Title: Exploring the Efficacy of Comparing Python Code Using Abstract Syntax Trees to Detect AI Code Submissions

Author: Junseok Hur

Advisor: Dr. John Santore

Department of Computer Science

Bridgewater State University

April 14, 2024

Table of Contents

INTRODUCTION:

In today's digital learning landscape, the omnipresence of readily available information and the sophistication of AI-driven tools have compounded the challenges of maintaining academic integrity. Cheating and plagiarism have become increasingly prevalent issues among students, primarily driven by the ease with which GPT-like language models can generate academic content. The allure of effortlessly producing assignments or code has led to a surge in academic dishonesty cases. Educators and institutions now face the critical task of countering this trend with innovative solutions that effectively detect and deter such deceptive practices.

This study explores the potential of leveraging Python code analysis and Abstract Syntax Trees (AST) to detect cheating and plagiarism in programming assignments. Examining the structural elements of code through AST aims to develop a comprehensive understanding of the unique patterns and characteristics associated with code submitted by different students. This enables us to distinguish between individual student submissions and identify similarities that may indicate potential plagiarism or unauthorized collaboration.

GOAL:

This project explores the potential of Python AST and advanced clustering algorithms for addressing academic dishonesty in programming assignments. By comparing code submitted by different students, including authentic submissions and those generated by AI tools, the goal is to provide educators with a more comprehensive toolset for safeguarding academic integrity and promoting a fair and equitable learning environment.

To achieve this goal, advanced clustering techniques, notably the k-means clustering algorithm, are used to enable educators to analyze a large volume of code submissions visually and accurately.

RELATED WORK:

Automated similarity checks of two code files using Abstract Syntax Trees (AST) are well-documented in existing literature (Paredes, 2020). However, in my case, the objective is to automatically identify files in a selected directory based on their file extensions and extract their contents for subsequent comparison. Once the files have been gathered, the comparison process begins. The aim is to compare all files at once. This integrated approach allows for a comprehensive assessment of the codebase's similarities.

The files are processed through an AST-based similarity check algorithm to achieve this. Each file's AST representation is computed, enabling a structured comparison across the entire dataset. A matrix of pairwise similarity scores is generated by comparing the AST of all files in the directory.

With the similarity scores produced, the next step is to employ K-means clustering. This algorithm partitions n observations into k clusters, each belonging to the cluster with the nearest mean. Once the clustering is complete, the shape of the clusters is visualized on a graph. Each cluster represents a group of files that share similar code structures. By examining these clusters' distribution and characteristics, I found it impossible to identify anomalous patterns indicative of academic dishonesty. However, the technique can specify the suspicious data sets with a clustering graph.

In summary, this approach leverages automated file retrieval, AST-based code comparison, and K-means clustering to detect patterns of similarity among code files within a directory. Visualizing cluster shapes aids in identifying clusters associated with potential dishonesty, thereby facilitating the identification of students engaging in academic misconduct.

TOOLS:

I utilized various prebuilt tools in this project to facilitate automated code similarity analysis. Firstly, I employed a Python library for Edit Distance computation, streamlining the calculation of Levenshtein's Edit Distance (Sidorov et al., 2015) between code snippets. This library enabled efficient quantification of the dissimilarity between pieces of code and directly provided similarity in percent format. Additionally, I altered two distinct methods for comparing Abstract Syntax Trees (AST), providing comprehensive coverage for assessing structural similarities across code files. These AST-based techniques offered sophisticated means of detecting resemblances in code structures. Finally, I utilized Python's matplotlib library to visualize K-means clustering results. This graphical representation facilitated the identification of distinct clusters within the dataset, aiding in detecting anomalous patterns suggestive of academic dishonesty. Through the seamless integration of these prebuilt tools, the project streamlined the process of code similarity analysis, enhancing the effectiveness of identifying potential instances of misconduct.

METHODOLOGY:

This section describes the methodology for investigating the potential of leveraging Python code analysis and Abstract Syntax Tree (AST) to detect cheating and plagiarism in programming assignments. It also discusses the utilization of advanced clustering techniques with the k-means algorithm.

1. Abstract Syntax Tree (AST) Analysis
    a. AST Generation: I will explain the process of generating Abstract Syntax Trees from code submissions and outline the tools or libraries used.
    b. Feature Extraction: I will describe extracting relevant features from the AST, such as node types, code structures, and other essential attributes for comparison.
2. Code Comparison and Clustering
    a. Edit Distance Algorithm: I will describe the utilization of the edit distance algorithm as an additional method for comparing code submissions. Explain how this algorithm measures the similarity or dissimilarity between code fragments by calculating the minimum number of edit operations (insertions, deletions, substitutions) required to transform one code snippet into another. Discuss its role in enhancing the accuracy of code comparison.

b. K-means Clustering: I will provide a step-by-step explanation of how the k-means clustering algorithm was applied to group similar code submissions, clarifying the choice of k (number of clusters) and how it was determined.
c. Batch Code Comparison: I will allow users to compare multiple code submissions simultaneously. This enhancement will expedite the detection of potential academic dishonesty.

ABSTRACT SYNTAX TREE (AST):

The Abstract Syntax Tree (AST) is the module that helps Python applications process trees of the Python abstract syntax grammar. The abstract syntax might change with each Python release; this module helps to determine what the current grammar looks like programmatically.

The AST in Python is a hierarchical representation of the structure of Python code. When you write Python code, it goes through a series of steps before it is executed. One of these steps is parsing, where the code is converted from text into a structured format that the Python interpreter can work with. This structured format is AST.

Here is how it works:

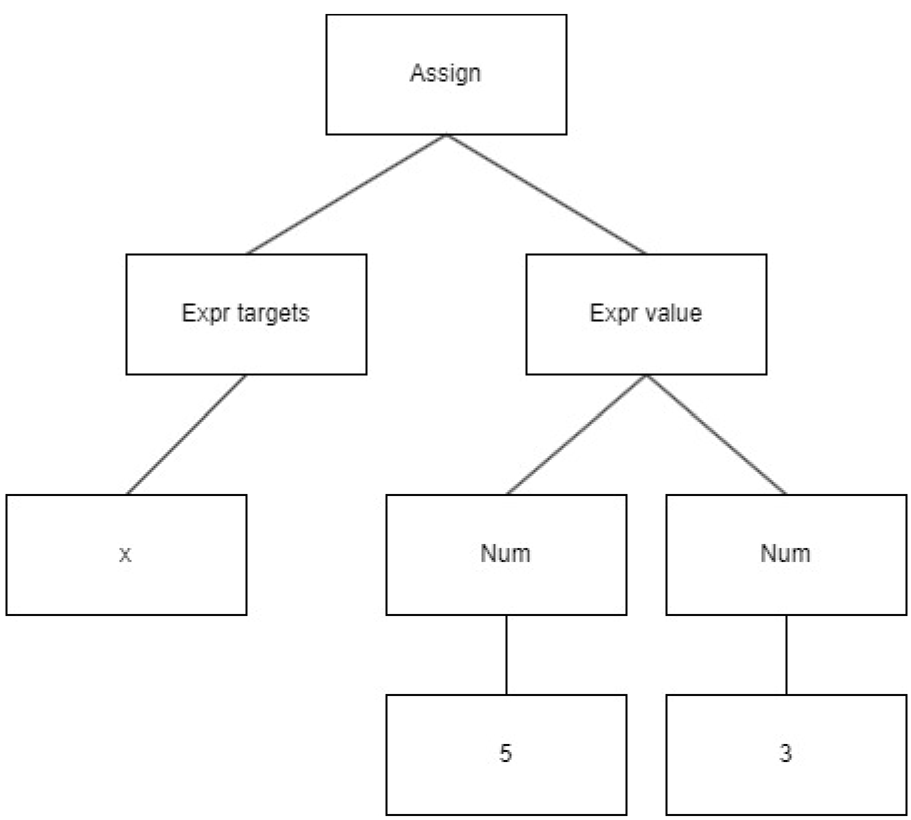
1. Parsing: The Python interpreter parses the code using the Python parser. During parsing, the code is broken down into tokens and organized into a tree-like structure according to the syntax rules of Python.

2. AST Generation: Once the code is parsed, the Python interpreter generates the AST. The AST represents the code's structure in a way that is easier for the interpreter to analyze and execute.
3. Traversal and Execution: The interpreter then traverses the AST to execute the code. During traversal, it interprets each node in the tree and performs the corresponding actions.

For example, if you have a simple Python statement like:

x = 5 + 3

The AST for this code might look something like this:



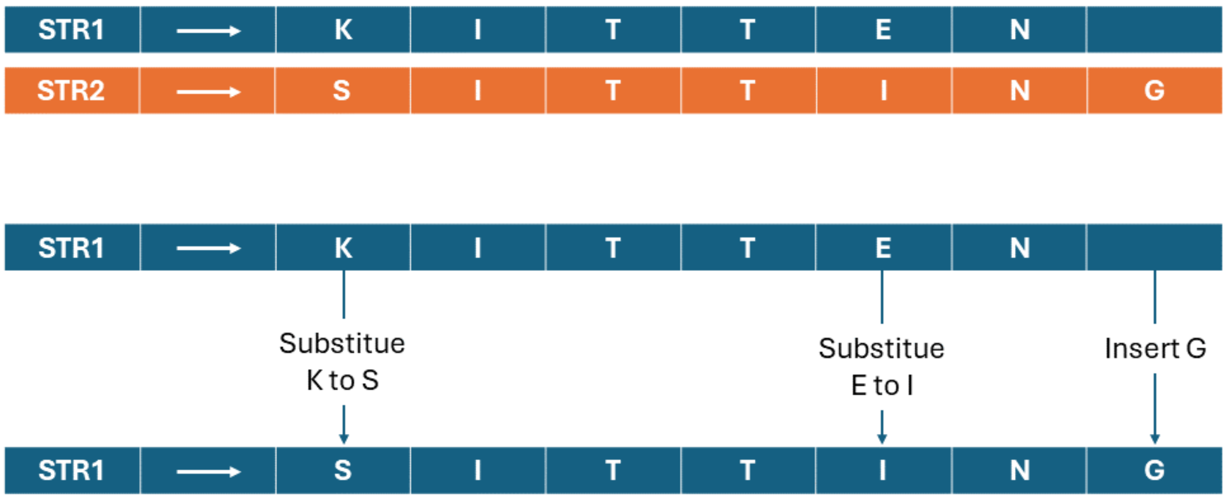
The tree structure is the same even if variables and log operands change. Here, the AST represents an assignment operation (Assign) where the variable x is assigned the result of an addition (Add) operation between two numbers (Num). The target stores the variable's name, and the value stores the value that is assigned. Using this tree, the similarity of the code is tested by excluding variable values from the method and comparing only values. That is, ignore the name of the target on the object, get the value and compare it.

The AST in Python provides a structured representation of Python code, facilitating various forms of code analysis and manipulation. It is generated during the parsing phase of code execution and is used by the interpreter to execute the code efficiently. Understanding the AST is useful for various purposes such as code analysis, transformation, optimization, and even for creating tools like linters or static analyzers.

EDIT DISTANCE ALGORITHM:

Edit Distance, also called Levenshtein Distance, measures the similarity between two strings. It quantifies the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into the other.

For example, the Edit distance between "kitten" and "sitting" is 3 because three edits are needed to change "kitten" to "sitting" (substitute 's' for 'k'; substitute 'i' for 'e'; and insert 'g' at the end).



Minimum number of edits to convert Str1 to Str2 = 3

This metric has various applications, including spell checking, DNA sequencing, and natural language processing tasks like machine translation and text summarization. It is particularly useful when understanding the similarity or difference between two sequences of characters is essential.

EXPERIMENTAL WORK:

In this experiment, I analyze AI code generation tools and the codes provided by the students using AST. Structured Python codes have objects that contain contrasting functions required for the coding process, and only the necessary parts of those objects are extracted and compared. When parsing the Python code and converting it into structured trees that the Python interpreter can work on, the desired specific value is extracted and compared with the Edit Distance algorithm to calculate the number of characters difference in the string and convert them into percentages to show similarity.

The code for comparison is created by posing identical questions to various code generation AIs (Artificial Intelligence) available for free distribution, akin to students engaging in academic dishonesty. Additionally, the code of 100-level students, supplied by the advisor, is included in the comparison group. The questions posted to Artificial Intelligence code generators are unified as below:

> "The task is to write a program in Python that prints numbers from 1 to a given number 'n'. However, for multiples of 3, the program should print 'Fizz', for multiples of 5, it should print 'Buzz', and for numbers that are multiples of both 3 and 5, it should print 'FizzBuzz'."

and all AIs initially presented code employing for and if-else statements. Consequently, requests were made to some AIs for code devoid of such specific statements, thereby augmenting the dataset.

SAMPLE OUTPUT WITH AI CODE GENERATOR:

1. ZZZ Code AI

Python Program to Print FizzBuzz Numbers

language-python Copy code

```python
def fizzbuzz(n):
    for i in range(1, n+1):
        if i % 3 == 0 and i % 5 == 0:
            print("FizzBuzz")
        elif i % 3 == 0:
            print("Fizz")
        elif i % 5 == 0:
            print("Buzz")
        else:
            print(i)

# Test the function with n = 15
fizzbuzz(15)
```

2. Replit

GJ gjwnstjr100 just now

Write a program that prints the numbers from 1 to n. But for multiples of 3, print 'Fizz' instead of the number, and for multiples of 5, print 'Buzz'. For numbers that are multiples of both 3 and 5, print 'FizzBuzz'.

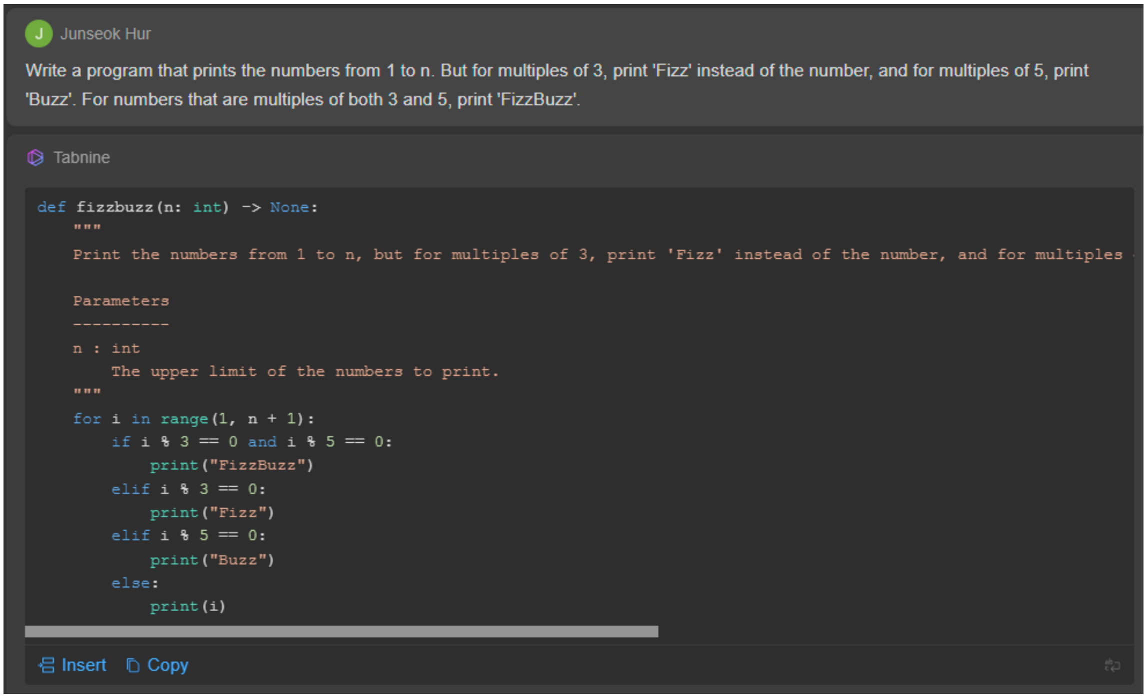
AI just now Generate Basic

Here is a program that prints the numbers from 1 to n. But for multiples of 3, prints 'Fizz' instead of the number, and for multiples of 5, prints 'Buzz'. For numbers that are multiples of both 3 and 5, prints 'FizzBuzz':

Python Copy Revert

```python
n = int(input("Enter a number: "))
for i in range(1, n + 1):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

3. Tabnine



J Junseok Hur

Write a program that prints the numbers from 1 to n. But for multiples of 3, print 'Fizz' instead of the number, and for multiples of 5, print 'Buzz'. For numbers that are multiples of both 3 and 5, print 'FizzBuzz'.

Tabnine

```
def fizzbuzz(n: int) -> None:
    """
    Print the numbers from 1 to n, but for multiples of 3, print 'Fizz' instead of the number, and for multiples

    Parameters
    ----------
    n : int
        The upper limit of the numbers to print.
    """
    for i in range(1, n + 1):
        if i % 3 == 0 and i % 5 == 0:
            print("FizzBuzz")
        elif i % 3 == 0:
            print("Fizz")
        elif i % 5 == 0:
            print("Buzz")
        else:
            print(i)
```

Insert Copy

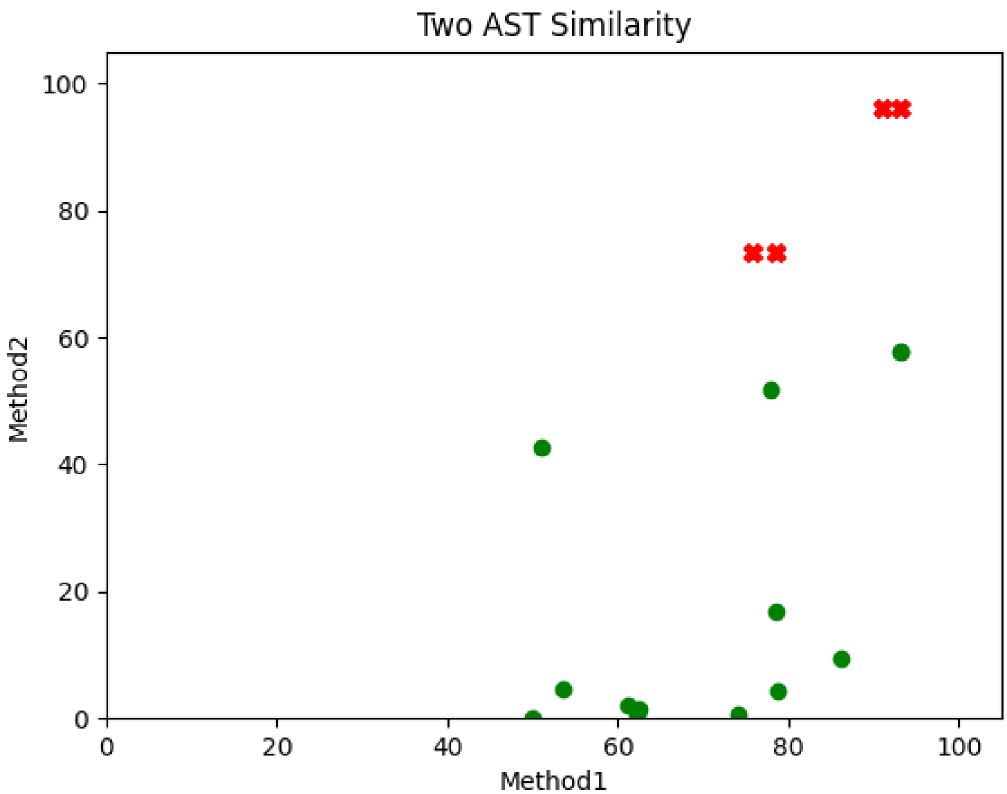
Comparison methods:

Both methods start by examining the sub-trees of the AST of the code in each file to determine if there are any matching sub-trees. The first method, comparing code files, involves examining the child trees of each file to determine if matching trees exist. This process includes defining the weight of trees based on the number of nodes they contain. Following this, the task involves defining the weight of all sub-trees and subsequently dividing the tree weight by the maximum depth reached to calculate a similarity index. A linear programming algorithm expedites the comparison process, significantly reducing the time complexity from O(n!) to O(n^3). The Munkres algorithm (Paredes, 2020, p. 9) is utilized for efficient comparison.

In the second method, duplicate values within the child trees of each code snippet are transformed into specified representations, such as function and variable names. This transformation enables selective comparison of essential syntax elements only. Subsequently, the substituted syntax undergoes comparison using the edit distance algorithm to compute similarity ratios. These ratios provide insights into the degree of similarity between code snippets, facilitating the identification of patterns indicative of code reuse or plagiarism.

Two comparison methods had to be performed for the k-mean cluster. The Munkres algorithm and the Edit Distance algorithm showed 100% similarity when comparing precisely the same codes. However, the Munkres algorithm had a significantly lower similarity percentage than the Edit Distance algorithm. This is because of the difference between the algorithms of the two methods.

The Edit Distance algorithm primarily measures the similarity between two strings. It calculates the minimum number of operations required to transform one string into another. The Munkres algorithm is used to solve the assignment problem in combinatorial optimization. It efficiently finds the optimal computing assignment for tasks based on the cost of completing each task by each computing for matching problems. In summary, while the Edit Distance algorithm measures the similarity between sequences of elements, the Munkres algorithm solves optimization problems related to assignment and matching. The result below figure:



If the code is more than 70% like both methods, it is marked with a red X mark. The X mark indicates that academic dishonesty is suspected because the code is highly similar. As in the figure above, when many data are compared, a cluster is formed and appears in the graph. The suspicious data is also distinguished by an X mark.

RESULTS:

The results of running a program with a small number of similar codes and the comparison codes are listed below:

- Student code 1

```python
for i in range(1,101):
    if i % 5 == 0 and i % 3 == 0:
        print("FizzBuzz")
    elif i % 5 == 0:
        print("Buzz")
    elif i % 3 == 0:
        print("Fizz")
    else:
        print(i)

```

- Student code 2

```python
number = 1

while number <= 100:
    number = number+1
    print(number)
    if number % 3 ==0  and number % 5 == 0:
        print("FizzBuzz")

    elif number % 3 == 0:
        print("Fizz")

    elif number % 5 == 0:
        print("Buzz")

```

- Student code 3

```python
list = [1, 2, "fizz", 4, "buzz", "fizz", 7, 8, "fizz", "buzz", 11, "fizz", 13, 14, "fizzbuzz", 16, 17, "fizz", 19,
        "buzz", "fizz", 22, 23, "fizz", "buzz", 26, 27, "fizz", 29, "fizzbuzz", 31, 32, "fizz", 34, "buzz", "fizz", 37,
        38, "fizz", "buzz", 41, "fizz", 43, 44, "fizzbuzz", 46, 47, "fizz", 49, "buzz", "fizz", 52, 53, "fizz", 55, 56,
        "fizz", 58, 59, "fizzbuzz", 61, 62, "fizz", 64, "buzz", "fizz", 67, 68, "fizz", "buzz", 71, "fizz", 73, 74,
        "fizzbuzz", 76, 77, "fizz", 79, "buzz", "fizz", 82, 83, "fizz", "buzz", 86, "fuzz", 88, 89, "fizzbuzz", 91, 92,
```

```
        "fizz", 94, "buzz", "fizz", 97, 98, "fizz", "buzz"]
print(list)

```

- AI code 1 (Code Pal)

```
def fizz_buzz(n):
    for i in range(1, n+1):
        if i % 3 == 0 and i % 5 == 0:
            print('FizzBuzz')
        elif i % 3 == 0:
            print('Fizz')
        elif i % 5 == 0:
            print('Buzz')
        else:
            print(i)

# Example usage:
n = 15
fizz_buzz(n)

```

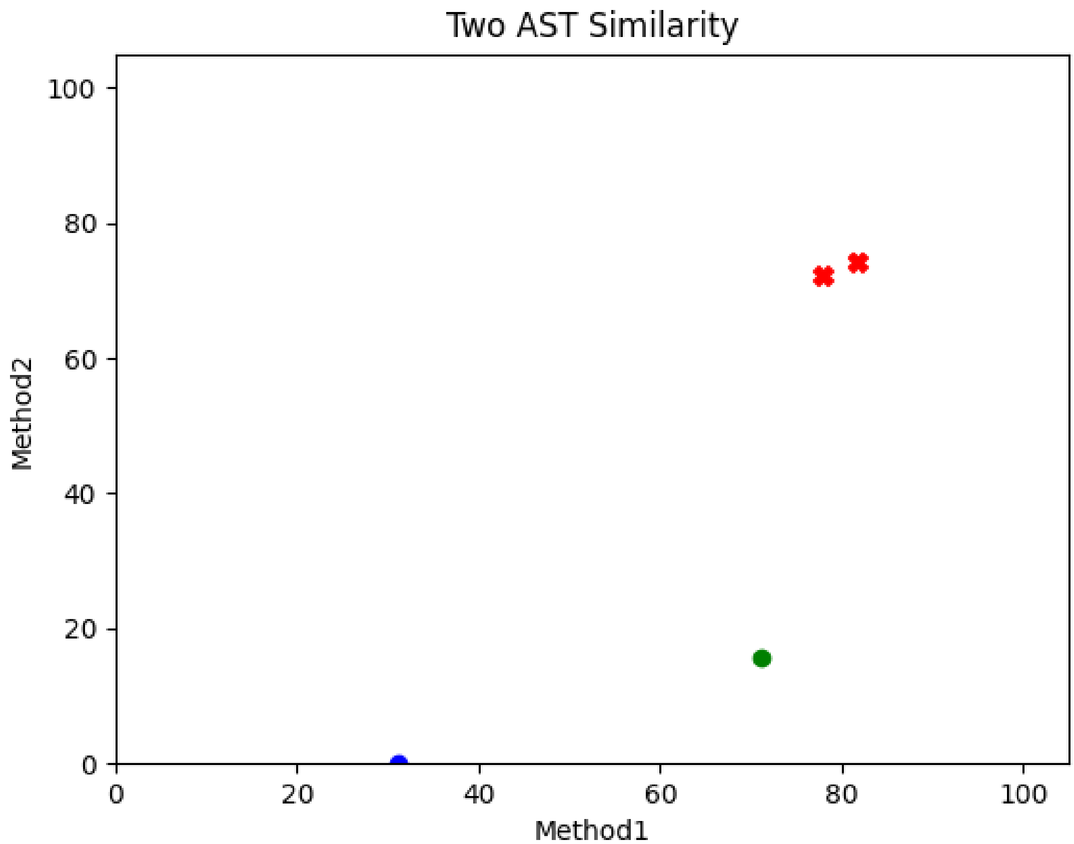
- AI code 2 (Replit)

```
n = 15

for i in range(1, n+1):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)

```

- Result




```
Base file is: codePal1.py
Compare files: ['James.py', 'Adorosa.py', 'BradyYanoff.py', 'replit1.py']
Method1 Similarity1: 71.12%
Method1 Similarity2: 77.84%
Method1 Similarity3: 31.11%
Method1 Similarity4: 81.78%
--------------------------------------------------
Method2 Similarity1: 15.63%
Method2 Similarity2: 72.11%
Method2 Similarity3: 0.00%
Method2 Similarity4: 74.20%
```

When comparing codes using two methods, one can say that the code is 100% similar, but the other shows a different similarity value, allowing more objective judgment. Both methods output 100% similarity when comparing precisely the same code. Still, due to algorithmic differences, when comparing even slightly different programming codes, they can only be similar.

CONCLUSION:

Exploring Python code analysis and Abstract Syntax Tree (AST) to detect cheating and plagiarism in programming assignments presents promising avenues for enhancing academic integrity in digital learning environments. This study used comparison methods and clustering techniques to provide educators with tools for identifying suspicious patterns indicative of AI-generated code.

The experimental work involved analyzing AI-generated code submissions and student submissions using AST-based comparisons with two different methods. By comparing the structural elements of code snippets, the study sought to quantify the similarity between submissions and identify potential instances of dishonesty.

In conclusion, no method could completely prevent cheating and plagiarism in programming. However, I think the possibilities have been fully shown. In the future, I passionately believe continued research and refinement of these methodologies are crucial for staying ahead of evolving deceptive practices and fostering a fair and equitable learning environment.

FUTURE WORK:

If I had six more months, I would build the program with the Graphic User Interface which provides user-friendly software. The program automatically detects error files that cannot be compiled before running. Currently, it has an error that if the file cannot be compiled on the machine, the comparing methods will not work. And I will label each data set with a filename or have the student's name appear, so the instructor clearly recognizes which data set belongs to whom.

References:

Paredes, P. S. (2020). *Comparing python programs using abstract syntax trees* (Doctoral dissertation, Uniandes).

Sidorov, G., Gómez-Adorno, H., Markov, I., Pinto, D., & Loya, N. (2015, August). Computing text similarity using tree edit distance. In *2015 Annual Conference of the North American Fuzzy Information Processing Society (NAFIPS) held jointly with 2015 5th World Conference on Soft Computing (WConSC)* (pp. 1-4). IEEE.