Fakultät Angewandte Mathematik,
Physik und Allgemeinwissenschaften

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Projectreport

Detecting AI-Generated code using machine learning

Antoine Weishaupt

31.07.24

Contents

1 Detection of AI-Generated Code

In recent years, large language models (LLMs) have become increasingly popular and are now widely used for generating text and code, whether through chatbots such as Chat-GPT or coding assistants like GitHub Copilot. While text generation has already reached an impressive level, code generation is also proving to get quite effective and yielding usable results. And just like with text generation, where it has become an increasingly discussed task to detect AI-Text, the question and need to detect AI-Code will be gaining more and more traction as well.

The ability to identify AI-generated content, this includes code, text, images and videos, will become increasingly important as generative tools continue to improve, especially on the internet. Ensuring the content we consume is human-made, with a clear purpose behind it, rather than AI-generated material designed to capture attention through sheer volume, is essential for maintaining content quality and authenticity. Detecting AI-generated content will be crucial to ensure that the digital content we engage with is genuine and purposeful, rather than an attempt to maximize engagement.

Transitioning from the broader context of AI content detection on the internet, one very important use case for detecting AI-generated code is in education. Especially in Computer Science based courses AI tools like Chat-GPT provide valuable help by offering code explanation, examples, and debugging. While LLMs can be of great assistance during studying, they also pose a risk of misuse. If students use them to complete entire assignments, it harms the learning process. Ensuring that students actively engage with their coursework and develop their problem-solving skills is essential. While LLMs can enhance this learning process, it is important to have tools in place to steer away from and detect cheating. Moreover, the ability to detect AI-Generated code has significant implications in the professional world. Companies may prohibit the use of AI code due to concerns related to security and the need to verify that employees have the requisite technical skills. Detection tools help to ensure that the work produced is original and that employees demonstrate their competence without undue reliance on AI tools, as well as identifying where problematic code came from.

In summary, the interesting future of AI content detection motivated the decision to look specifically into the detection of AI-Generated code as the topic of this project. The goal is to look into techniques that have already been developed and to explore them further through own implementation.

2 Current state of research

After deciding on the topic, the first task at hand was to gain a clear overview and understanding of the current state of research. This was crucial not only to start learning about the topic and acquiring information but also for structuring the project and outlining a rough plan of future tasks. Surprisingly there was not much available material to be found. The following will summarize the findings during the first part of the project. This overview and explanation of the existing papers will also build the foundation to understand the own work later on.

2.1 Originality Score

This paper called "Calculating Originality of LLM Assisted Source Code" [1] by Balwinder Sodhi and Shipra Sharma puts forth a unique approach. The authors propose the idea of classifying code based on an originality score. This score evaluates the amount of original contribution someone has made to a piece of code D. This is done by inputting the code into a Neural Network which then generates potential prompts P that were given to an LLM to create the code. These prompts are then given to an LLM to generate new code with them. A conventional plagiarism tool is then used to compare the two sets of code. The original contribution O is determined by removing the matching parts of both codes. The ratio of O and D gives the originality score. Additionally, the total effort is defined as the sum of P and O. Unfortunately the paper only proposed this algorithm together with some early experiments proving its functionality without going into details about the implementation itself.
During the early phases of this project it was considered to explore this approach. However, without any sort of guidance on the implementation itself and unknowns involved, it was decided to focus on more documented methods and only look into it at the end which unfortunately fell short due to a lack of time. In most cases when using LLMs to aid during coding only snippets of code are generated and then integrated into existing self-written code. This makes a binary yes or no classification somewhat meaningless and which is why the idea of an originality score remains very intriguing. Especially in educational situations it will be of importance to tell how much effort a student has put into an assignment themselve and assess their performance.

2.2 Detection based on existing Text-Detectors

These two papers [5] [6] try to modify existing AI text detection tools to work with code. These approaches are highly complex, and it was quickly realised that these were too ambitious for this project. Therefore, attempting to thoroughly understanding was not continued and is the reason why they also are not explained further here. Their results do not quite match others and thus do not justify their complexity. However, it is possible that they hold more potential in the future and can be developed further.

2.3 Detection based on ML

The next paper under review, titled "Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models." [4], is closely aligned with the own work conducted in this project. It focuses on code detection in education and emphasizes its importance as well. It attempts to classify code assignments from a computer science 1 course using various machine learning models. Very interesting are the results achieved with models that utilize their own way of encapsuling the meaning of code. These are code2vec, a Abstract Syntax Tree Neural Network and Subtree based Attention Neural Network which gain some of the best results across all mentioned papers, having machine learning metrics exceeding 90%. Unfortunately they also do not give details about their implementation and the simplicity of the coding problems from the CS1 course most likely make the classification easy, hence the good results.

2.4 Papers which the project is based on

2.4.1 n-Grams

Another intriguing way to en capsule the meaning of code and subsequently use it for the classification process is used in "Distinguishing AI- and Human-Generated Code: A Case Study." [2] by Bukhari et. al.. This method involves creating n-Grams of the Abstract Syntax Tree (AST). An AST is a tree based representation of code. An n-Gram is simply a sequence of length n, which, in the context of an AST means a sequence of nodes with length n. The number of the different n-Grams in the code are counted and normalized by the total number of n-Grams. All these values are then used to train various ML models, lengths of two, three and four were used. Due to many different nodes in an AST, most of the n-Grams have a low occurrence, resulting in many values being zero or close to zero which lowers the performance of the classifiers. To mitigate this issue, categories of nodes are introduced and the n-Grams with respect to the categories are counted to obtain higher values.
As mentioned before when using LLMs to aid during coding mostly snippets are generated. With this in mind, the paper trained their models on C++ functions only instead of entire programs to align with the typical use case of LLMs and how the generated code is used.

2.4.2 Code Features

In the paper "Whodunit: Classifying Code as Human Authored or GPT-4 Generated - a Case Study on CodeChef Problems." [3] by Idialu et al. code features are used to train classifiers. A closer look at these features will be provided later on. They thoroughly created their own dataset with over 300 python coding problems from CodeChef and their corresponding human solutions. For each problem, AI solutions were created using Chat-GPT. The usefulness of features was analyzed together with different types of features. The results were then even compared to those of the n-Gram paper, with necessary modifications made to accommodate Python files.

3 Own experiments

After reading about the different approaches and getting a good understanding of the current state of code detection, some own experiments were done. This involved implementing some of the code features using Python. Then the idea emerged to try and implement both the code features and n-Grams together. The goal was to achieve improved results by capturing both the stylistic aspects of the code through the features and the structural elements through the n-Grams. From here on, whenever features are mentioned, it refers to both of these together unless specified otherwise.

3.1 Features

There are different types of features. Gameable ones that can easily be used to trick the model. These are things like comments, whitespaces and empty lines. No understanding of the code itself is needed to change them and neither will modifying them change the codes function. Since comments have nothing to do with the coding style or structure they have not been included.

Then there are features specifically based on the AST that extract values regarding it. Other features may not belong in this category yet they still use the AST in the implementation because it makes the feature extraction considerably easier. Most of the time this means walking through the AST and processing the nodes.

Usually longer pieces of code also mean more functions, keywords etc.. This needs to be normalized since it is only problem specific and not related to how AI and humans code differently. For most features this normalization is done with the length of the code.

The following lists provide an overview of all of the features carried over from the paper, as well as newly self introduced ones.

Features by Idialu et. al.	
Name	**Description (if needed)**
Avg. line length	
Avg. function length	Average amount of lines in functions
Avg. variable length	Average lenght of variable names
Avg. number of parameters	Average number of parameters of functions
Avg. branching factor	Average of how often the AST splits
Std. deviation of line length	
Std. deviation of parameters	
Cyclomatic Complexity	
Maintainability Index	
Max. decision tokens	Maximum number of decision tokens
Max. AST depth	Depth of the AST

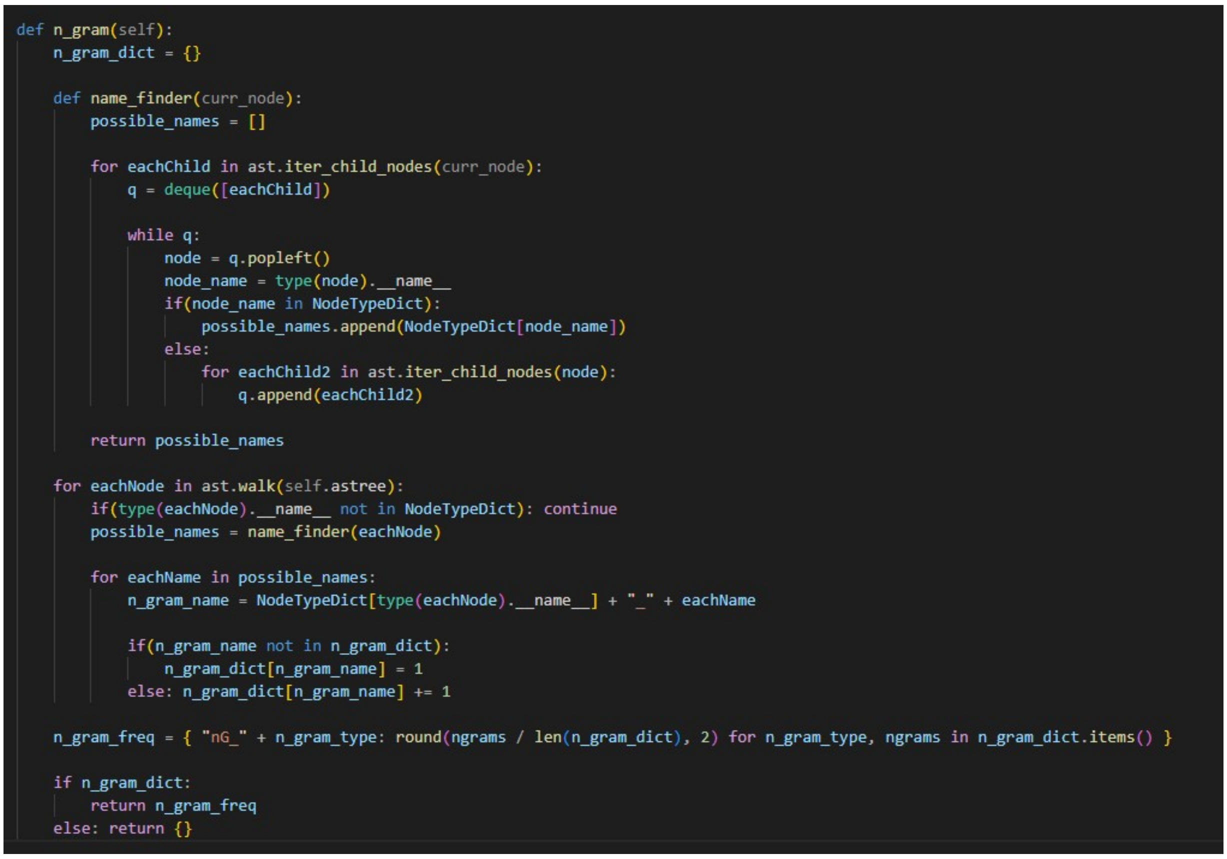
Max. nesting depth	Max nesting of functions, loops in each other
SLOC	Number of source lines of code
Empty lines	Number of empty lines
Empty lines ratio	Ratio of empty lines ot non-empty lines
Whitespace ratio	Ratio of whitespaces to non-whitespaces
Empty line density	Ratio of empty lines to SLOC
Whitespace density	Ratio of whitespaces to SLOC
Function density	Ratio of functions to SLOC
Assignment density	Ratio of assignments to SLOC
Keyword density	Ratio of keywords to SLOC
Class density	Ratio of classes to SLOC
Function call density	Ratio of function calls to SLOC
Unique keyword density	Ratio of different keywords used to SLOC
Statement density	Ratio of statements to SLOC
Assignment variable density	Ratio of variables used for assignments to SLOC
Input density	Ratio of inputs to SLOC
Literal density	Ratio of literals to SLOC

Own Features	
Name	**Description (if needed)**
LLOC/SLOC	Ratio of logical lines of code to SLOC
String quotation	Types of string quotation used (" " or ' ')
Avg. keyword frequency	Average frequency of all individual keywords
Library uses	Number of library uses
Avg. reassignments	Average reassignments of variables
Function/Function call	Ratio between functions and function calls
Assign/AugAssign	Ratio between Assignments and Aug-Assignments
Std. deviation of variables	Std. deviation of variable names
Avg. variable usage	Average number each variable is used
Keyword frequencies	Ratio of individual keywords to total keywords

All of these features are fairly straight forward and self-explanatory in terms of how they were implemented. However, three more complex features have not been named yet: Obviously the n-Grams themselves, the Node Type Average Depth and the Node Type Average Term Frequency. The implementation of these features, along with some additional information, is detailed below.

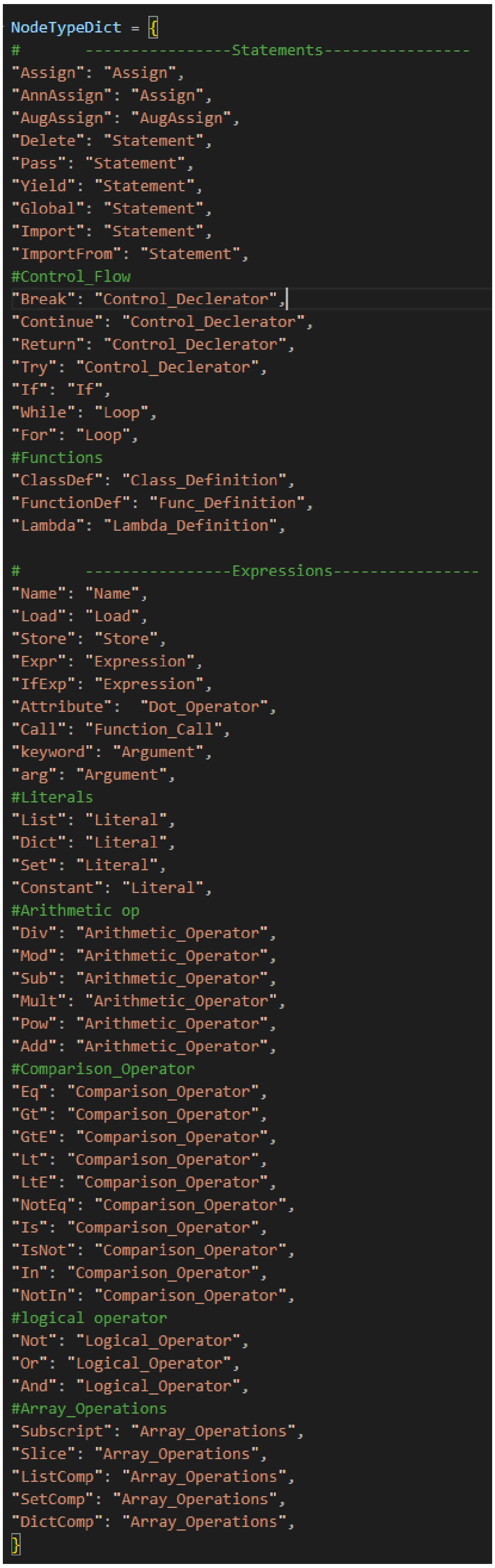
n-Gram

This implementation has a few differences from the paper and is overall simpler. Only n-Grams of two were used because the paper did not find enough of an improvement for higher values to warrant the extra effort of implementing them. Since in this case python code is being classified, a different parser, the Python 'ast' module which has different nodes, had to be used. This meant creating new categories of nodes shown in Figure 2. After shuffling things around, this appeared to be more or less the best version. It is possible that a different arrangement and different categories might yield slightly better results. Not every single node from the module was included. Some were omitted because they were essentially counted in a different way. For example, the BinOp node, which appears before any arithmetic operation, was not counted with the operator nodes themselves. Other nodes were omitted because their appearance is very rare or do not contribute anything significant. For example 'match case' nodes or 'module' nodes, which are the root node for nearly every AST, were excluded.



```python
def n_gram(self):
    n_gram_dict = {}

    def name_finder(curr_node):
        possible_names = []

        for eachChild in ast.iter_child_nodes(curr_node):
            q = deque([eachChild])

            while q:
                node = q.popleft()
                node_name = type(node).__name__
                if(node_name in NodeTypeDict):
                    possible_names.append(NodeTypeDict[node_name])
                else:
                    for eachChild2 in ast.iter_child_nodes(node):
                        q.append(eachChild2)

        return possible_names

    for eachNode in ast.walk(self.astree):
        if(type(eachNode).__name__ not in NodeTypeDict): continue
        possible_names = name_finder(eachNode)

        for eachName in possible_names:
            n_gram_name = NodeTypeDict[type(eachNode).__name__] + "_" + eachName

            if(n_gram_name not in n_gram_dict):
                n_gram_dict[n_gram_name] = 1
            else: n_gram_dict[n_gram_name] += 1

    n_gram_freq = { "nG_" + n_gram_type: round(ngrams / len(n_gram_dict), 2) for n_gram_type, ngrams in n_gram_dict.items() }

    if n_gram_dict:
        return n_gram_freq
    else: return {}
```

Figure 1: *n-Gram implementation*

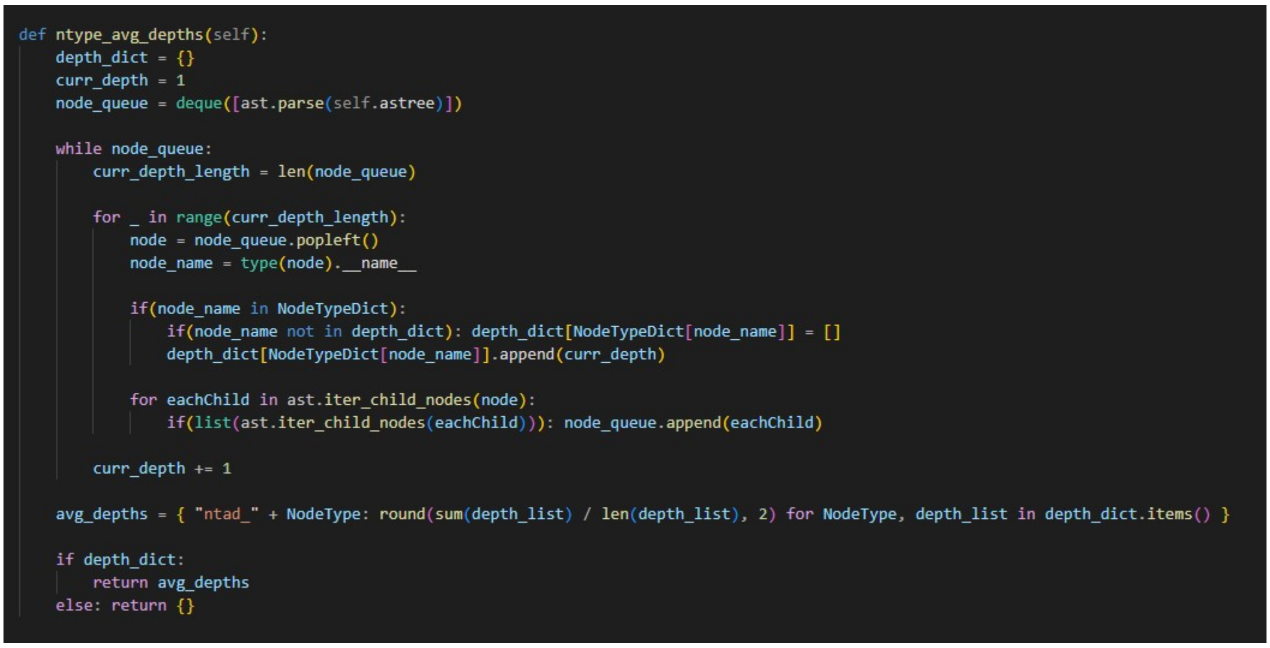


```
NodeTypeDict = {
#       ----------------Statements----------------
"Assign": "Assign",
"AnnAssign": "Assign",
"AugAssign": "AugAssign",
"Delete": "Statement",
"Pass": "Statement",
"Yield": "Statement",
"Global": "Statement",
"Import": "Statement",
"ImportFrom": "Statement",
#Control_Flow
"Break": "Control_Declerator",
"Continue": "Control_Declerator",
"Return": "Control_Declerator",
"Try": "Control_Declerator",
"If": "If",
"While": "Loop",
"For": "Loop",
#Functions
"ClassDef": "Class_Definition",
"FunctionDef": "Func_Definition",
"Lambda": "Lambda_Definition",

#       ----------------Expressions----------------
"Name": "Name",
"Load": "Load",
"Store": "Store",
"Expr": "Expression",
"IfExp": "Expression",
"Attribute":  "Dot_Operator",
"Call": "Function_Call",
"keyword": "Argument",
"arg": "Argument",
#Literals
"List": "Literal",
"Dict": "Literal",
"Set": "Literal",
"Constant": "Literal",
#Arithmetic op
"Div": "Arithmetic_Operator",
"Mod": "Arithmetic_Operator",
"Sub": "Arithmetic_Operator",
"Mult": "Arithmetic_Operator",
"Pow": "Arithmetic_Operator",
"Add": "Arithmetic_Operator",
#Comparison_Operator
"Eq": "Comparison_Operator",
"Gt": "Comparison_Operator",
"GtE": "Comparison_Operator",
"Lt": "Comparison_Operator",
"LtE": "Comparison_Operator",
"NotEq": "Comparison_Operator",
"Is": "Comparison_Operator",
"IsNot": "Comparison_Operator",
"In": "Comparison_Operator",
"NotIn": "Comparison_Operator",
#logical operator
"Not": "Logical_Operator",
"Or": "Logical_Operator",
"And": "Logical_Operator",
#Array_Operations
"Subscript": "Array_Operations",
"Slice": "Array_Operations",
"ListComp": "Array_Operations",
"SetComp": "Array_Operations",
"DictComp": "Array_Operations",
}
```

Figure 2: *Node Type Categories*

Node Type average Depths

In the referenced paper, this feature calculates the average depth of each node. Upon examining some of the extracted features vectors from the paper, it was realized that a many of them often were zero due to those particular nodes not appearing in the corresponding code file. This inspired the idea to apply the node type categories from the n-Grams here as well, reducing the amount of zeros to potentially improve the classifiers performance. Thus, the average depth of the categories was calculated instead of individual nodes.

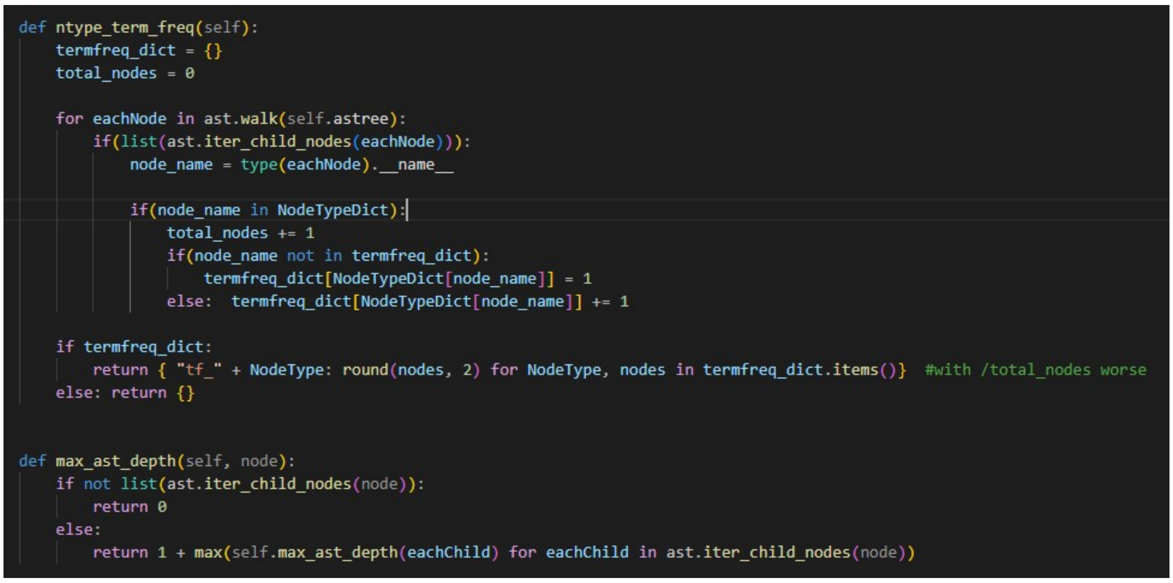


```python
def ntype_avg_depths(self):
    depth_dict = {}
    curr_depth = 1
    node_queue = deque([ast.parse(self.astree)])

    while node_queue:
        curr_depth_length = len(node_queue)

        for _ in range(curr_depth_length):
            node = node_queue.popleft()
            node_name = type(node).__name__

            if(node_name in NodeTypeDict):
                if(node_name not in depth_dict): depth_dict[NodeTypeDict[node_name]] = []
                depth_dict[NodeTypeDict[node_name]].append(curr_depth)

            for eachChild in ast.iter_child_nodes(node):
                if(list(ast.iter_child_nodes(eachChild))): node_queue.append(eachChild)

        curr_depth += 1

    avg_depths = { "ntad_" + NodeType: round(sum(depth_list) / len(depth_list), 2) for NodeType, depth_list in depth_dict.items() }

    if depth_dict:
        return avg_depths
    else: return {}
```

Figure 3: *Node Type Average Depths implementation*

Node Type term frequency

The same principal applies to this feature as well. Instead of counting the frequency of each individual node, the frequency of the categories was counted. It is worth mentioning that in this case, the results were slightly worse when the actual frequency was calculated (i.e., the individual counts divided by the total number of nodes). This is likely due to the counts of individual nodes being much smaller relative to the total number nodes, resulting in small values that are not optimal for the classifier.



```
def ntype_term_freq(self):
    termfreq_dict = {}
    total_nodes = 0

    for eachNode in ast.walk(self.astree):
        if(list(ast.iter_child_nodes(eachNode))):
            node_name = type(eachNode).__name__

            if(node_name in NodeTypeDict):
                total_nodes += 1
                if(node_name not in termfreq_dict):
                    termfreq_dict[NodeTypeDict[node_name]] = 1
                else:  termfreq_dict[NodeTypeDict[node_name]] += 1

    if termfreq_dict:
        return { "tf_" + NodeType: round(nodes, 2) for NodeType, nodes in termfreq_dict.items()}  #with /total_nodes worse
    else: return {}


def max_ast_depth(self, node):
    if not list(ast.iter_child_nodes(node)):
        return 0
    else:
        return 1 + max(self.max_ast_depth(eachChild) for eachChild in ast.iter_child_nodes(node))
```

Figure 4: *Node Type Term Frequency implementation*

The rest of the implementation was fairly straight forward. All of the features were stored in a dictionary. For each file of code a function fills up missing features with 0 to ensure that each feature appearing across any dictionary is present in every one of them. This ensures that all of them are of equal length for the classifier.

3.2 Dataset

Selecting the right dataset is crucial for the success of any machine learning project. Creating and refining a new dataset would have been highly time consuming and might not even have yielded optimal results. Fortunately, Idialu et. al. [3] made their dataset publicly available. The dataset they used also included AI-Generated code that did not correctly solve the problems. Consequently, the model likely trained on these errors. If wrong files should be taken into account depends on the situation the model is used for. For example, in a CS1 course, it would probably make sense because there are likely errors made by the students. Contrarily, in a workplace environment code is expected to be correct and including them might be less relevant. For this project, the incorrect files were excluded to observe the effects. As anticipated, the exclusion of incorrect files resulted in slightly lower scores as shown later, likely due to the model not training on the variety of errors.

3.3 Classifier

XGBoost, a decision tree classifier, was selected as the main classifier. The decision to utilize it was based on two considerations. Firstly, all three papers [4] [3] [2] where it was used reported some of the best results with XGBoost. Also it made it possible to directly and accurately compare the effect of the changes to the paper by Idialu et. al. due to using the same classifier. It was trained an evaluated in the same way too because of this reason. It involved 10-fold-cross-validation, where all files of a given problem were only used once in the validation set to avoid both training and validating on a problem during the same fold.

4 Results

The overall best results achieved in the commonly used machine learning metrics are below.

	Accuracy	**Precision**	**Recall**	**F1-Score**	**Auc-Roc**
This project	0.91	0.90	0.92	0.91	0.91
Paper	0.87	0.89	0.86	0.87	0.87

For an accurate comparison, the model from Idialu et. al. was downloaded and trained on the dataset described in 3.2. An increase of 2-6% can be seen across all metrics. While these gains seem promising, they do not fully capture the details of the findings, and some important factors need to be addressed. During the implementation process, it was observed that adding certain features to the model did not always lead to improved performance; in some cases, results even worsened by approximately 0.2-1%. The following list shows which of the features exhibited this behavior.

- No Change:
  Std. deviation of parameters, Class density, Function call density, Unique keyword density, Statement density
  (Own:) Function/Function call, Assign/AugAssign
- Worse:
  Avg. branching factor, Assignment variable density, Input density, Literal density
  (Own:) Avg. keyword frequency, Library uses, Avg. reassignments, Std. deviation of variables, Avg. variable usage, Keyword frequencies

The removal of the negative features led to improved results. Unlike initial expectations, incorporating the n-Grams into the model did not enhance its performance. The same goes for the node categories introduced in the functions for Node Type average Depths and Node Type term frequency. The latter is understandable as the amount of n-Grams scale exponentially without the categories which is why they are very much needed in this case. For the amount of nodes themselves, the categories only reduce them by a smaller, fixed amount of around 100. This relative difference is not significant enough to impact the model.
To assess the effectiveness of the features independent of XGBoost, three additional models were trained. This included a Feed Forward Neural Network, a Recurrent Neural Network, and Logistic Regression. FFNN well suited for modeling relationships between features, RNN effective for capturing sequential data like n-Grams, and Log-Reg since it is a strong binary classifier.

	Accuracy	Precision	Recall	F1-Score	Auc-Roc
FFNN	0.85	0.83	0.87	0.85	0.85
RNN	0.84	0.84	0.83	0.84	0.84
Log-Reg	0.86	0.85	0.86	0.86	0.86

They prove to be less effective than XGBoost. Nevertheless, they come very close to the performance of XGBoost in [3], where it also had an 'easier' dataset, despite XGBoost being the best one across all papers. This further supports the following assumption:
It appears that a performance ceiling may have been reached with the current set of 'simple' features. More sophisticated models, potentially incorporating advanced techniques (using attention etc.), might be necessary for further improvements. There are multiple reasons why this is believed to be the case. As already mentioned, XGBoost was ahead of other classifiers but with the combined approached shown here FFNN, Log-Reg and RNN managed to get very close to the papers implementation where code features only were used. This possibly shows that n-Grams in combination with the other features, do work better but XGBoost is already performing at the limit that it does not improve it any further.
This is also supported by how very little the performance changed after getting close to the limit of above 90%. Improvements were only small, at around 0.2-0.5%. Performance drops by features listed in (4) were in the same range. Some even managed to yield improvements when below 90%, again suggesting that a threshold has been reached.

5 Possible changes and conclusion

Several potential improvements could still enhance this work. The current n-Gram implementation was relatively simple compared to what Bukhari et. al. [2] did and can be extended upon. n-Grams of greater length could be used. Additionally, they could be created using a different parser, creating a different AST. However, using the n-Grams without the rest of the features it was managed to achieve results of around 80%. This is fairly close to what Idialu et. al. achieved with their exact modification of Bukhari et. al.'s version to work with Python code in their dataset. Considering that the dataset used here is slightly more difficult, this simple n-Gram version already performed very well for how simple it is. One type of features that could bring better performance are values in respect to total occurrences across all files. For example, the deviation of the average frequency of keywords across all AI and human files.
A couple of strange things were observed during the project and I want to mention. Notably, XGBoost exhibited varying results based on the order in which the features were arranged in the feature vector's dictionary. This is unexpected but could be explained by how it is calculated which feature is picked as the next decision in the decision tree. However, another thing is entirely unexplainable. Results fluctuated when minor changes were made at certain points, such as adding a comment in the code or simple a empty line. Although these variations were minimal (less than 1%), after some time gains and losses were in the same region. This

complicated the task of accurately assessing a feature's impact and introduced a degree of frustration.

Despite these challenges and the less significant improvements than initially hoped, achieving high classification accuracy (around 90%) is still impressive. The work on the project was enjoyable, apart from the just mentioned frustration, and it will be interesting to see how code detection evolves further in the future. While 90% accuracy are not enough to to make definitive statements in something like a CS1 course, it could already be used as a guide for human decision making.

References

[1] Sharma, Smriti and Balwinder Sodhi. "Calculating Originality of LLM Assisted Source Code" arXiv (Indian Institute of Technology), Jan. 2023, https://arxiv.org/pdf/2307.04492.pdf

[2] Bukhari, Syed Faisal Ahmed, et al. "Distinguishing AI- and Human-Generated Code: A Case Study." arXiv (University of Calgary), Nov. 2023, https://ldklab.github.io/assets/papers/scored23-aicode.pdf

[3] Idialu, Oseremen Joy, et al. "Whodunit: Classifying Code as Human Authored or GPT-4 Generated - a Case Study on CodeChef Problems." arXiv (Unifersity of Waterloo), Mar. 2024, https://arxiv.org/pdf/2403.04013.pdf

[4] Hoq, Muntasir, et al. "Detecting ChatGPT-Generated Code Submissions in a CS1 Course Using Machine Learning Models." Proceedings of the 55th ACM Technical Symposium on Computer Science Education, Mar. 2024, pp. 526-532, https://doi.org/10.1145/3626252.3630826

[5] Xu, Zhenyu, and Victor S. Sheng. "Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models." Proceedings of the 24th AAAI Conference on Artificial Intelligence vol. 38 no. 21, Mar. 2024, https://doi.org/10.1609/aaai.v38i21.30361

[6] Yang, Xiaobo, et al. "Zero-Shot Detection of Machine-Generated Codes." arXiv (University of California), Jan. 2023, https://arxiv.org/pdf/2310.05103.pdf