decision Tree with depth 3



Evaluation and Model Comparison

Model	F1	ROC/AUC	Precision	Recall	Accuracy	Time (sec)	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
Decision tree AST	0.796	0.787	0.794	0.798	0.788	**0.001**	**0.04**
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

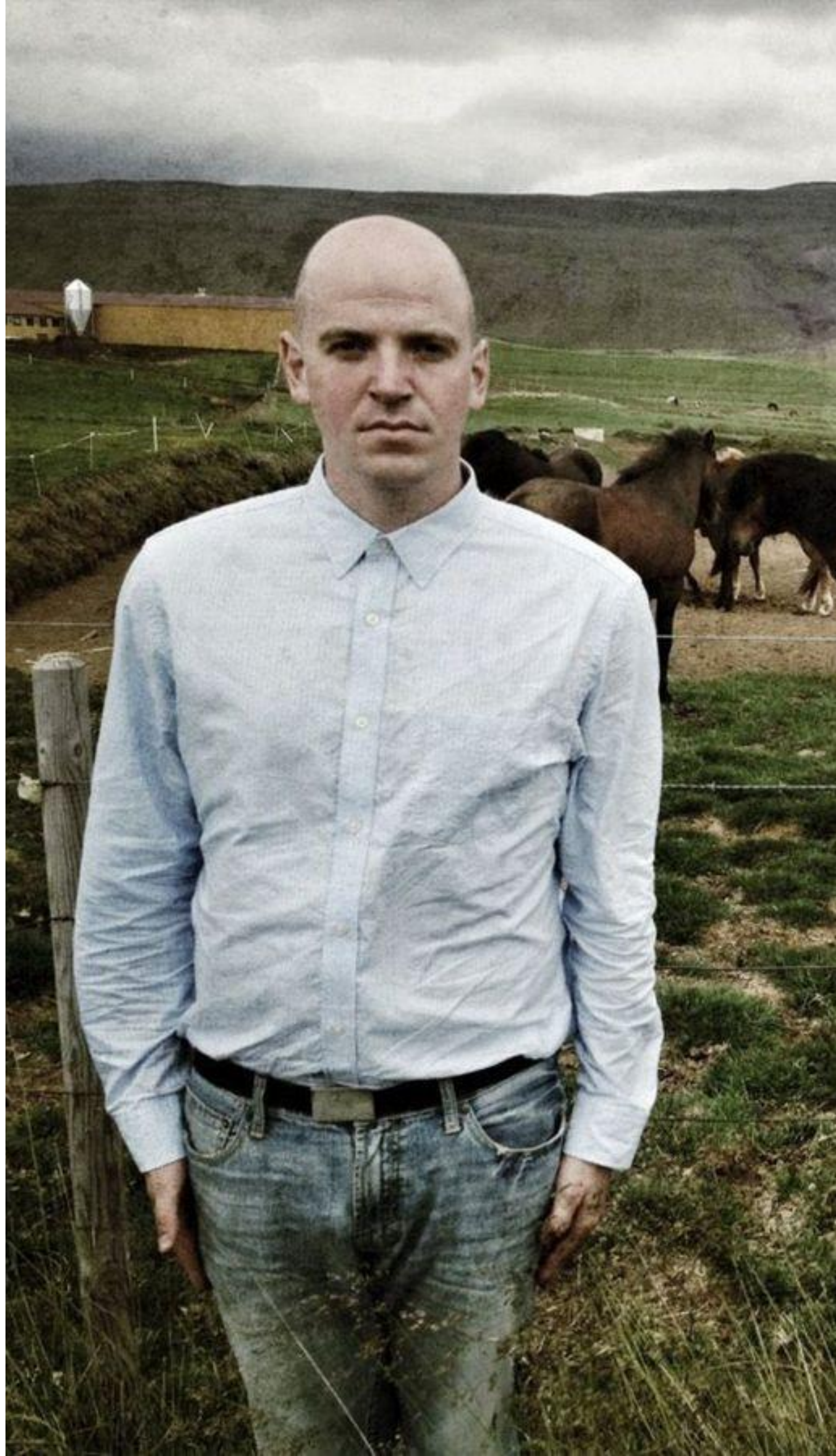
As you can notice, the CodeBERT-base model demonstrated the best performance in terms of all metrics. However, at the same time it is the second biggest and slowest model.
The AST-based Random Forest demonstrates decent performance though is quite fast and memory-efficient, though its metrics are still descent.

Results and Discussion

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



Demo

The demo showcases the detection system in action, highlighting its ability to classify Python code as human or AI-generated. Future work includes expanding datasets, refining models, and integrating the system into Accept contest platforms to ensure fairness and originality

Continued research will focus on improving model robustness against evolving AI techniques and optimizing computational efficiency for real-time use. Also, the next step is to interpolate solution on different programming languages

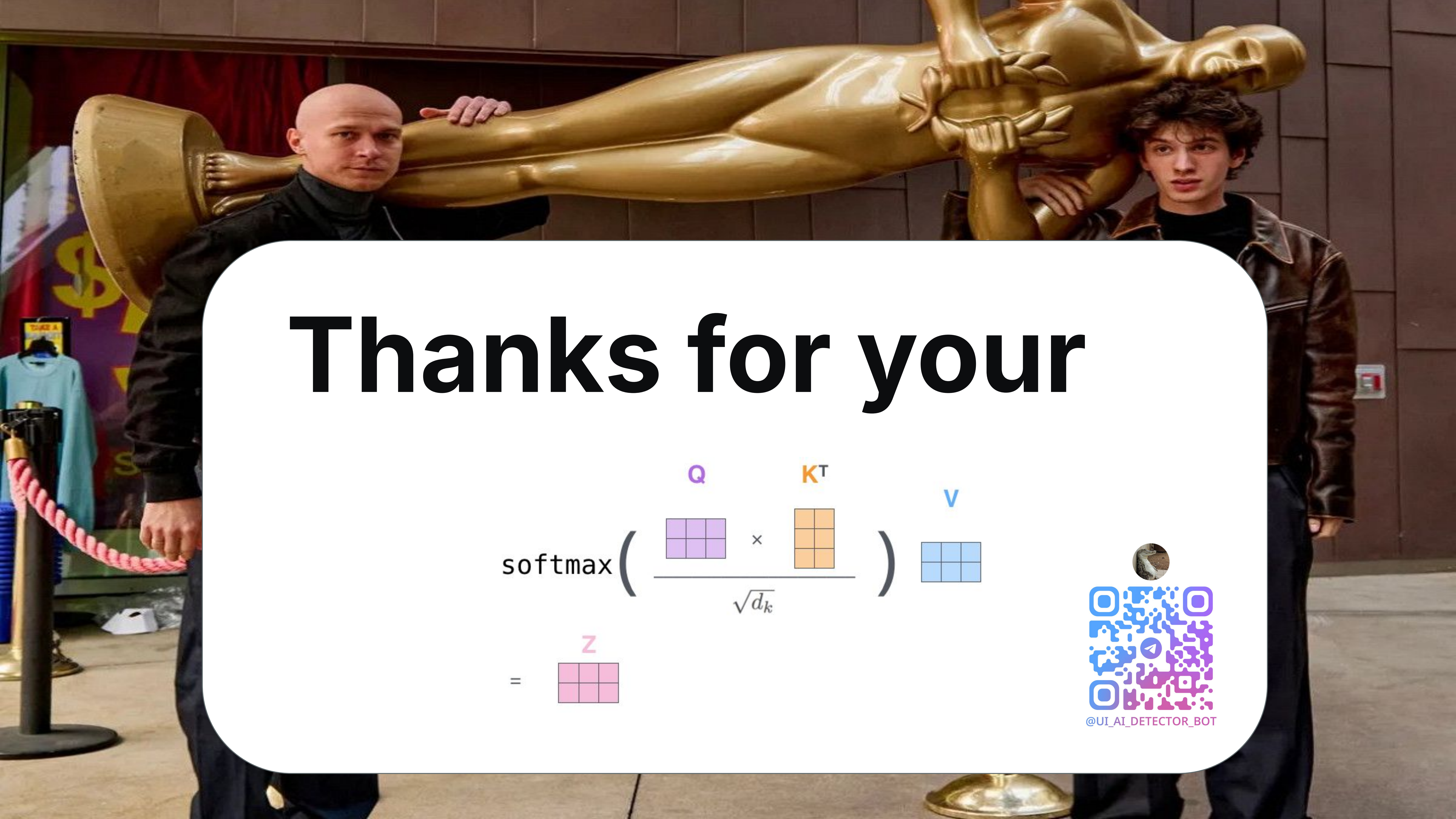
Demo

(no sorry for Rick-Roll)

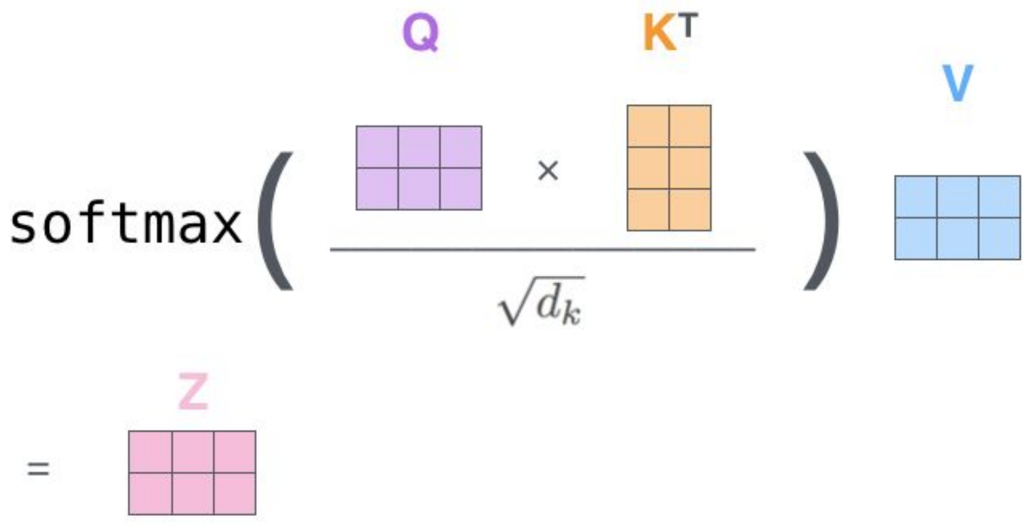
The demo showcases the detection system in action, highlighting its ability to classify Python code as human or AI-generated. Future work includes expanding datasets, refining models, and integrating the system into Accept contest platforms to ensure fairness and originality

Continued research will focus on improving model robustness against evolving AI techniques and optimizing computational efficiency for real-time use. Also, the next step is to interpolate solution on different programming languages

@UI_AI_DETECTOR_BOT



Thanks for your



@UI_AI_DETECTOR_BOT