1) Aim:-

The aim of using Abstract Syntax Tree (AST) for plagiarism check is to analyze the structure and semantics of code or text in order to detect instances of copied or closely paraphrased content. The system can evaluate the underlying structure of various documents instead of just comparing plain text, which can result in false positives or misses, by creating an AST for each individual piece of code or text. This method makes it possible to detect plagiarism more precisely and can be particularly helpful in situations when the copied material has been significantly altered in order to avoid being picked up by more conventional methods of plagiarism detection.

2) ABSTRACT:-

a) Literature Survey:

Plagiarism detection has been an area of active research for several years, and a variety of approaches have been proposed to address this problem. One approach that has gained traction in recent years is the use of Abstract Syntax Tree (AST) for plagiarism detection. The following literature survey highlights some of the recent research in this area:

- "Using Abstract Syntax Trees to Detect Source Code Plagiarism" by Kevin W. Hamlen and Khaled M. Elleithy (2005)

This paper proposes an approach to detecting source code plagiarism using AST. The authors use AST to represent the structure and semantics of code

and then compare the trees to identify similarities between different programs.

- "AST-Based Plagiarism Detection for Programs with Complex Syntax" by Xiaodong Liu, Jian Zhang, and Zhiqiu Huang (2013)

This paper proposes an approach to AST-based plagiarism detection that can handle programs with complex syntax. The authors use a tree matching algorithm to compare ASTs and evaluate their approach on a dataset of Java programs.

- "A Lightweight Approach to Detecting Plagiarism in Text Documents using Abstract Syntax Trees" by Andrea De Lucia, Rita Francese, and Ignazio Passero (2014)

This paper proposes an approach to detecting plagiarism in text documents using AST. The authors generate ASTs for each document and then use tree comparison techniques to identify similarities between them.

- "Source Code Plagiarism Detection using Abstract Syntax Trees and Semantics-preserving Techniques" by Yang Liu, Wei Lu, and W.K. Chan (2015)

This paper proposes an approach to source code plagiarism detection using AST and semantics-preserving techniques. The authors use a tree edit distance algorithm to compare ASTs and evaluate their approach on a dataset of C programs.

b) Problem Statement:

In learning environments and other professions that place a high value on original thought and employment, plagiarism is a grave issue. Since they frequently ignore instances where plagiarised information has been marginally altered, traditional methods of plagiarism detection, including string matching, are frequently inadequate. The validity of the work being reviewed may be compromised by false positives or misses as a result. Programming language plagiarism detection also presents a hurdle because it is simple to modify the code to avoid plagiarism detection while keeping the same underlying functionality. As a result, a more advanced method of plagiarism detection is required, one that considers the structure and semantics of code alongside text. Particularly in circumstances where the copied information has been altered, using Abstract Syntax Tree (AST) to analyse the underlying structure of code and text might offer a more accurate and thorough method of detecting plagiarism. An abstract syntax tree represents all of the syntactical elements of a programming language, similar to syntax trees that linguists use for human languages. The tree focuses on the rules rather than elements like braces or semicolons that terminate statements in some languages. The tree is hierarchical, with the elements of programming statements broken down into their parts. For example, a tree for a conditional statement has the rules for variables hanging down from the required operator.The goal is to create an AST-based plagiarism detection system that is accurate and minimises false positives and misses while detecting plagiarism.

c) Design Techniques:

The techniques to create the Plagiarism Detector:

- **Tokenization:** Tokenizing the source code or text input is the first step in developing an AST plagiarism detector. In order to do this, the input must be divided into separate tokens, such as keywords, operators, and identifiers. The AST is then built using these tokens.
- **Construction of the AST:** After the input has been tokenized, the AST must be built. In order to describe the syntax and semantics of the code or language, it is necessary to parse the tokens into a hierarchical tree structure.
- **Tree Matching**: The system must compare the ASTs of two or more documents in order to find plagiarism. A tree matching technique is commonly used for this, which compares the nodes in each tree to find similarities and differences.
- **Thresholding:** The system must choose a threshold for the degree of similarity between the two ASTs in order to assess whether two documents are plagiarised. This limit can be established in accordance with the requirements of the application, such as the degree of uniqueness needed for a given activity.
- **Handling Code Obfuscation:** The system may employ methods like renaming and variable substitution to normalise the code before creating the AST in circumstances where copied code has been obfuscated. This can aid in lowering false positives and enhancing the precision of plagiarism detection.

d) Methodology:

1. Create the Abstract Syntax Tree:

   a. We create the AST of the two programmes .

2. a. Use filters to determine whether disregarding the variable name results in the same AST.

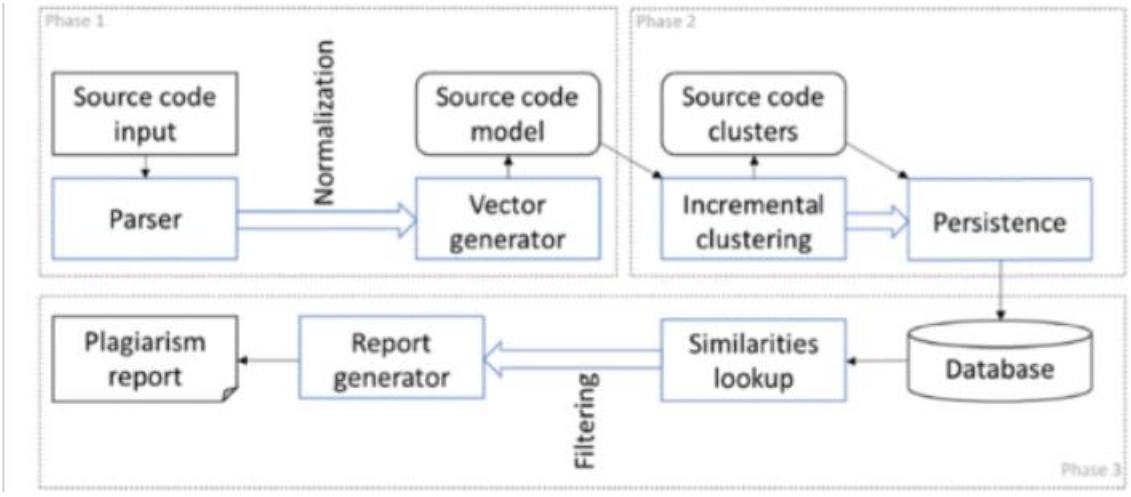
   b. Verify whether using the same AST when disregarding the function name

   c. Verify that both programmes have the same number of functions, classes, and arguments.

3. Display outcome

   a. The programmes are the same if all of the aforementioned flags are positive.

   b. Present results in an approachable manner.



e) Architecture of the Methodology:

The Abstract Syntax Tree (AST) of the plagiarism detector normally adheres to a predefined architecture. The system tokenizes the input first, breaking it down into individual words or symbols, before producing an AST for a piece of code or text. The syntax and semantics of the document are then represented via a hierarchical tree structure created by parsing these tokens.

Each node has a type that identifies the sort of element it represents and attributes that give the element more details. As an illustration, a node denoting a function might have properties defining the function's name, arguments, and return type.

Parent-child relationships that depict the hierarchical structure of the code or text are used to connect the nodes to one another. The document as a whole is represented by the root node, and each of its constituent parts is represented by a child node.

The system commonly utilises a tree matching technique to compare the tree structures of two ASTs when comparing them to look for plagiarism. This entails contrasting the parent-child relationships between the nodes as well as the kinds and properties of the nodes in each tree. To increase the precision of plagiarism detection, the system may additionally include other methods such subtree matching and normalisation.

3) Functionality of the Code:

- Use of efficient ways to process and represent source code because the amount of data stored grows over time and will be enormous.

- A scalable way to detect matches, as the number of works increases (the difficulty as well).

- Appropriate way of data persistence that the speed and efficiency of the search will depend on.

- Versatility of the system that guarantees use on different platforms, seamless access to data input and evaluation results.

4) Source Code:

data.py 9+

C: > Users > Gayathri > AppData > Local > Programs > Python > Python310 > lib > site-packages > nltk > data.py > ...

```
import ast
import ast
import code1
import code2
code1_contents=""
with open('code1.py') as f:
 code1_contents = f.read()
code2_contents=""
with open('code2.py') as f:
 code2_contents = f.read()
class toLower(ast.NodeTransformer):
 def visit_arg(self, node):
 return ast.arg(**{**node.__dict__, 'arg':str(0) }) def visit_Name(self, node):
 return ast.Name(**{**node.__dict__, 'id':str(0)}) # return ast.Name(**{**node.__dict__,
'id':node.id.lower()})
code1_final =
ast.unparse(toLower().visit(ast.parse(code1_contents)))
code2_final =
ast.unparse(toLower().visit(ast.parse(code2_contents)))
function_count=0
program1={}
program2={}
```

data.py 9+

C: > Users > Gayathri > AppData > Local > Programs > Python > Python310 > lib > site-packages > nltk > data.py > ...

```
23    def show_info(functionNode):
24     global function_count
25     program1[function_count]=len(functionNode.args.args)
26     function_count=function_count+1
27     print("Function name:", functionNode.name)
28     print("Args:")
29    for arg in functionNode.args.args:
30     #import pdb; pdb.set_trace()
31    print("\tParameter name:", arg.arg)
32    filename = "1.py"
33    with open(filename) as file:
34     node = ast.parse(file.read())
35    functions = [n for n in node.body if isinstance(n,
36    ast.FunctionDef)]
37    classes = [n for n in node.body if isinstance(n, ast.ClassDef)]for function in functions:
38     show_info(function)
39    for class_ in classes:
40     print("Class name:", class_.name)
41     methods = [n for n in class_.body if isinstance(n,
42    ast.FunctionDef)]
43     for method in methods:
44     show_info(method)
45    print("-----------------------")
46    function_count_2=0
47    def show_info(functionNode):
48     global function_count_2
49     function_coun_2=function_count+1
50     program2[function_count_2]=len(functionNode.args.args) print("Function name:", functionNode.name)
```

data.py 9+

C: > Users > Gayathri > AppData > Local > Programs > Python > Python310 > lib > site-packages > nltk > data.py > ...

```
50     program2[function_count_2]=len(functionNode.args.args) print("Function name:", functionNode.name)
51     print("Args:")
52    for arg in functionNode.args.args:
53     #import pdb; pdb.set_trace()
54    print("\tParameter name:", arg.arg)
55    filename = "2.py"
56    with open(filename) as file:
57     node = ast.parse(file.read())
58     functions = [n for n in node.body if isinstance(n,
59    ast.FunctionDef)]
60    classes = [n for n in node.body if isinstance(n, ast.ClassDef)]for function in functions:
61     show_info(function)
62    for class_ in classes:
63     print("Class name:", class_.name)
64     methods = [n for n in class_.body if isinstance(n,
65    ast.FunctionDef)]
66     for method in methods:
67     show_info(method)
68    print("-------------")
69    if((program1 == program2) and (code1_final==code2_final)): print("1.py and 2.py are same")
70    else:
71     print("1.py and 2.py are not same")
```

5) Validation:

a) Program 1:

```
def original():
    fake_name=["a","b","c"]
    for i in range(fake_name):
        print(i)
```

b) Program 2:

```
def copied():
    fake_name_chnaged=["a","b","c"]
    for j in range(fake_name_changed):
        print(j)
```

Output:

```
Function name: original
Args:
------------------------
Function name: copied
Args:
-------------
1.py and 2.py are same
```

Even if the functions and argument have changed, the program's results are the same, and both programmes' complexity is the same, hence the assertion can be concluded that both programmes are the same.

6) Conclusion:

An effective method for identifying plagiarism in text- and code-based materials is the Abstract Syntax Tree (AST). Achieving reliable comparison and difference detection between texts, AST creates a hierarchical tree structure that represents the syntax and semantics of the input. The integrity of original works can be effectively preserved by a plagiarism detection using AST by employing methods like tokenization, tree matching, and thresholding. Performance optimisation approaches can lessen the computational burden associated with the creation and comparison of ASTs. Overall, the application of AST to the detection of plagiarism shows significant promise for ensuring the originality and authenticity of code- and text-based creations.

7) References

1. Al-Khanjari, Z.A., Fiaidhi, J.A., Al-Hinai, R.A., Kutti, N.S.: Plagdetect: A java programming plagiarism detection tool. ACM Inroads 1(4), 66–71 (Dec 2010), http://doi.acm.org.ucd.idm.oclc.org/10.1145/1869746.1869766

2. Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C.: Sourcerer: a search engine for open source code supporting structure-based search. vol. 2006, pp. 681–682. ACM (2006)

3. Bukh, B., Ma, J.: Longest common subsequences in sets of words. SIAM Journal on Discrete Mathematics 28(4), 2042–2049 (2014)

4. Chilowicz, M., Duris, ., Roussel, G.: Viewing functions as token sequences to highlight similarities in source code. Science of Computer Programming 78(10), 1871– 1891 (2013)

5. Clough, P.: Plagiarism in natural and programming languages: an overview of current tools and technologies (2000)

6. Gitchell, D., Tran, N.: Sim: A utility for detecting similarity in computer programs. SIGCSE Bull. 31(1), 266–270 (Mar 1999), http://doi.acm.org.ucd.idm.oclc. org/10.1145/384266.299783

7. Greenan, K.: Method-level code clone detection on transformed abstract syntax trees using sequence matching algorithms. Student Report, University of California-Santa Cruz, Winter (2005)

9. Li, Y., Wang, L., Li, X., Cai, Y.: Detecting source code changes to maintain the consistence of behavioral model. pp. 1–6. ACM (2012) 10. Neamtiu, I., Foster, J., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Software Engineering Notes 30(4), 1–5 (2005)