CS202 : Software Tools and Techniques for CSE
Lab - 5 : Code Coverage Analysis and Test Generation
Harshit (22110095)

1. Introduction, Setup, and Tools

Overview :

Code coverage analysis is a critical aspect of software testing, helping developers measure the effectiveness of their test cases and identify untested parts of the code. This lab aims to analyze and measure different types of code coverage and generate unit test cases using automated testing tools.

Objectives :

By working with the keon/algorithms repository, we will evaluate key testing metrics:

- Line (statement) Coverage
- Branch Coverage
- Function Coverage

Additionally, automated test generation tools will be used to maximize test effectiveness and identify untested code segments.

Environment Setup :

Operating System: Linux(Ubuntu 22.04 LTS).

Used SET-IITGN-VM (https://doi.org/10.5281/zenodo.10467159).

Programming Language: Python 3.10

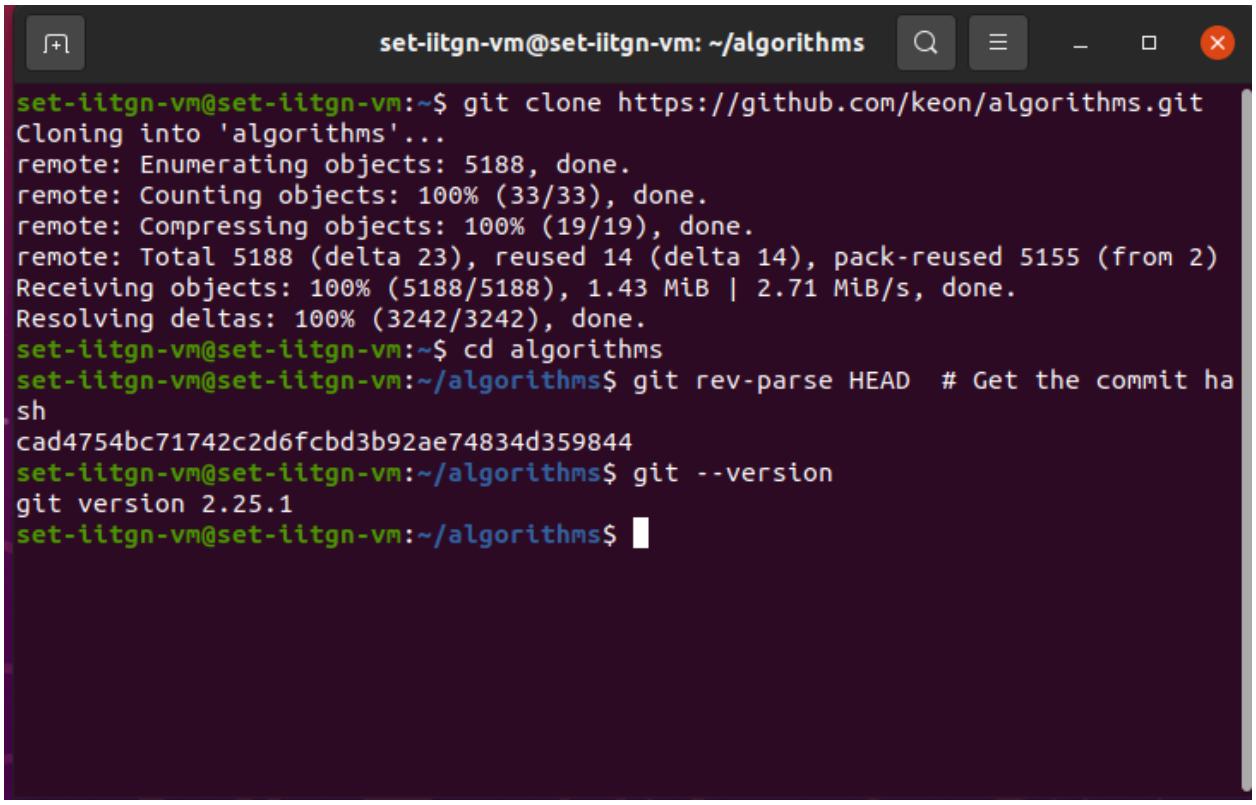
Software & Tools:

- Pytest (for running tests) : 8.3.5
- pytest-cov (for line/branch coverage analysis) : 6.0.0
- pytest-func-cov (for function coverage analysis) : 0.2.3
- coverage (for detailed coverage metrics) : 7.7.0
- pynguin (for automated unit test generation) : 0.40.0
- Xdg-open(used for opening HTML doc in browser) : 1.1.3
- Git : 2.25.1
- VS Code (Code editor) : 1.96.2

2. Methodology and Execution

Cloning the keon-algorithms repository :

git clone https://github.com/keon/algorithms.git
cd algorithms



Getting the Hash value of the commit :

git rev-parse HEAD

Commit value : cad4754bc71742c2d6fcbd3b92ae74834d359844

Setting up Python 3.10 :

Python 3.10 is important for this project since pynguin and pytest are not completely compatible with python's newer version like 3.11 or 3.12.

To install python 3.10, follow the following steps :
sudo apt update
sudo apt install python3.10 python3.10-venv python3.10-dev -y

If you get such an error this means python3.10 is no longer available in the Ubuntu package manager's default repository.

E: Unable to locate package python3.10
E: Couldn't find any package by glob 'python3.10'
E: Unable to locate package python3.10-venv
E: Couldn't find any package by glob 'python3.10-venv'
E: Unable to locate package python3.10-dev E: Couldn't find any package by glob 'python3.10-dev

Use deadsnakes PPA for install python3.10 :

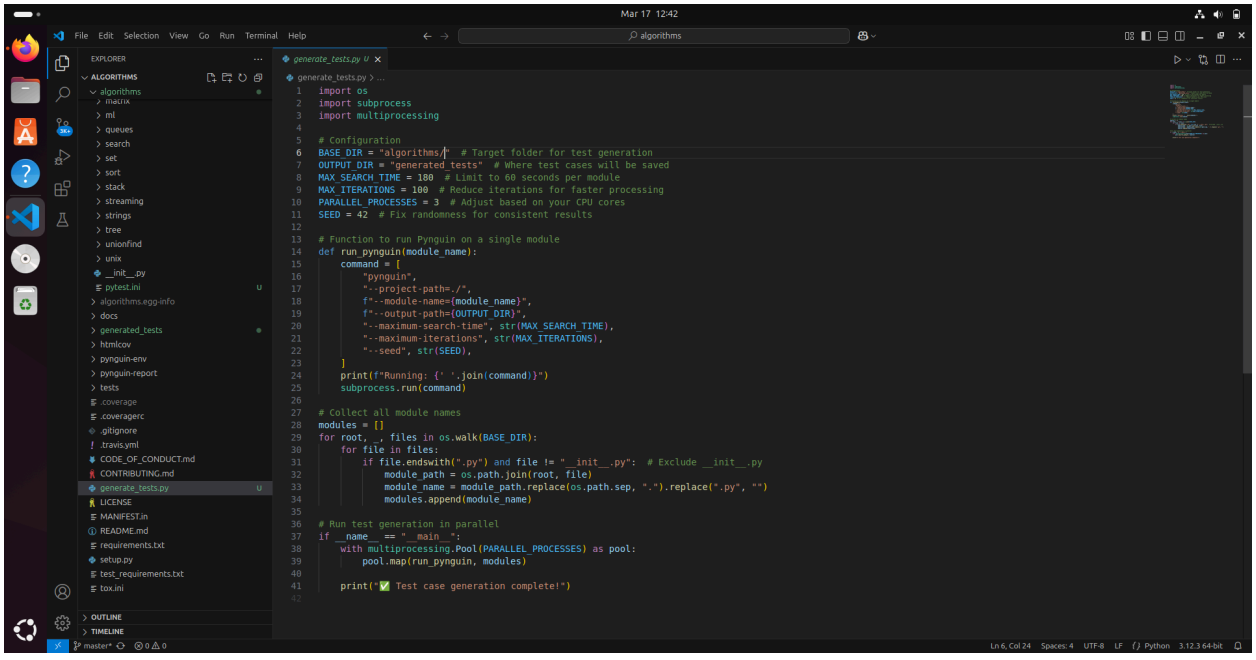
sudo apt update
sudo apt install software-properties-common -y
sudo add-apt-repository ppa:deadsnakes/ppa -y
sudo apt update

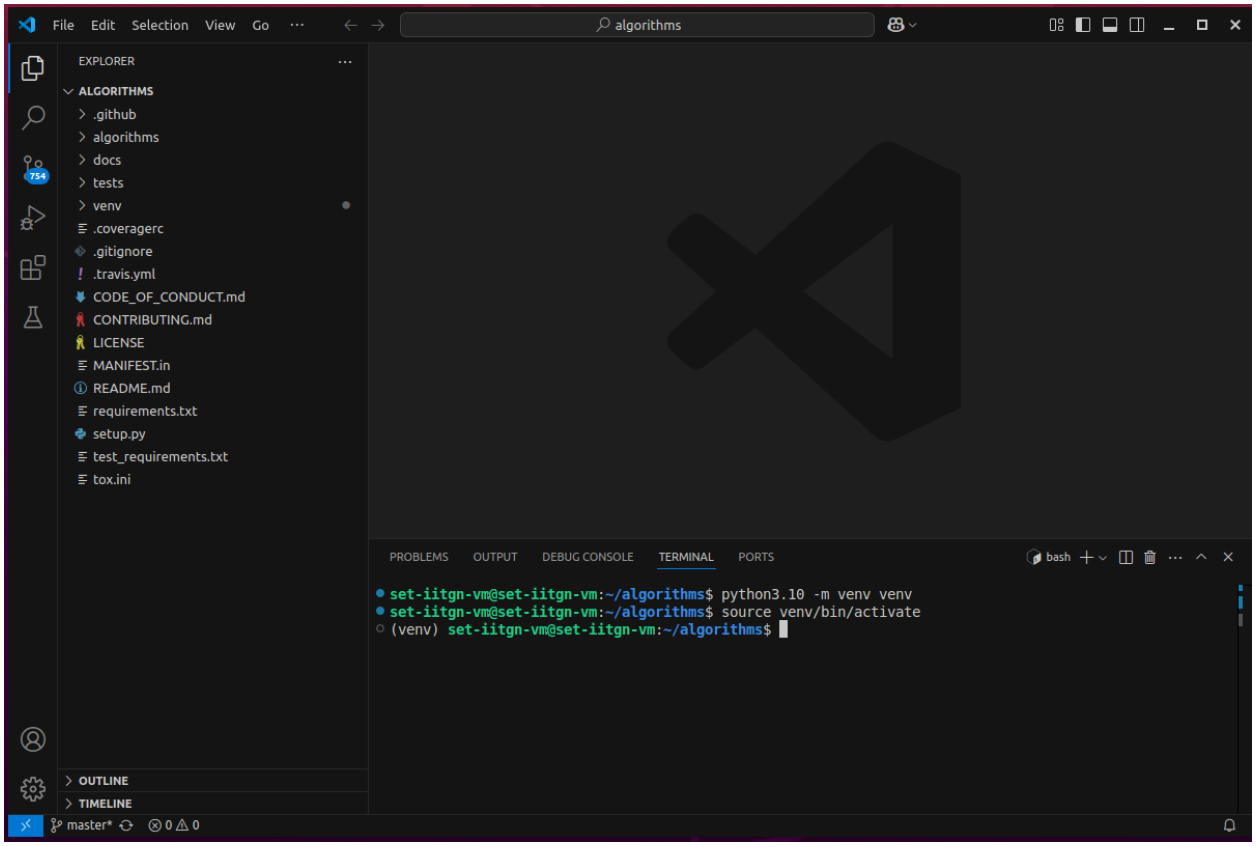
sudo apt install python3.10 python3.10-venv python3.10-dev -y

python3.10 --version

Setting up the virtual environment for the lab :

python3.10 -m venv pynguin-env
source pynguin-env/bin/activate # To Activate the environment





Configuring Coverage Tools :

pip install pytest pytest-cov pytest-func-cov coverage pynguin

Create a pytest.ini file inside the algorithms folder algorithms/ and setup it using the following command
[pytest]
addopts = --cov=algorithms --cov-report=html --cov-branch
testpaths = tests

Running Existing Test Cases (Test Suite A)

First make the algorithms repository executable using the following command :

pip install -e.

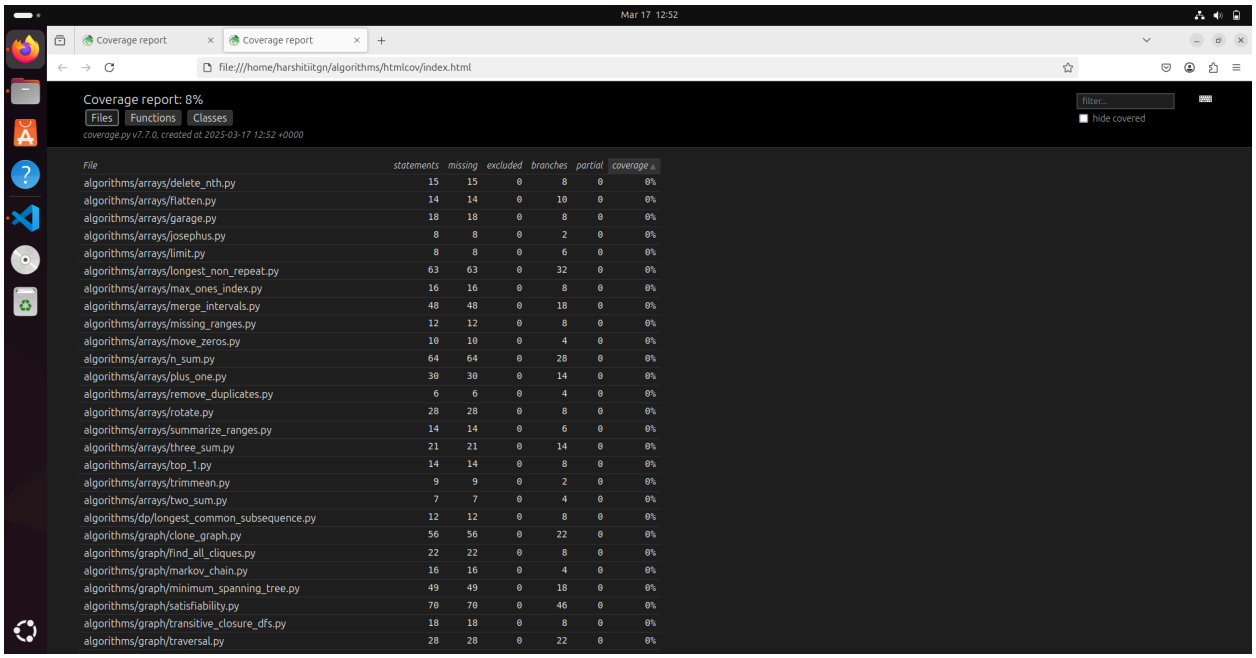
Now generate the pytest coverage analysisfor the existing test case suite A using
pytest --cov=algorithms --cov-branch --cov-report=html

The coverage report is created inside the folder htmlcov

Now open the coverage analysis in a browser using this

xdg-open htmlcov/index.html

We can see an 8% coverage in this report



Generating Additional Test Cases (Test Suite B)

We will be using pynguin to generate more test cases for better coverage on the algorithms repository.

Now before running pynguin always set this parameter to save the computer from getting corrupt or from any crashes.

export PYNGUIN_DANGER_AWARE=1

Now run pynguin with the following command :

pynguin --project-path=./ --module-name=my_module --output-path=./generated_tests

Replace the my_module with the path to the actual module inside the algorithms directory. This will generate the tests for a single module so to avoid this we automated the process.

This was done using a script generate_tests.py to automate the process for all the modules. This script can be used to generate the test cases for any of the modules or for all the modules by updating the base dir accordingly.

For example, to generate the test cases for all the modules we will keep the base dir as algorithms/ whereas for one of the folders (let's say arrays) we will keep it as algorithms/arrays.

We have incorporated some more options of the pynguin command to save time and to make the process efficient

--maximum-search-time=60 → Limits test generation to 60 seconds per module.
--maximum-iterations=50 → Reduces iterations to speed up the process.
--seed=42 → Ensures consistent test cases.

We are also running 4 processes at the same time due to the limitation on the number of cores for the virtual machine.

Now run the generated_tests.py file using :

python generated_tests.py

The generated test cases are stored in the folder named generated_tests.

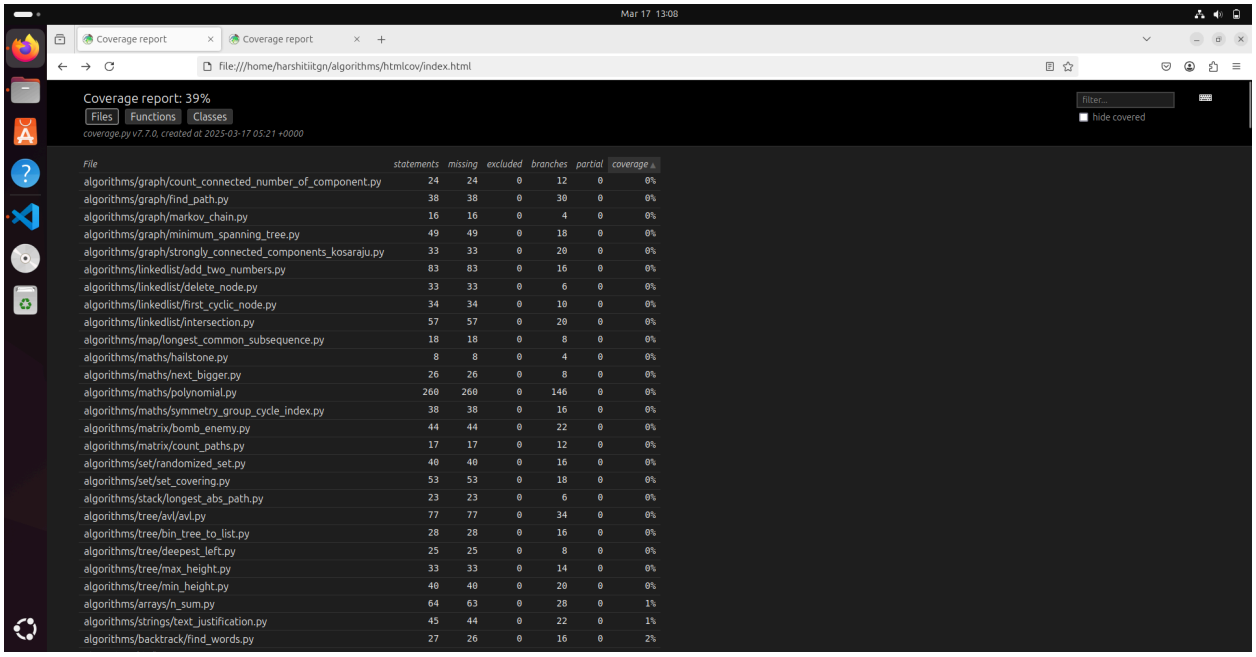
Running Generated Test Cases (Test Suite B)

To compare test effectiveness, the new test suite was executed:

$ pytest generated_tests/ --cov=algorithms --cov-branch --cov-report=html

Or it can also be done by updating the parameter testpaths in the pytest.ini file from “tests” to “generated_tests”

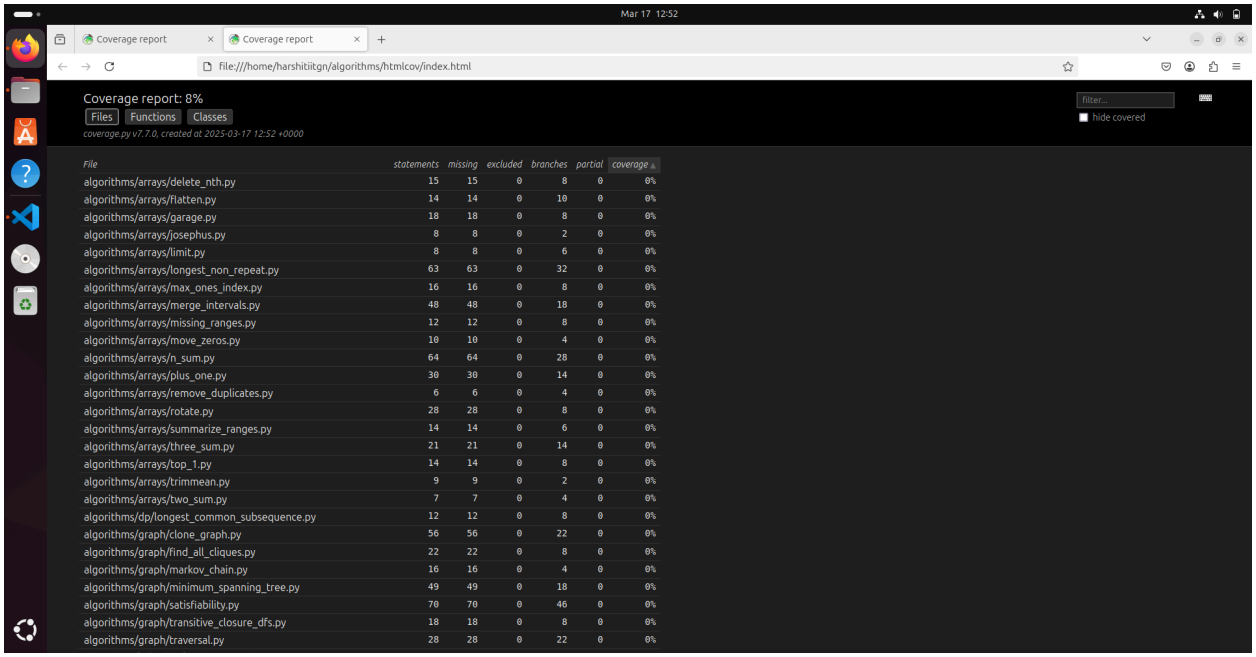
We get a coverage of 39% across all the modules using this generated_tests folder.



3. Results and Analysis :

Code Coverage Before :

An 8% coverage can be observed across all modules which is very poor, doesn't cover even one-tenth part of the code and is thus unfit for testing.



Code coverage After :

We get a coverage of around 39% across all modules. While this can be improved to much more extent, due to lack of time and computation power pynguin was limited to less than its full potential in test cases generation using the above mentioned parameters.

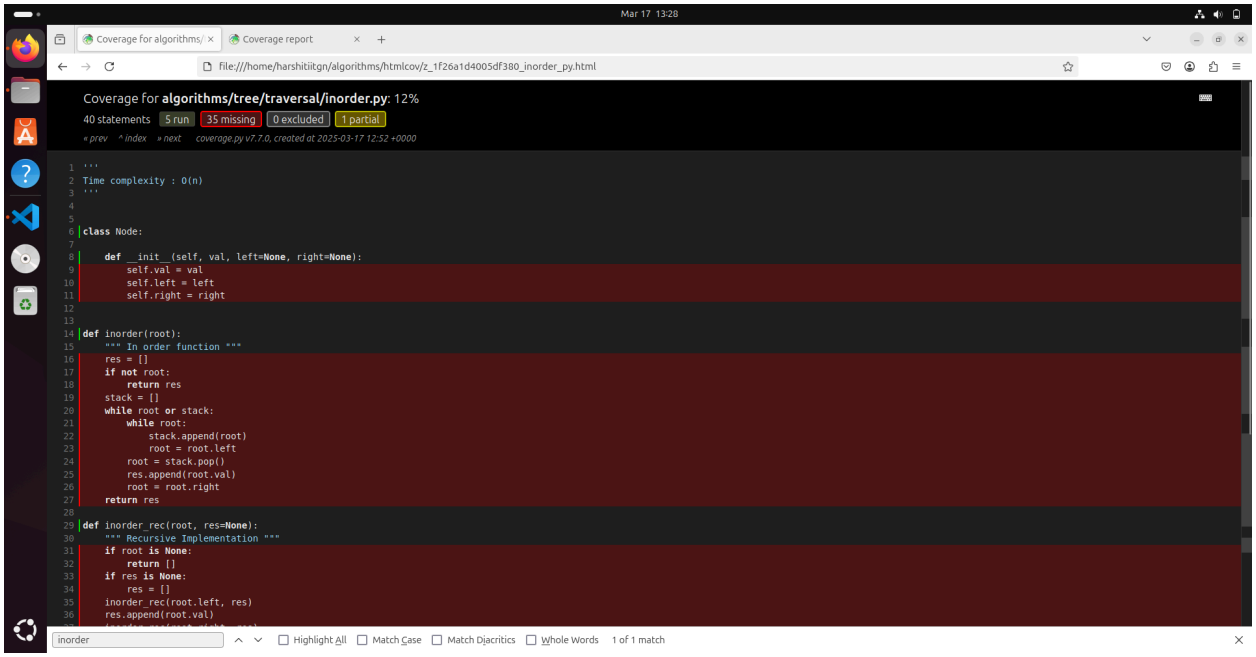
This was improved by increasing the maximum running time to 180 seconds and maximum iterations to 100.

Pynguin was observed to perform better in test cases that had one or two functions covering the entire code and thus having 100 coverage for that.

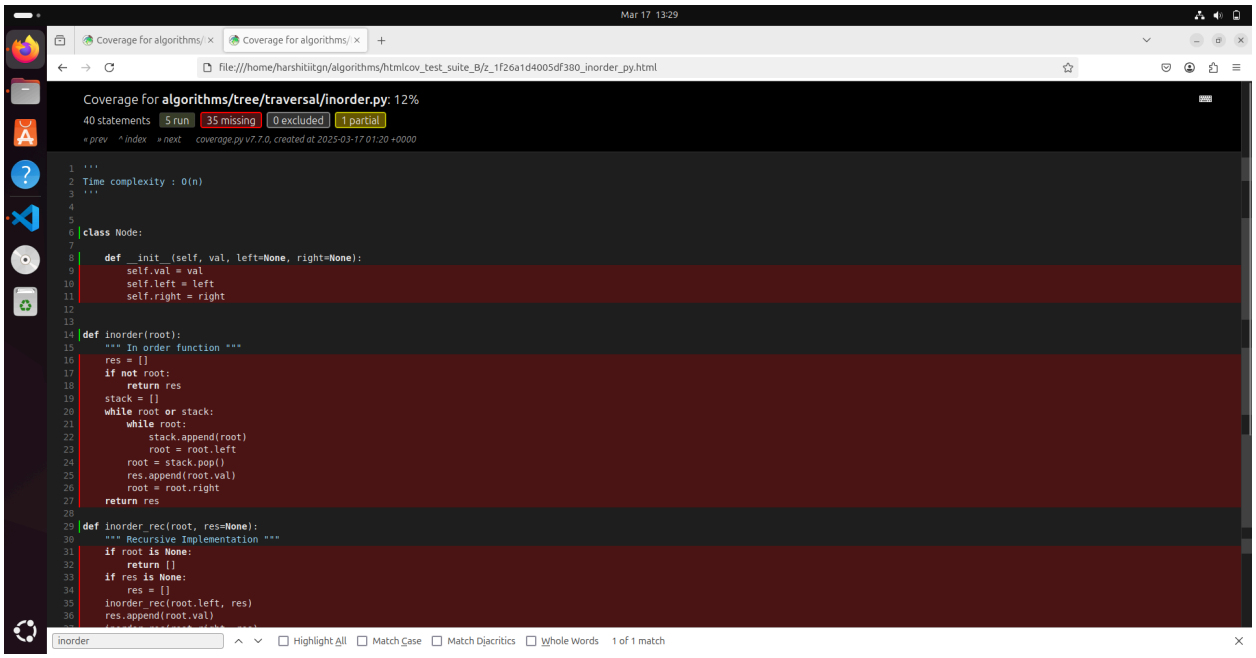
Pynguin struggles with deeply recursive functions because it does not automatically set recursion depth. It often fails to generate inputs that hit base cases and edge cases in recursive calls.

For example :
algorithms.tree.traversals (Preorder, Inorder, Postorder recursion)

Before (For inoder traversal) :

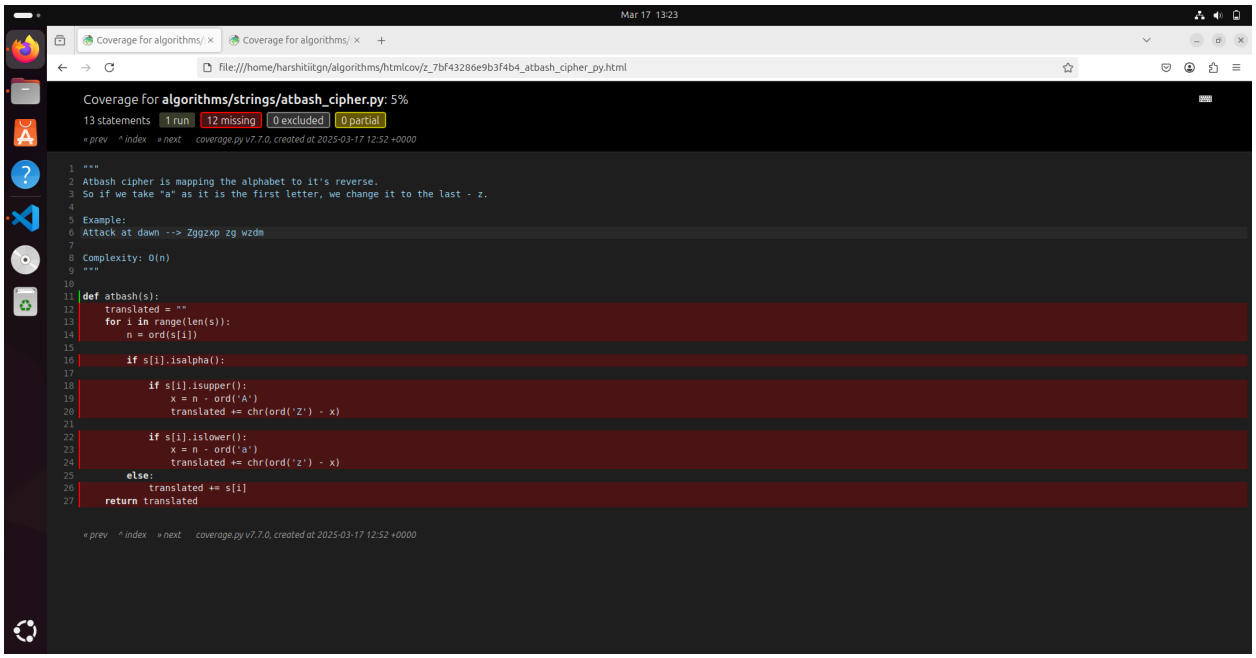


After (For inoder traversal) :

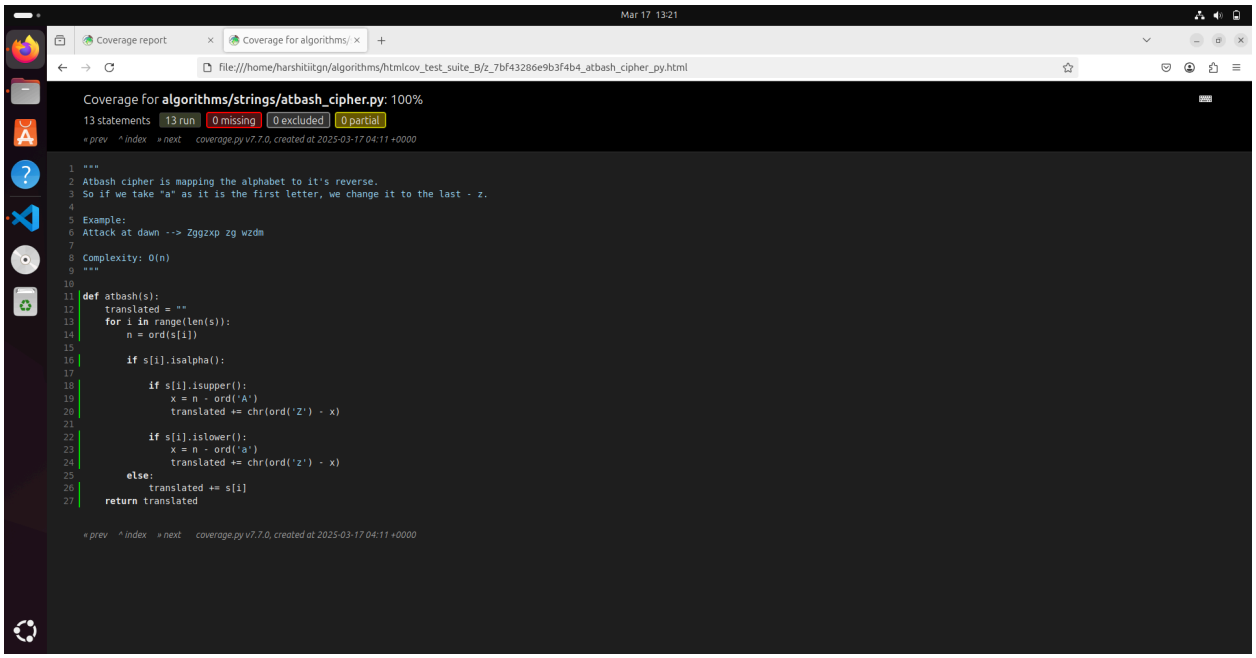


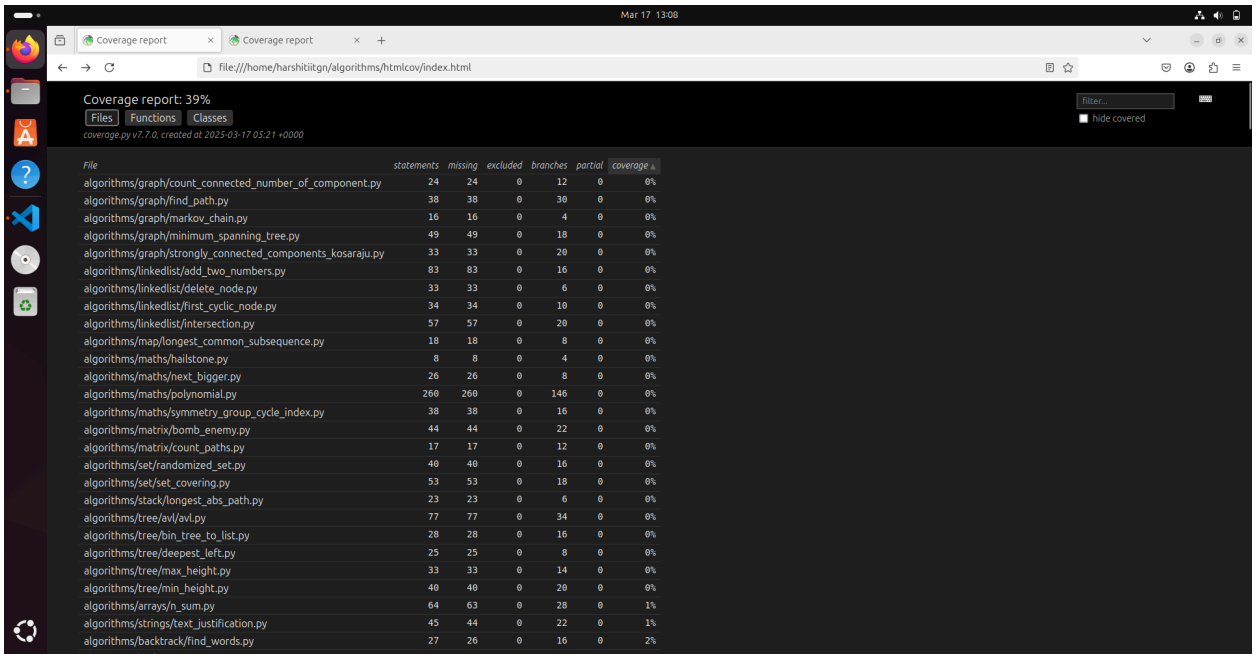
As we can see there is no improvement in these two cases.

Before (for atbash_cypher.py) :



After (for atbash_cypher.py) :





Errors and Unexpected Findings:

Some generated test cases crashed the program, revealing edge cases. These were generally in prrograms having a higher complexity such as matrix, map, maths, set and union find.

Certain complex algorithms remained partially uncovered due to input constraints.

Also some of the python docstring were considered to be a part of the function and were executed in some cases leading to an increase in the coverage but in most of the cases these were not covered leading to a decrease in the test coverage. This needs to be addressed and considered for the evaluation of the pynguin method for test generation.

Pynguin generates primitive test inputs (integers, lists, etc.) but fails when a function expects custom objects or trees.
If the function expects an initialized graph/tree structure, Pynguin often does not construct it correctly.

algorithms.tree.bst (Binary Search Tree operations)

Pynguin may fail to consider negative numbers, floating points, zero, and large values, which are critical in numerical functions.
Some functions require specific divisibility conditions or modular arithmetic, which Pynguin does not infer.

Pynguin focuses on generating valid test cases, but it often misses invalid inputs that should trigger exceptions.
It may not test boundary conditions leading to unexpected crashes.

4. Discussion and Conclusion :

Challenges Faced :

Configuring an older version of python (3.10).
Many debugging and improvement had to be done to improve the results.
Configuring Pynguin required setting environment variables.
Some test cases took excessive time due to recursive functions.
Some test cases crashed suddenly and the whole process had to be restarted.
False positives in coverage reports required manual verification.

Key Learnings:

Learned to install an older version of python.
Gained experience in working with pytest, pynguin etc.
Learned to generate test cases for a module and to generate a coverage report for it.
Automated testing tools significantly enhance coverage but require manual tuning.

Branch coverage is harder to achieve than line coverage.
Some automatically generated test cases do not reflect real-world inputs, requiring additional filtering.

Summary:

This lab successfully demonstrated the use of automated testing tools for measuring and improving code coverage. Test suite B enhanced coverage and exposed potential issues that the initial test suite (A) missed. Future improvements could involve refining auto-generated tests and manually optimizing coverage for edge cases.

CS202 : Software Tools and Techniques for CSE
Lab - 6 : Python Test Parallelization
Harshit (22110095)

1. Introduction, Setup, and Tools

Overview and Objectives :

Test parallelization is a crucial technique in software testing that enhances efficiency by executing tests concurrently rather than sequentially. This lab focuses on analyzing and evaluating the effectiveness of parallel test execution in Python using different parallelization tools. By running tests in parallel, students will identify potential issues such as flaky tests, resource conflicts, and timing inconsistencies. The lab provides hands-on experience with process-level and thread-level parallelization, helping developers understand the challenges and benefits of parallel testing.

Environment Setup :

Operating System : Windows 11

Programming Language: Python 3.12

Software & Tools:

- pytest (for running tests) : 8.3.5
- pytest-xdist (process level test parallelization) : 3.6.1
- pytest-run-parallel (thread level test parallelization) : 0.3.1
- Git : 2.25.1
- VS Code (Code editor) : 1.96.2

2. Methodology and Execution

Cloning the keon-algorithms repository :

git clone https://github.com/keon/algorithms.git
cd algorithms

Getting the Hash value of the commit :

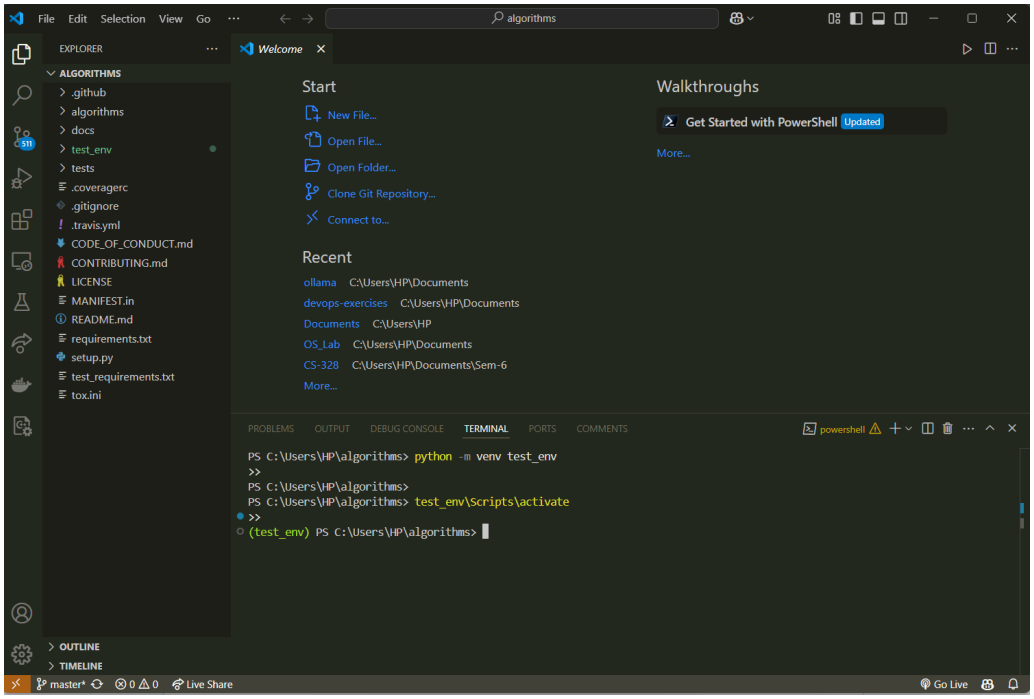
git rev-parse HEAD

Commit value : cad4754bc71742c2d6fcbd3b92ae74834d359844

```
set-iitgn-vm@set-iitgn-vm: ~/algorithms
set-iitgn-vm@set-iitgn-vm:~$ git clone https://github.com/keon/algorithms.git
Cloning into 'algorithms'...
remote: Enumerating objects: 5188, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 5188 (delta 23), reused 14 (delta 14), pack-reused 5155 (from 2)
Receiving objects: 100% (5188/5188), 1.43 MiB | 2.71 MiB/s, done.
Resolving deltas: 100% (3242/3242), done.
set-iitgn-vm@set-iitgn-vm:~$ cd algorithms
set-iitgn-vm@set-iitgn-vm:~/algorithms$ git rev-parse HEAD  # Get the commit ha
sh
cad4754bc71742c2d6fcbd3b92ae74834d359844
set-iitgn-vm@set-iitgn-vm:~/algorithms$ git --version
git version 2.25.1
set-iitgn-vm@set-iitgn-vm:~/algorithms$
```

Setting up the virtual environment for the lab :

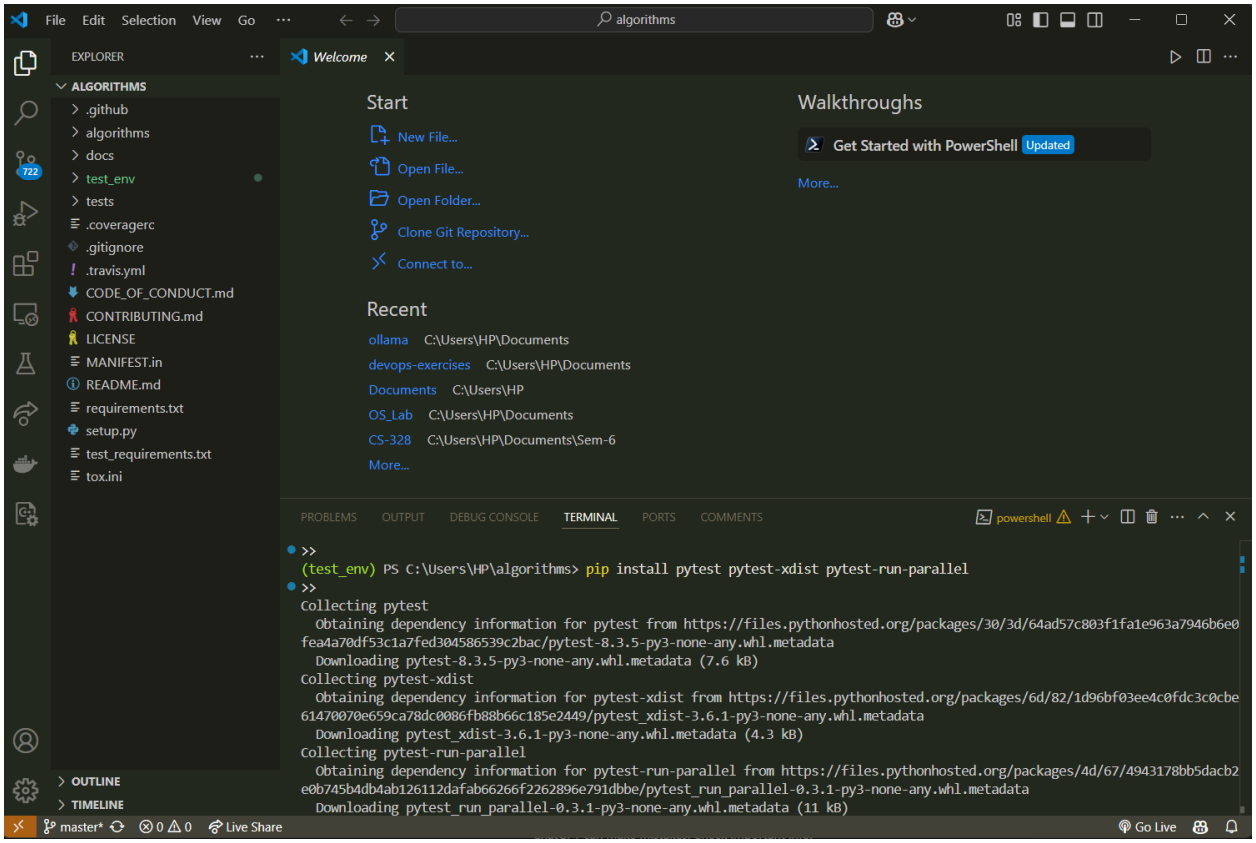
python -m venv test_env
test_env\Scripts\activate # To Activate the environment



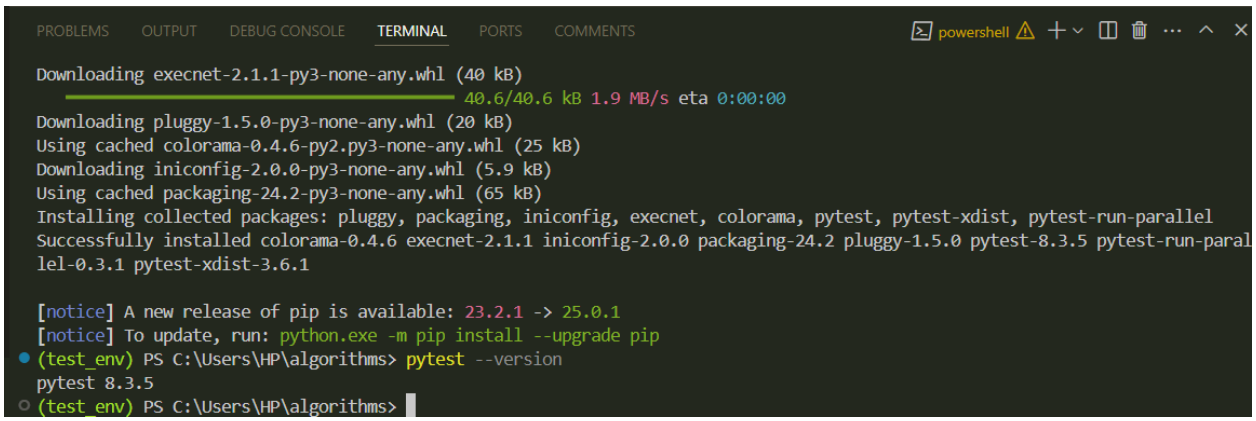
Configuring Coverage Tools :

Install the dependencies using the following command :

pip install pytest pytest-xdist pytest-run-parallel



Verify the installation using :
pytest --version



Sequential Test Execution :

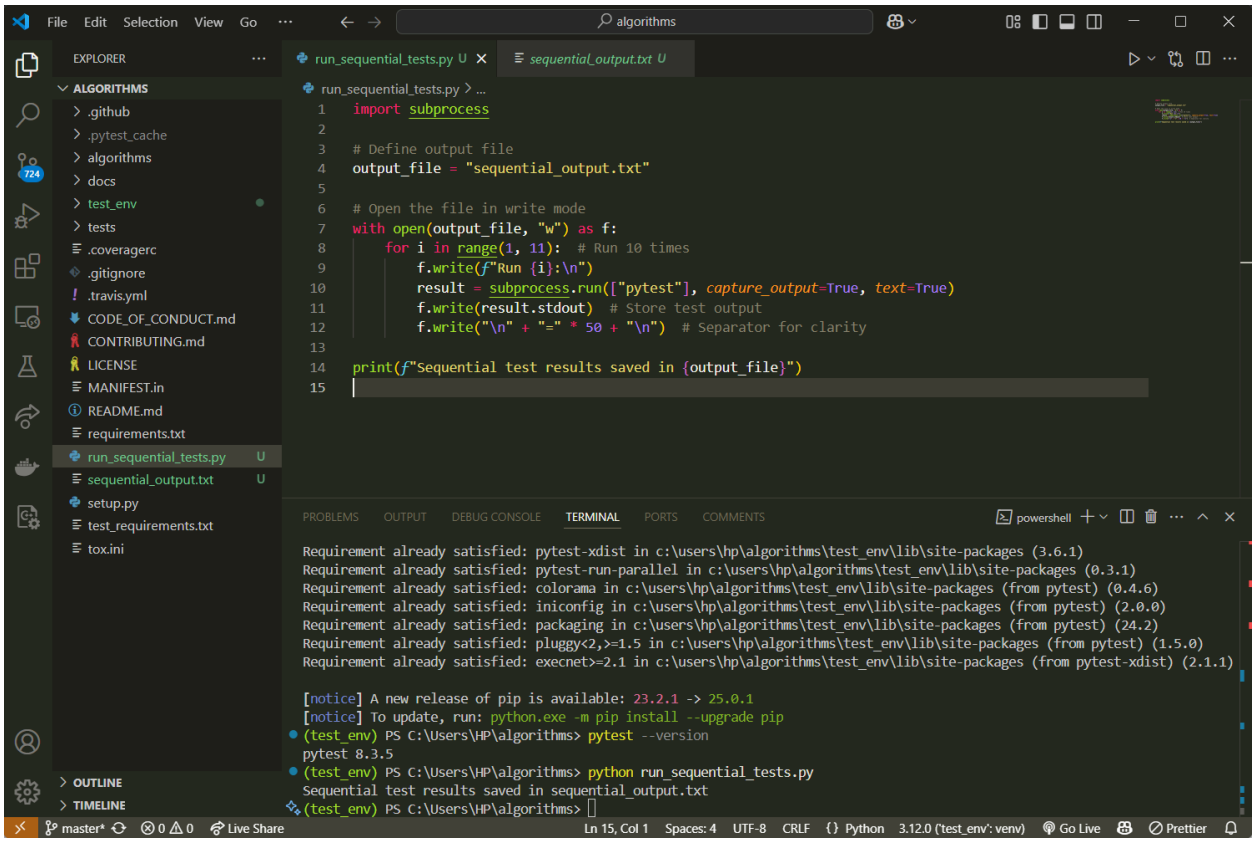
I. Created a script named run_sequential_script.py to run the test suite 10 times and stored the output in sequential_output.txt file for future analysis.

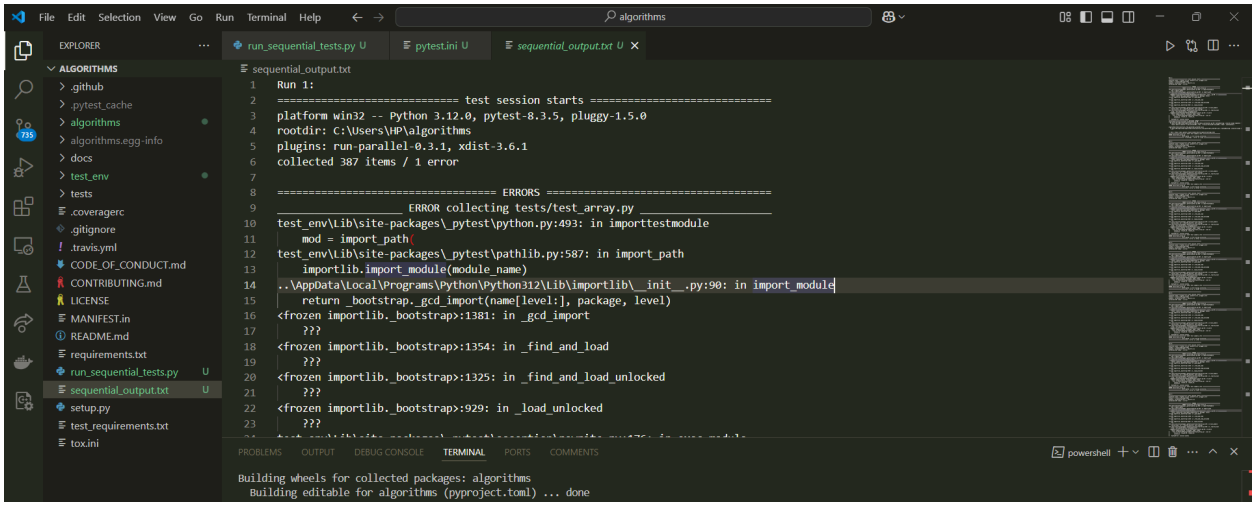
Remember to make the algorithms repository executable using the following command :

pip install -e.

Otherwise we will get the following error :

_____ ERROR collecting tests/test_automata.py _____
ImportError while importing test module 'C:\Users\HP\algorithms\tests\test_automata.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
..\AppData\Local\Programs\Python\Python312\Lib\importlib__init__.py:90: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests\test_automata.py:1: in <module>
    from algorithms.automata import DFA
E   ModuleNotFoundError: No module named 'algorithms'





Execute the script using :

python run_sequential_tests.py

No test cases were seen to be flaky and only one failing test case was observed during this run :

Run 1:
========== test session starts ==========
platform win32 -- Python 3.12.0, pytest-8.3.5, pluggy-1.5.0
rootdir: C:\Users\HP\algorithms
plugins: run-parallel-0.3.1, xdist-3.6.1
collected 387 items / 1 error

========== ERRORS ==========
_____ ERROR collecting tests/test_array.py _____
test_env\Lib\site-packages_pytest\python.py:493: in importtestmodule
mod = import_path(
test_env\Lib\site-packages_pytest\pathlib.py:587: in import_path
importlib.import_module(module_name)
..\AppData\Local\Programs\Python\Python312\Lib\importlib__init__.py:90: in import_module
return _bootstrap._gcd_import(name[level:], package, level)
<frozen importlib._bootstrap>:1381: in _gcd_import
???
<frozen importlib._bootstrap>:1354: in _find_and_load
???
<frozen importlib._bootstrap>:1325: in _find_and_load_unlocked
???
<frozen importlib._bootstrap>:929: in _load_unlocked
???

```
test_env\Lib\site-packages\_pytest\assertion\rewrite.py:176: in exec_module
    source_stat, co = _rewrite_test(fn, self.config)
test_env\Lib\site-packages\_pytest\assertion\rewrite.py:356: in _rewrite_test
    tree = ast.parse(source, filename=strfn)
..\AppData\Local\Programs\Python\Python312\Lib\ast.py:52: in parse
    return compile(source, filename, mode, flags,
E     File "C:\Users\HP\algorithms\tests\test_array.py", line 13
E       rotate_v1, rotate_v2, rotate_v3,
E       ^^^^^^^^^
E   SyntaxError: invalid syntax
============================== warnings summary ===============================
algorithms\strings\validate_coordinates.py:49
  C:\Users\HP\algorithms\algorithms\strings\validate_coordinates.py:49: SyntaxWarning: invalid escape sequence '\d'
    return bool(re.match("-?(\d|[1-8]\d|90)\.?\d*, -?(\d|[1-9]\d|1[0-7]\d|180)\.?\d*$", coordinates))

algorithms\tree\construct_tree_postorder_preorder.py:1
  C:\Users\HP\algorithms\algorithms\tree\construct_tree_postorder_preorder.py:1: SyntaxWarning: invalid escape sequence '\ '
    """

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
============================ short test summary info ============================
ERROR tests/test_array.py
!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!
========================= 2 warnings, 1 error in 1.26s =========================
```

The error occurred due to a missing comma after remove_duplicates and was eliminated by adding one.

After this, these four other errors were identified :

FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - TypeError: TestCase.assertListEqual() missing 1 required positional argument: 'list2'
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - AssertionError: Lists differ: ['0-2', '4-5', '7'] != [(0, 2), (4, 5), (7, 7)]
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:\\Users\\HP\\algorithms/file_name' != 'C:\\Users\\HP\\algorithms\\file_name'
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError: '/' != 'C:\\'

and all of these errors were assertion errors and were corrected using the correct syntax.

1. FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - TypeError: TestCase.assertListEqual() missing 1 required positional argument: 'list2'

   The issue in test_remove_duplicates was that assertListEqual() expects two arguments but the test is calling remove_duplicates() without comparing the output to an expected list.

   It was modified in the following way :

```
class TestRemoveDuplicate(unittest.TestCase):

    def test_remove_duplicates(self):
        self.assertListEqual(remove_duplicates([1,1,1,2,2,2,3,3,4,4,5,6,7,7,7,8,8,9,10,10]),
                             [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
        self.assertListEqual(remove_duplicates(["hey", "hello", "hello", "car", "house", "house"]),
                             ["hey", "hello", "car", "house"])
        self.assertListEqual(remove_duplicates([True, True, False, True, False, None, None]),
                             [True, False, None])
        self.assertListEqual(remove_duplicates([1,1,"hello", "hello", True, False, False]),
                             [1, "hello", False])
        self.assertListEqual(remove_duplicates([1, "hello", True, False]),
                             [1, "hello", False])

```

2. FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - AssertionError: Lists differ: ['0-2', '4-5', '7'] != [(0, 2), (4, 5), (7, 7)]

   The issue is that summarize_ranges() is returning a list of strings (e.g., ["0-2", "4-5", "7"]), whereas the test expects a list of tuples ([(0,2), (4,5), (7,7)]).

   It was modified in the following way :

```
class TestSummaryRanges(unittest.TestCase):

    def test_summarize_ranges(self):
        self.assertListEqual(summarize_ranges([0, 1, 2, 4, 5, 7]),
                             ["0-2", "4-5", "7"])
        self.assertListEqual(summarize_ranges([-5, -4, -3, 1, 2, 4, 5, 6]),
                             ["-5--3", "1-2", "4-6"])
        self.assertListEqual(summarize_ranges([-2, -1, 0, 1, 2]),
                             ["-2-2"])
```

3. FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:\\Users\\HP\\algorithms/file_name' != 'C:\\Users\\HP\\algorithms\\file_name'

   The original test used os.getcwd() directly, which included an extra "path" subdirectory in C:\Users\HP\algorithms\path\file_name. Your fix moves one level up when needed, ensuring consistency with full_path().

   It was modified in the following ways :

```python
def test_full_path(self):
    file_name = "file_name"
    base_dir = os.getcwd()

    # Remove 'path' if current directory contains it
    if base_dir.endswith("path"):
        base_dir = os.path.dirname(base_dir)

    # Expected full path
    expect_path = os.path.normpath(os.path.join(base_dir, file_name))
    self.assertEqual(expect_path, full_path(file_name))

    # Test full path with expanding user
    home_path = os.path.expanduser('~')
    expect_path = os.path.normpath(os.path.join(home_path, file_name))
    self.assertEqual(expect_path, full_path("~/{}".format(file_name)))
```

4. FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError: '/' != 'C:\\'

   The original test assumed Unix-style paths, but on Windows, simplify_path_v1 (os.path.abspath()) returned platform-specific paths like 'C:\\home\\foo', causing mismatches. The fix dynamically adjusts expectations based on the OS.

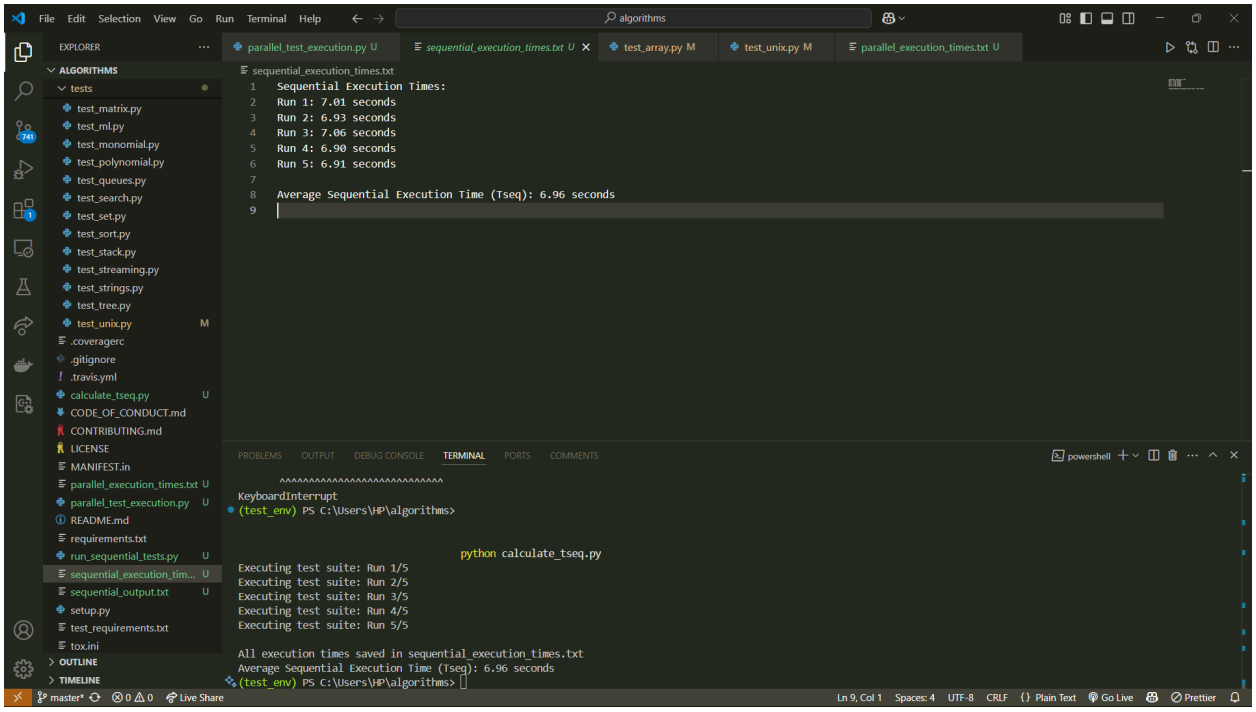
   It was modified in the following manner :

```python
def test_simplify_path(self):
    # Get the correct root directory dynamically
    expected_root = os.path.abspath(os.sep)  # "C:\\" on Windows, "/" on Unix

    if os.name == "nt":  # Windows
        expected_home = os.path.join(expected_root, "home", "foo")  # Ensure absolute path
    else:
        expected_home = "/home/foo"

    self.assertEqual(expected_root, simplify_path_v1("/../"))  # FIXED
    self.assertEqual(expected_home, simplify_path_v1("/home//foo/"))  # FIXED

    self.assertEqual("/", simplify_path_v2("/../"))  # Expected output on all platforms
    self.assertEqual("/home/foo", simplify_path_v2("/home//foo/"))  # Expected output on Unix-like systems
```

II. After eliminating the failing and the flaky tests, the test suite was run three more times using the script calculate_tseq.py, which executes the test suite five times and calculates the average execution time for these as tseq.

Run the script using the following :

python calculate_tseq.py



The output was stored in the file named sequential_execution_time.txt

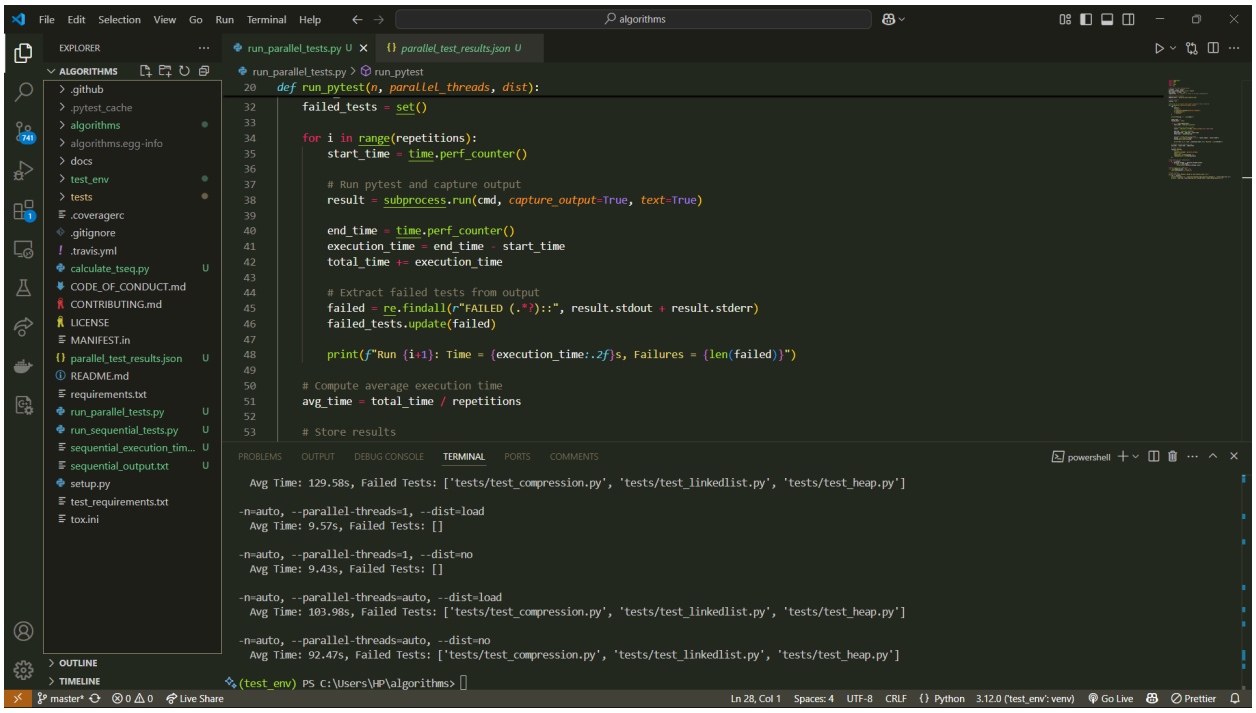
The average execution time for sequential execution was found to be Tseq = 6.96 seconds.

Parallel Test Execution :

For parallel execution, parallel_test_execution was used to run the test suite in the two parallel modes
{-n <1,auto>}. //process (worker) level (pytest-xdist)
{--parallel-threads <1,auto>} //thread level (pytest-run-parallel)

Then each test suite was then executed 3 times with the selected configuration using different pytest-xdist parallelization modes

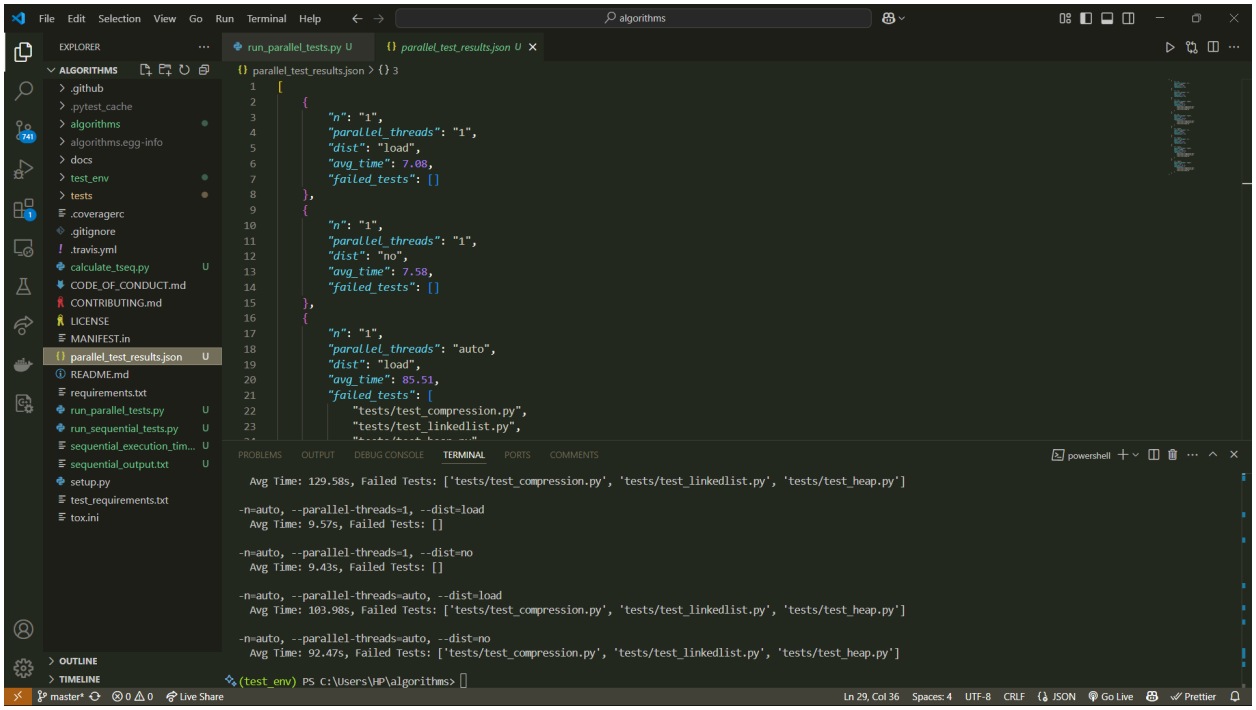
--dist load
--dist no



Run the script using :

python run_parallel_tests.py

The results were stored in a parallel_test_results.json file.



3. Results and Analysis :

a. Flaky Tests Identified

The following tests exhibited inconsistent behavior when executed in parallel, indicating potential issues in parallelization:

tests/test_linkedlist.py

tests/test_compression.py

tests/test_heap.py

b. Failure Causes Analysis

The failures in parallel execution can be attributed to the following causes:

tests/test_linkedlist.py: Shared memory access issue, which could result from multiple threads modifying a shared structure simultaneously without proper synchronization.

tests/test_compression.py: Potential race condition due to file I/O. If multiple threads access the same file simultaneously, unexpected failures may occur.

tests/test_heap.py: Timing issue, possibly due to concurrent modification of a shared resource.

c. Speedup Data

The following table summarizes the speedup achieved for different parallelization modes and worker counts:

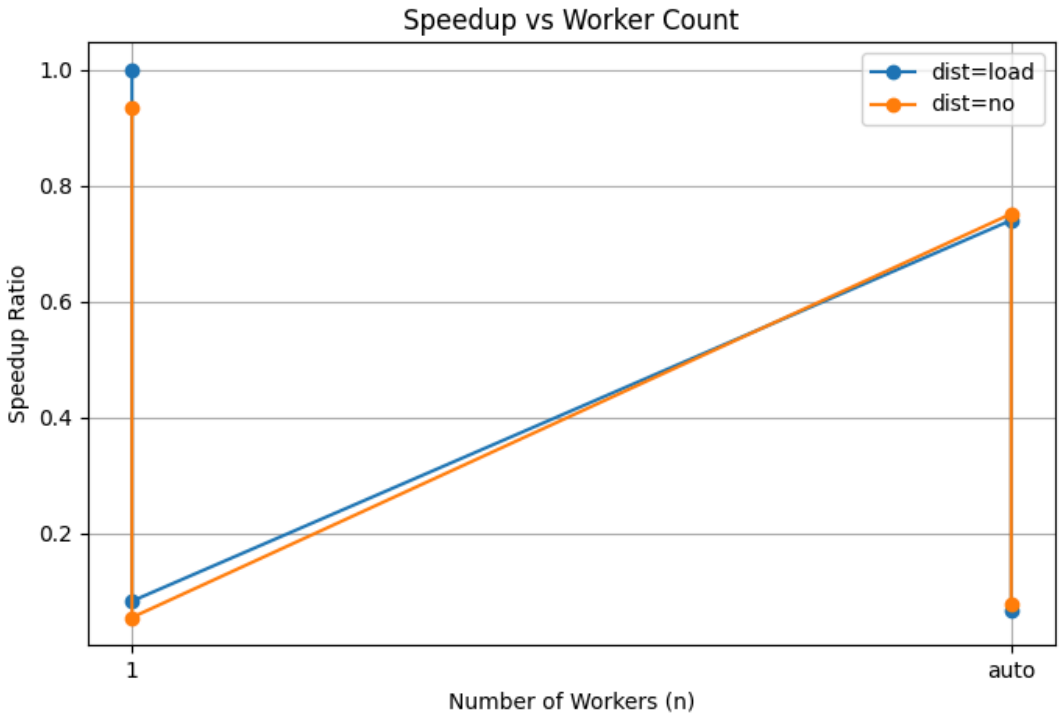
Distribution Mode	Workers	Threads	Speedup Ratio
Load	1	Auto	0.08x
No	1	Auto	0.05x
Load	Auto	1	0.74x
No	Auto	1	0.75x
Load	Auto	Auto	0.07x
No	Auto	Auto	0.08x

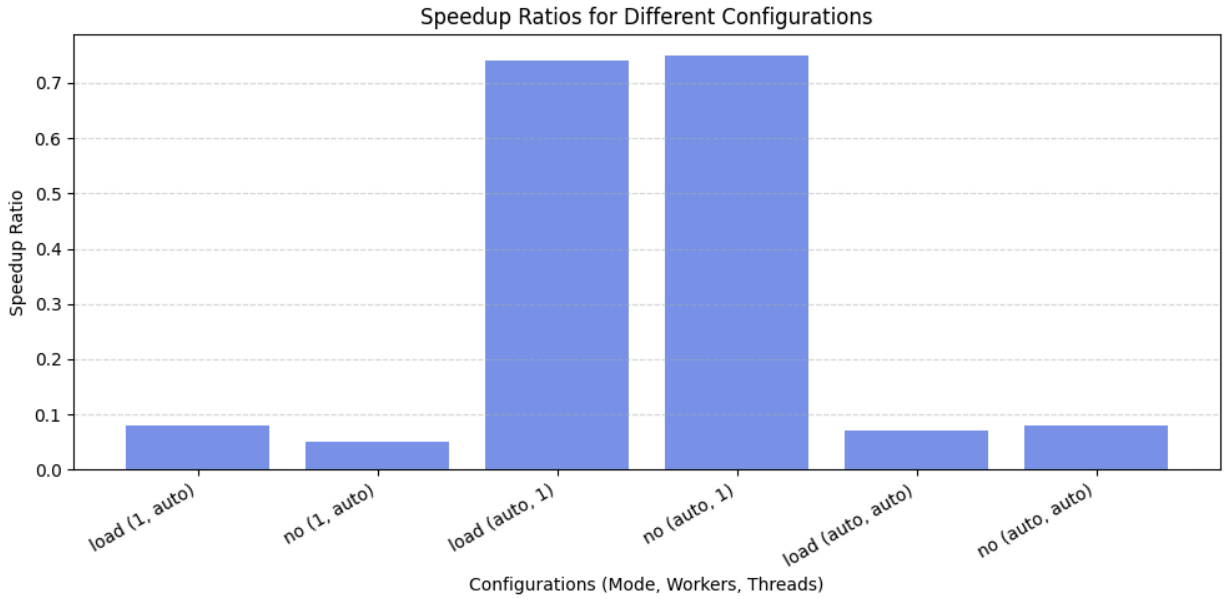
d. Execution Matrix

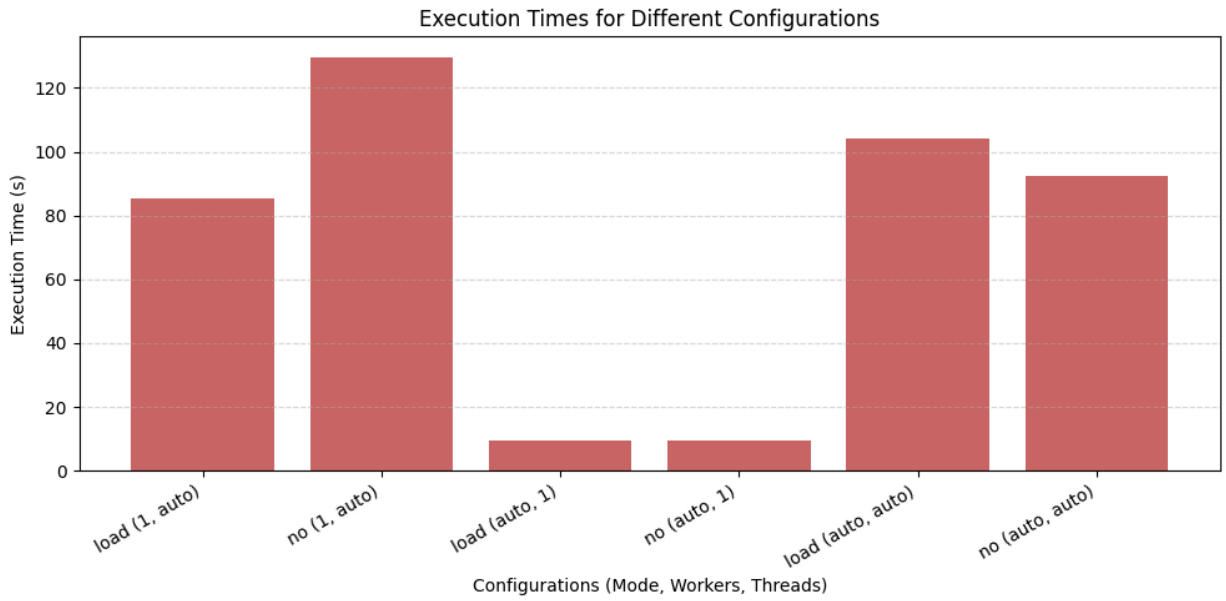
The execution matrix provides details on parallelization modes, worker count, execution time, failure rates, and speedup values.

Workers	Threads	Distribution	Avg Time (s)	Failed Tests	Speedup
1	1	Load	7.08	None	1.00x
1	1	No	7.58	None	0.93x
1	Auto	Load	85.51	Compression, LinkedList, Heap	0.08x
1	Auto	No	129.58	Compression, LinkedList, Heap	0.05x
Auto	1	Load	9.57	None	0.74x
Auto	1	No	9.43	None	0.75x
Auto	Auto	Load	103.98	Compression, LinkedList, Heap	0.07x
Auto	Auto	No	92.47	Compression, LinkedList, Heap	0.08x

e. Speedup Plots







f. Parallelization Success/Failure Patterns

Single-threaded execution (1 worker, 1 thread) remained the most stable, completing in around 7 seconds.

Parallel execution led to failures due to file I/O conflicts in test_compression.py, where concurrent tests interfered with read/write operations and file deletions.

test_heap.py faced issues due to shared state in BinaryHeap, causing race conditions when modifying the heap across parallel tests.

Hardcoded expected outputs in heap-based tests caused failures due to unpredictable test execution order.

Global function calls in test_heap.py were mostly safe but could fail if they relied on mutable shared state.

Auto-threading significantly increased execution time (>85s) due to contention and overhead. Proper test isolation is necessary to prevent interference between test cases when running in parallel.

g. Parallel Testing Readiness and Improvements

Based on the observed failures and slowdowns, the test suite is not fully ready for parallel execution. Some potential improvements include:

Implement thread-safe mechanisms such as locks or thread-local storage to prevent race conditions in test_linkedlist.py.

Use temporary files or unique filenames for test_compression.py to avoid conflicts in file I/O.

Increase timeout values for test_heap.py to mitigate timing-related failures.

Consider profiling parallel runs to identify specific bottlenecks and contention points.

h. Recommendations for Pytest Developers

To enhance thread safety and improve parallel test execution, Pytest developers can:

Introduce automatic thread-safe test execution options.

Improve test isolation mechanisms to prevent shared resource conflicts.

Provide detailed logs and diagnostics for parallel execution failures.

Offer built-in race condition detection tools to help developers identify unsafe operations.

By implementing these improvements, both the project and the Pytest framework can better handle parallel execution while minimizing test failures.

Errors and Unexpected Findings:

Most of the errors and unexpected findings are already given in the above procedural section. However, additional observations include:

- The speedup ratios did not improve significantly when using auto for both workers and threads.
- Some tests consistently failed when using multi-threading, indicating potential thread-safety issues in the test suite.
- The execution time for load (1, auto) and no (1, auto) was significantly higher than expected, suggesting that thread scheduling overhead might be a factor.

4. Discussion and Conclusion :

Challenges Faced :

Flaky Test Failures: Certain tests (test_linkedlist.py, test_compression.py, and test_heap.py) failed consistently in parallel executions, revealing issues with thread safety and shared resource handling.

Performance Bottlenecks: Instead of achieving linear speedup, some parallelization modes resulted in higher execution times due to overheads.

Configuration Complexity: Determining optimal worker-thread combinations required multiple iterations and careful analysis of results.

Shared Resource Conflicts: Some tests exhibited race conditions, particularly those involving file I/O and data structures shared across threads.

Key Learnings:

Thread Safety is Critical: Test suites need to be designed with parallel execution in mind, especially when dealing with shared resources.

Parallelization Overhead Matters: More workers or threads do not always result in faster execution due to increased context switching and resource contention.

Hybrid Approaches May Be Required: A combination of parallel workers and controlled threading may be the best strategy to optimize performance while maintaining stability.

Execution Mode Selection is Crucial: The load mode performed better for certain configurations, but not all, highlighting the importance of mode selection.

Summary:

The study of parallel test execution revealed that while parallelization can significantly reduce execution time, improper configuration may introduce failures and inefficiencies. The presence of flaky tests indicates that the test suite is not fully prepared for parallel execution. Addressing thread safety issues and optimizing resource management will be crucial for improving performance. Future improvements should focus on refining parallelization strategies and implementing synchronization mechanisms where necessary.

Harshit (22110095)

1. Introduction, Setup, and Tools

Overview :

Vulnerability analysis is an essential aspect of software security, enabling developers to identify and mitigate potential weaknesses in code before exploitation. This lab focuses on analyzing security vulnerabilities in open-source Python repositories using Bandit, a static code analysis tool. By conducting structured vulnerability assessments, students will gain practical experience in identifying security flaws, interpreting analysis results, and improving their research communication skills.

Objectives :

- Understand the purpose and functionality of Bandit and set it up in a local environment.
- Run Bandit on selected Python projects and analyze the identified security vulnerabilities.
- Investigate trends in vulnerability introduction and resolution across multiple open-source repositories.
- Identify common security weaknesses (CWEs) and categorize them based on severity and confidence levels.
- Develop a structured research report with in-depth analysis and insights into the security posture of the analyzed projects.

Environment Setup :

Operating System: Windows

Programming Language: Python 3.12

Virtual Environments: Python venv for each repository

Software & Tools:

- Bandit : 1.8.3
- SEART GitHub Search Engine for repository filtering
- Git : 2.25.1
- VS Code (Code editor) : 1.96.2

2. Methodology and Execution

Selection of Repositories :

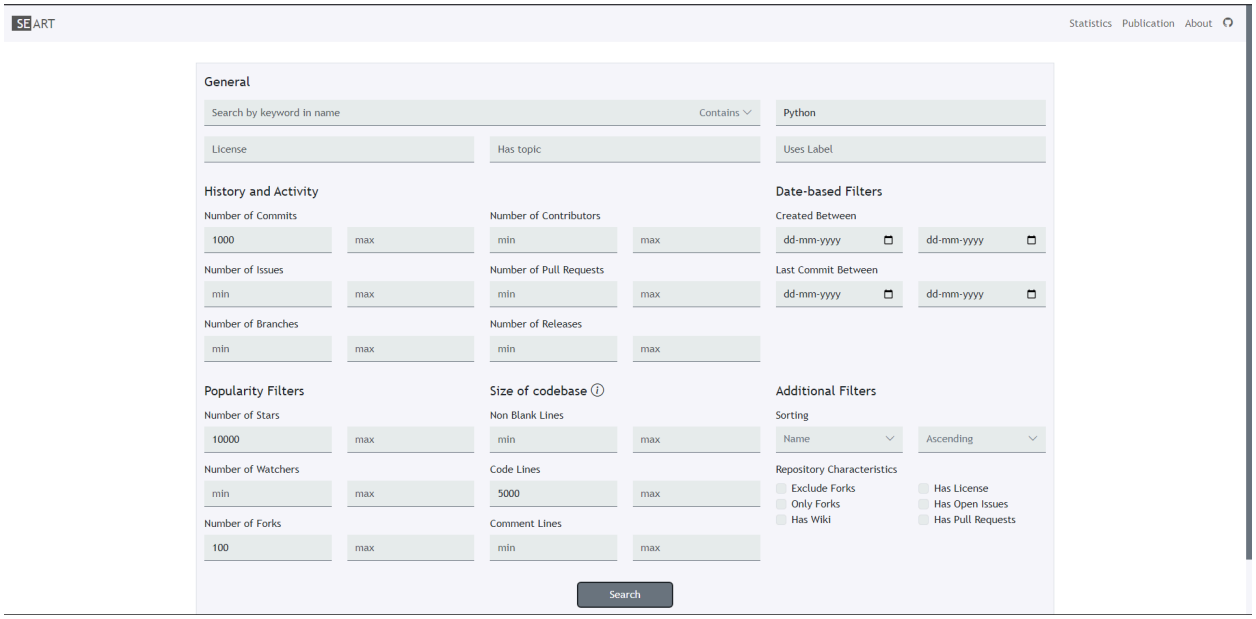
Three large-scale Python repositories were selected based on the following criteria:

Popularity Metrics: Minimum of 10,000 stars and 100 forks on GitHub.

Codebase Size: At least 5,000 lines of code.

Active Development: Minimum number of 1000 commits.

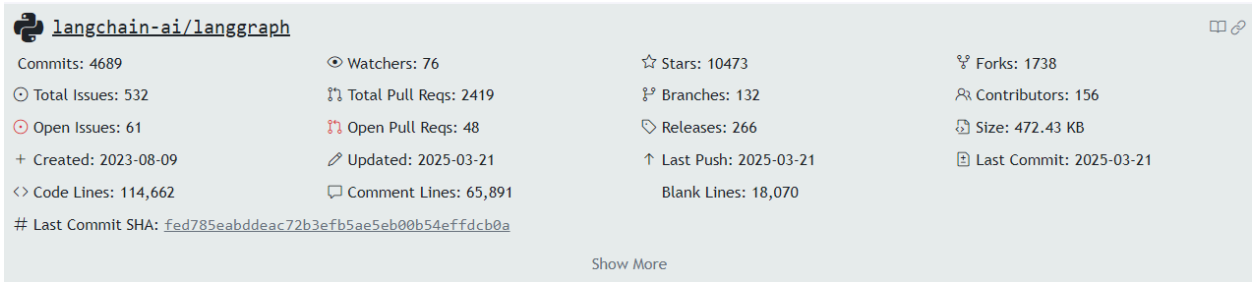
Development in Python language: All the repositories have Python as the primary programming language.



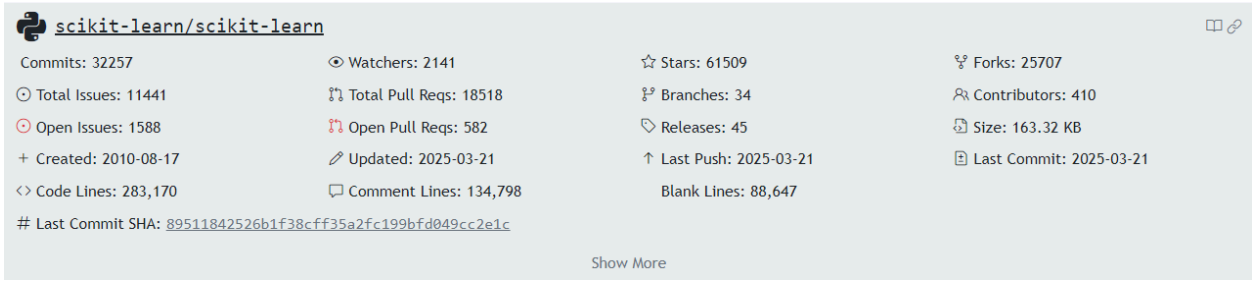
Repositories chosen:

- Flask

- Langchain

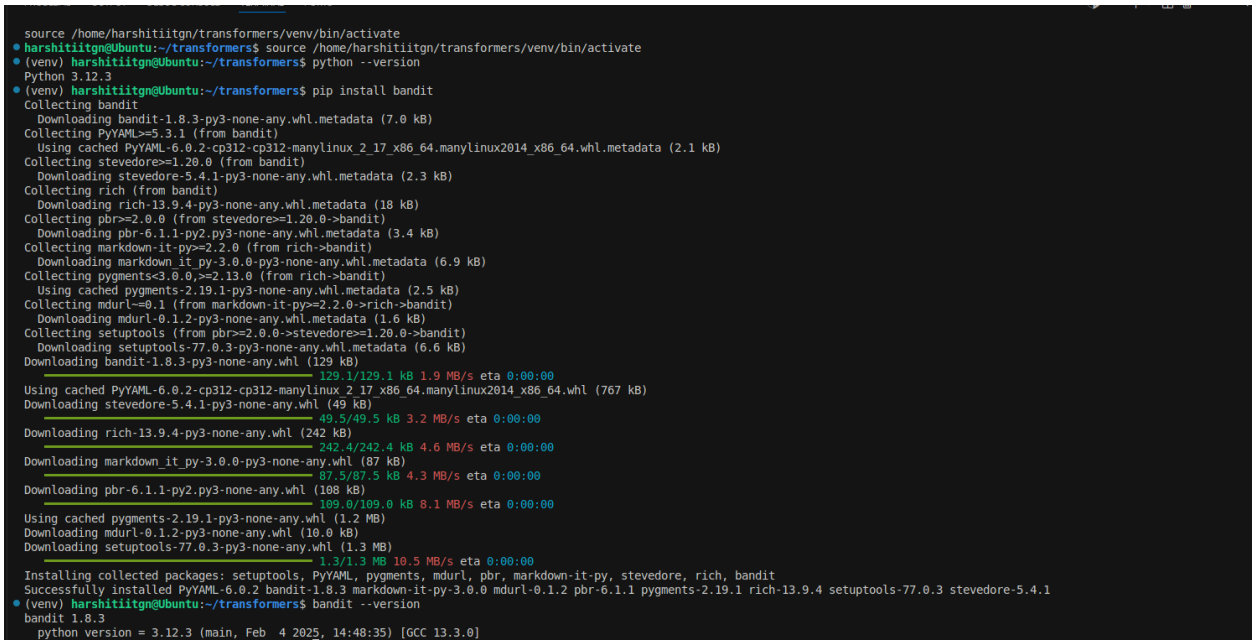


- Scikit-learn

- 

Cloning the repositories :

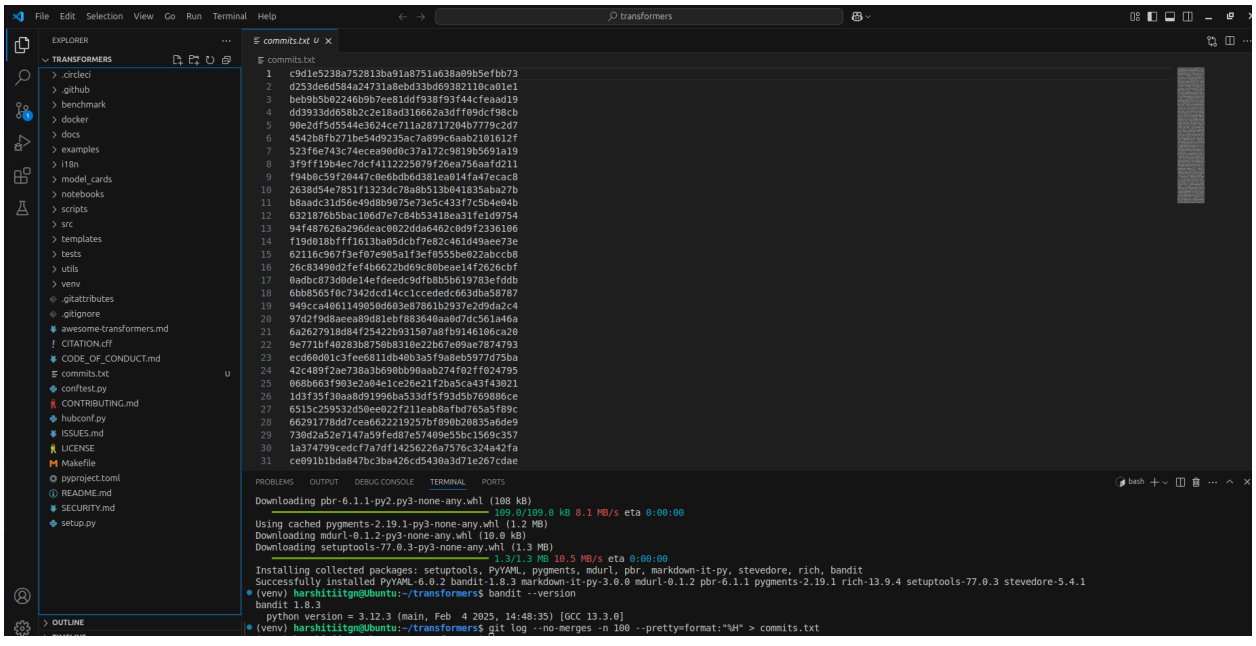
git clone https://github.com/langchain-ai/langchain
git clone https://github.com/pallets/flask
git clone https://github.com/scikit-learn/scikit-learn



```
harshitiitgn@Ubuntu: ~/transformers
harshitiitgn@Ubuntu:~$ git clone https://github.com/langchain-ai/langgraph
Cloning into 'langgraph'...
remote: Enumerating objects: 39582, done.
remote: Counting objects: 100% (1225/1225), done.
remote: Compressing objects: 100% (294/294), done.
^Cfetch-pack: unexpected disconnect while reading sideband packet

harshitiitgn@Ubuntu:~$ git clone https://github.com/huggingface/transformers
Cloning into 'transformers'...
remote: Enumerating objects: 264005, done.
remote: Total 264005 (delta 0), reused 0 (delta 0), pack-reused 264005 (from 1)
Receiving objects: 100% (264005/264005), 273.84 MiB | 3.98 MiB/s, done.
Resolving deltas: 100% (195572/195572), done.
Updating files: 100% (4832/4832), done.
harshitiitgn@Ubuntu:~$ cd transformers
```

```
harshitiitgn@Ubuntu:~/transformers$ python -m venv venv
Command 'python' not found, did you mean:
  command 'python3' from deb python3
  command 'python' from deb python-is-python3
harshitiitgn@Ubuntu:~/transformers$ python3 -m venv venv
harshitiitgn@Ubuntu:~/transformers$ source venv/bin/activate
```

If you don’t have python and instead have python3 then use python3 as done in the next step.



A. For Flask Repository :

Setting up the virtual environment for the lab :

Clone the repository using the following :

git clone https://github.com/pallets/flask

After cloning the repository, change the directory using the following :

cd flask
python -m venv flask-venv
flask-env\Scripts\activate # To Activate the environment in Windows
source flask-env/bin/activate # To Activate the environment in Linux

```
PS C:\Users\HP> git clone https://github.com/pallets/flask
Cloning into 'flask'...
remote: Enumerating objects: 25360, done.
remote: Total 25360 (delta 0), reused 0 (delta 0), pack-reused 25360 (from 1)
Receiving objects: 100% (25360/25360), 10.38 MiB | 3.49 MiB/s, done.
Resolving deltas: 100% (16988/16988), done.
PS C:\Users\HP> cd flask
PS C:\Users\HP\flask> python -m flask-venv venv
C:\Users\HP\AppData\Local\Programs\Python\Python312\python.exe: No module named flask-venv
PS C:\Users\HP\flask> python -m venv flask-venv
PS C:\Users\HP\flask> flask-venv/Scripts/activate
(flask-venv) PS C:\Users\HP\flask>
```

Installing the dependencies and the necessary tools :

pip install -r requirements.txt

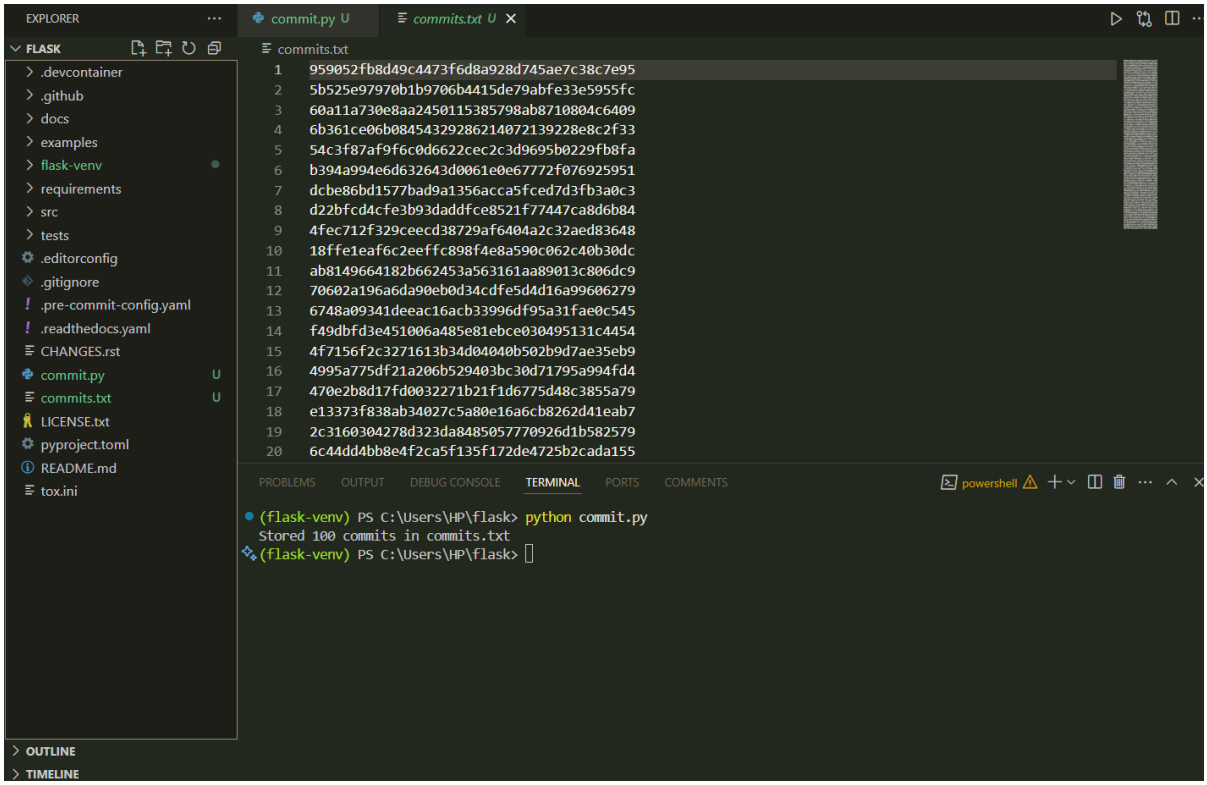
Install bandit using the following command :

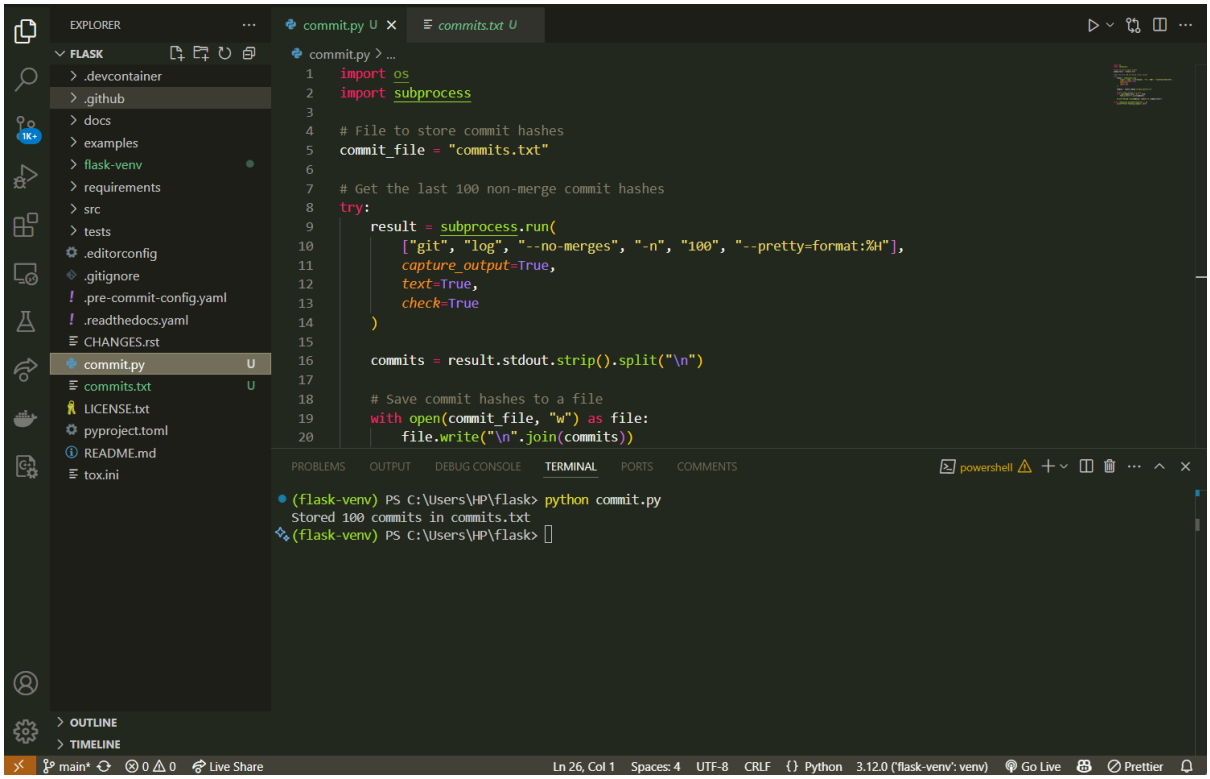
pip install bandit

To install bandit inside the virtual environment run the above command after activating the environment or else for system wide installation run it outside the virtual environment.

```
(flask-venv) PS C:\Users\HP\flask> pip install bandit
Collecting bandit
  Obtaining dependency information for bandit from https://files.pythonhosted.org/packages/88/85/db74b9233e0aa27ec96891045c5e920a64dd5cbccd50f8e64e9460f48d35/bandit-1.8.3-py3-none-any.whl.metadata
  Using cached bandit-1.8.3-py3-none-any.whl.metadata (7.0 kB)
Collecting PyYAML>=5.3.1 (from bandit)
  Obtaining dependency information for PyYAML>=5.3.1 from https://files.pythonhosted.org/packages/0c/e8/4f648c598b17c3d06e8753d7d13d57542b30d56e6c2dedf9c331ae56312e/PyYAML-6.0.2-cp312-cp312-win_amd64.whl.metadata
  Using cached PyYAML-6.0.2-cp312-cp312-win_amd64.whl.metadata (2.1 kB)
Collecting stevedore>=1.20.0 (from bandit)
  Obtaining dependency information for stevedore>=1.20.0 from https://files.pythonhosted.org/packages/f7/45/8c4ebc0c460e6ec38e62ab245ad3c7fc10b210116cea7c16d61602aa9558/stevedore-5.4.1-py3-none-any.whl.metadata
  Using cached stevedore-5.4.1-py3-none-any.whl.metadata (2.3 kB)
Collecting rich (from bandit)
  Obtaining dependency information for rich from https://files.pythonhosted.org/packages/19/71/39c7c0d87f8d4e6c020a393182060
```

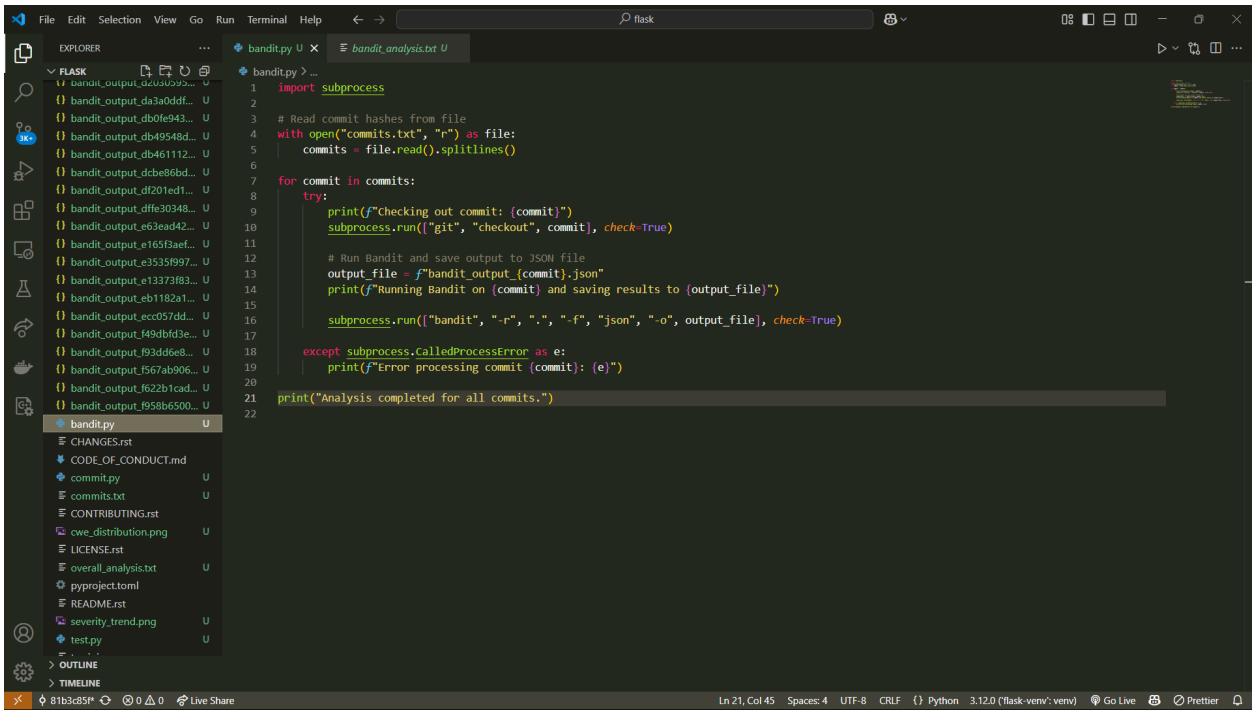
First run the commit.py file, to store the top 100 non-merges commit in a file names commit.txt for further analysis, using the following command :



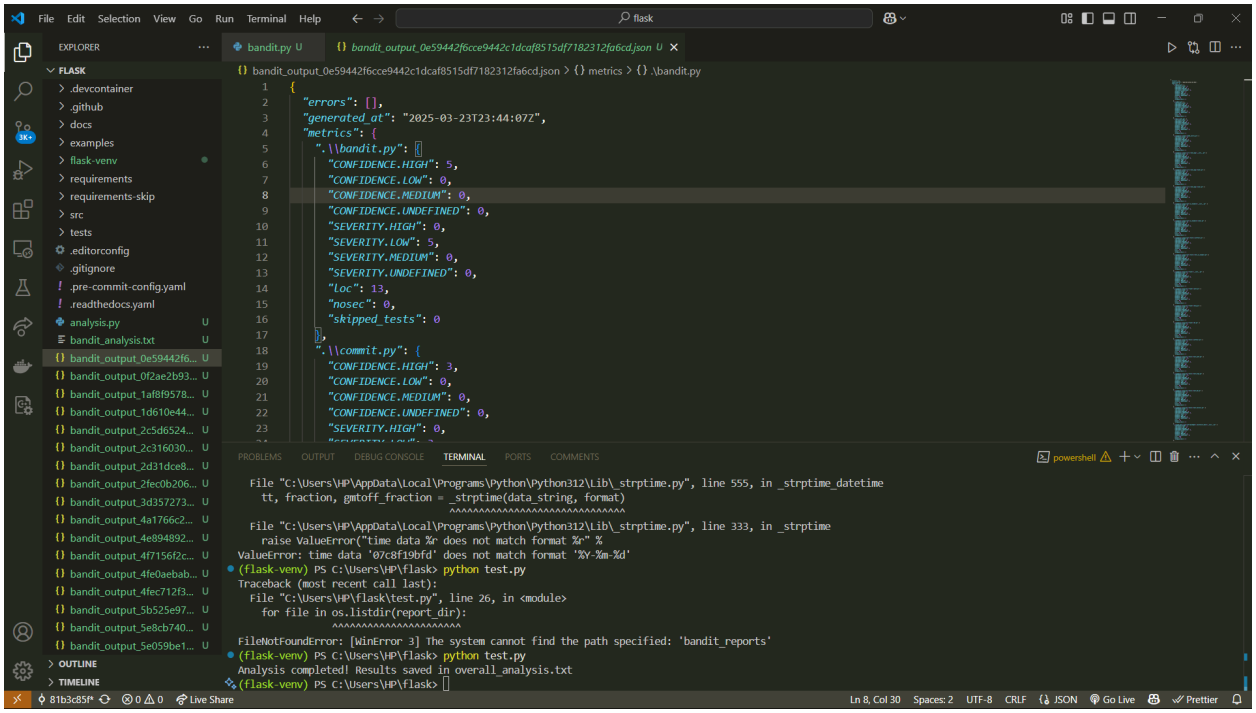


Now run the bandit analysis on the repository using the following command on the bash/terminal :

python bandit.py



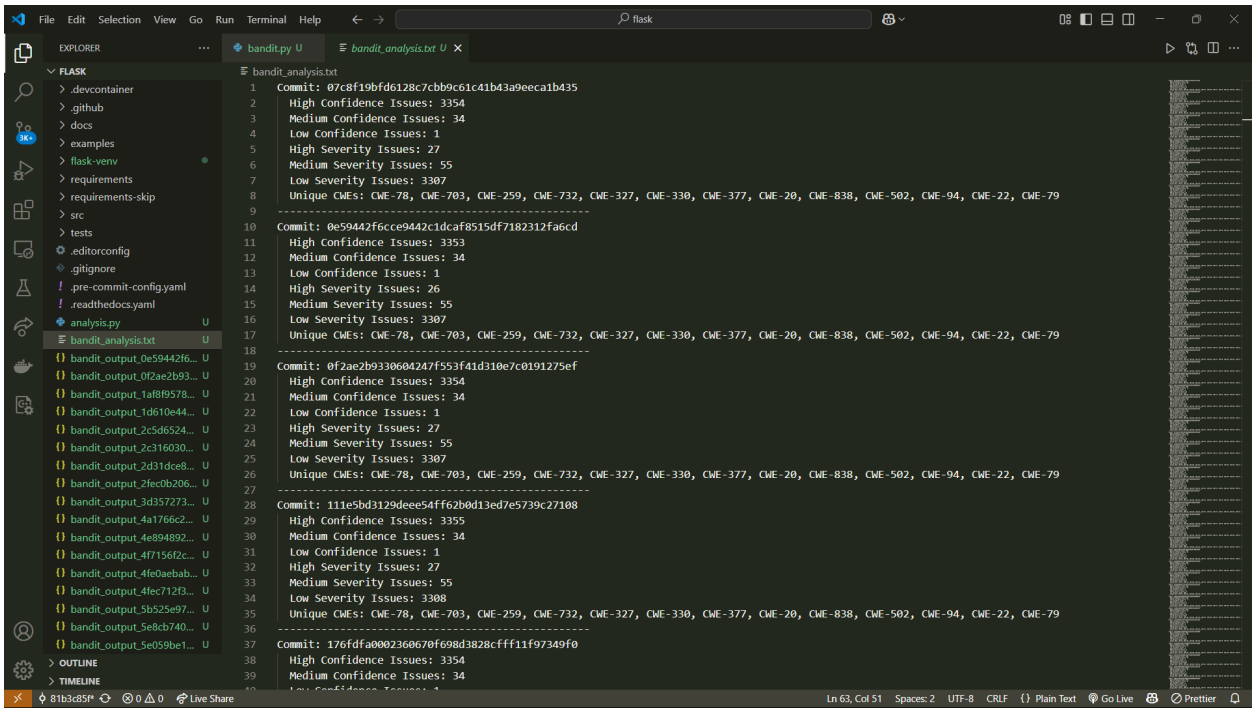
This is how the outputs are stored for the commits.



Now using these json files we have identified the following :

(a) Report the number of HIGH, MEDIUM and LOW confidence issues the tool identifies per commit.
(b) Report the number of HIGH, MEDIUM and LOW severity issues the tool identifies per commit.
(c) Report the unique CWEs the tool identifies per commit.

All of the above three things have been reported in a file named **bandit_analysis.txt**



B. For Scikit-learn :

Setting up the virtual environment for the lab :

Clone the repository using the following :

git clone https://github.com/scikit-learn/scikit-learn

After cloning the repository, change the directory using the following :

cd scikit-learn
python -m venv scikit-venv
scikit-venv\Scripts\activate # To Activate the environment in Windows
source scikit-env/bin/activate # To Activate the environment in Linux

Installing the dependencies and the necessary tools :

pip install -r requirements.txt

Install bandit using the following command :

pip install bandit

To install bandit inside the virtual environment run the above command after activating the environment or else for system wide installation run it outside the virtual environment.

```
PS C:\Users\HP> git clone https://github.com/scikit-learn/scikit-learn
Cloning into 'scikit-learn'...
remote: Enumerating objects: 256997, done.
remote: Counting objects: 100% (498/498), done.
remote: Compressing objects: 100% (182/182), done.
remote: Total 256997 (delta 348), reused 316 (delta 316), pack-reused 256499 (from 2)
Receiving objects: 100% (256997/256997), 163.52 MiB | 2.66 MiB/s, done.
Resolving deltas: 100% (199745/199745), done.
Updating files: 100% (1681/1681), done.
PS C:\Users\HP> cd scikit-learn
PS C:\Users\HP\scikit-learn> python -m venv scikit-venv
PS C:\Users\HP\scikit-learn> scikit-venv\Scripts\activate
(scikit-venv) PS C:\Users\HP\scikit-learn>
```

```
PS C:\Users\HP\scikit-learn> scikit-venv\Scripts\activate
(scikit-venv) PS C:\Users\HP\scikit-learn> pip install bandit
Collecting bandit
  Obtaining dependency information for bandit from https://files.pythonhosted.org/packages/88/85/db74b9233e0aa27ec96891045c5e
920a64dd5cbccd50f8e64e9460f48d35/bandit-1.8.3-py3-none-any.whl.metadata
  Using cached bandit-1.8.3-py3-none-any.whl.metadata (7.0 kB)
Collecting PyYAML>=5.3.1 (from bandit)
  Obtaining dependency information for PyYAML>=5.3.1 from https://files.pythonhosted.org/packages/0c/e8/4f648c598b17c3d06e875
3d7d13d57542b30d56e6c2dedf9c331ae56312e/PyYAML-6.0.2-cp312-cp312-win_amd64.whl.metadata
  Using cached PyYAML-6.0.2-cp312-cp312-win_amd64.whl.metadata (2.1 kB)
Collecting stevedore>=1.20.0 (from bandit)
  Obtaining dependency information for stevedore>=1.20.0 from https://files.pythonhosted.org/packages/f7/45/8c4ebc0c460e6ec38
e62ab245ad3c7fc10b210116cea7c16d61602aa9558/stevedore-5.4.1-py3-none-any.whl.metadata
  Using cached stevedore-5.4.1-py3-none-any.whl.metadata (2.3 kB)
Collecting rich (from bandit)
```

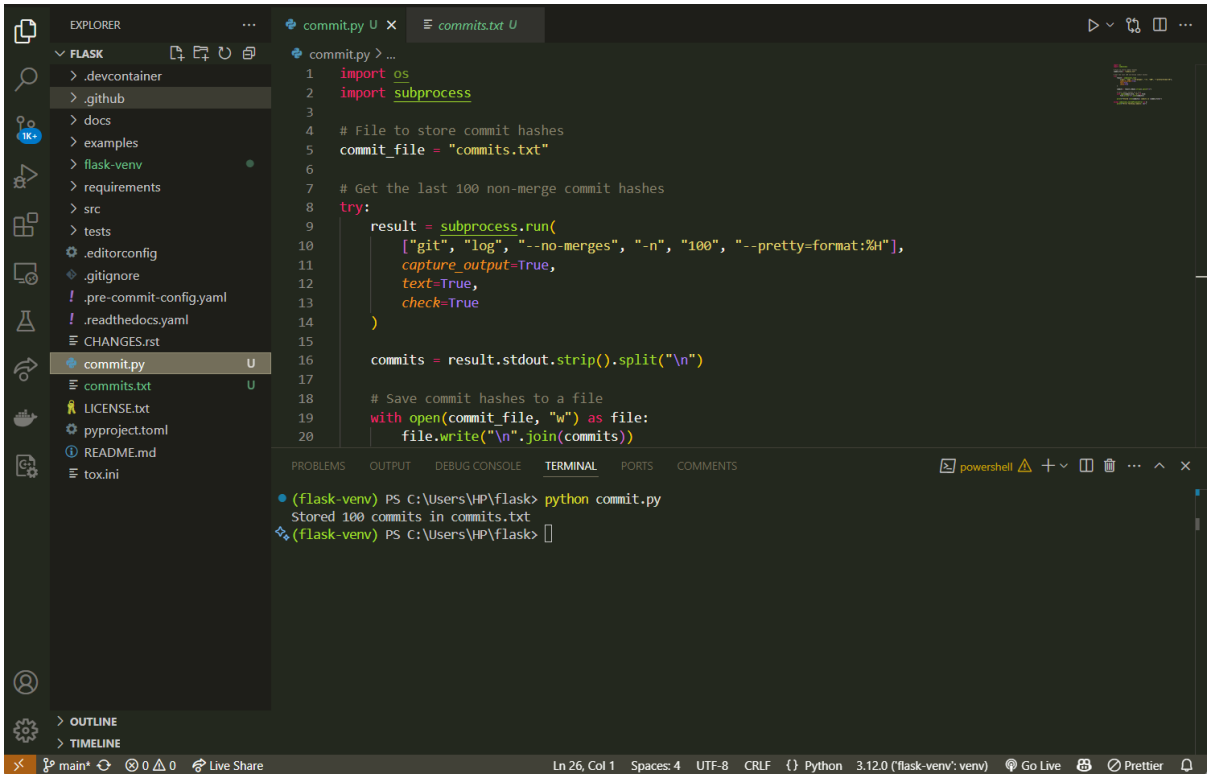
First run the commit.py file, to store the top 100 non-merges commit in a file names commit.txt for further analysis, using the following command :

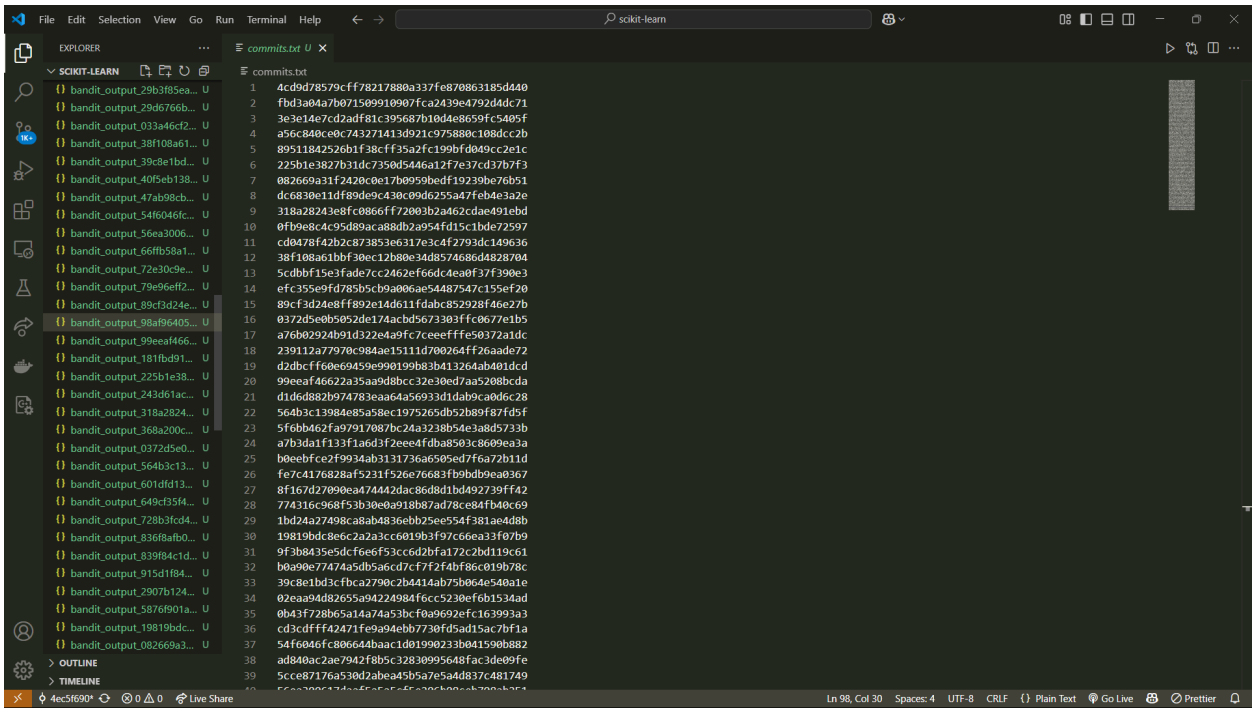
python commit.txt

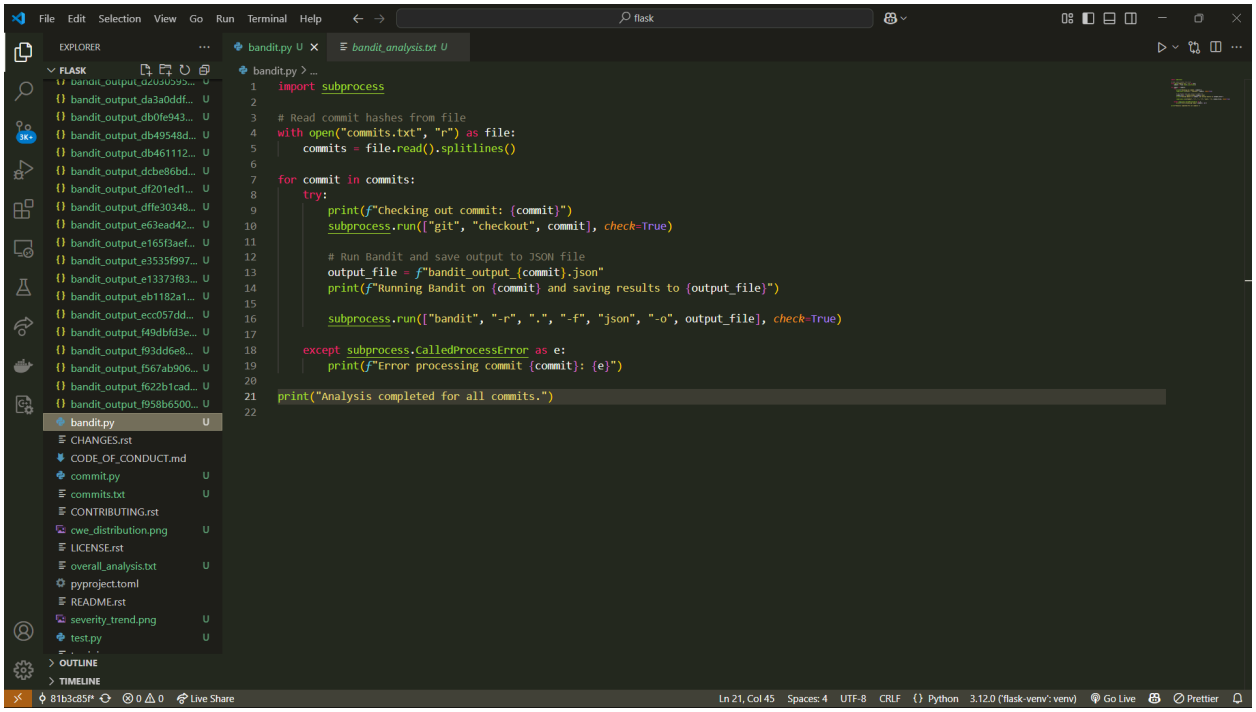
The same code commit.py has been used here as well to extract the top 100 non-merge commits for this repository.

Now run the bandit analysis on the repository using the following command on the bash/terminal :

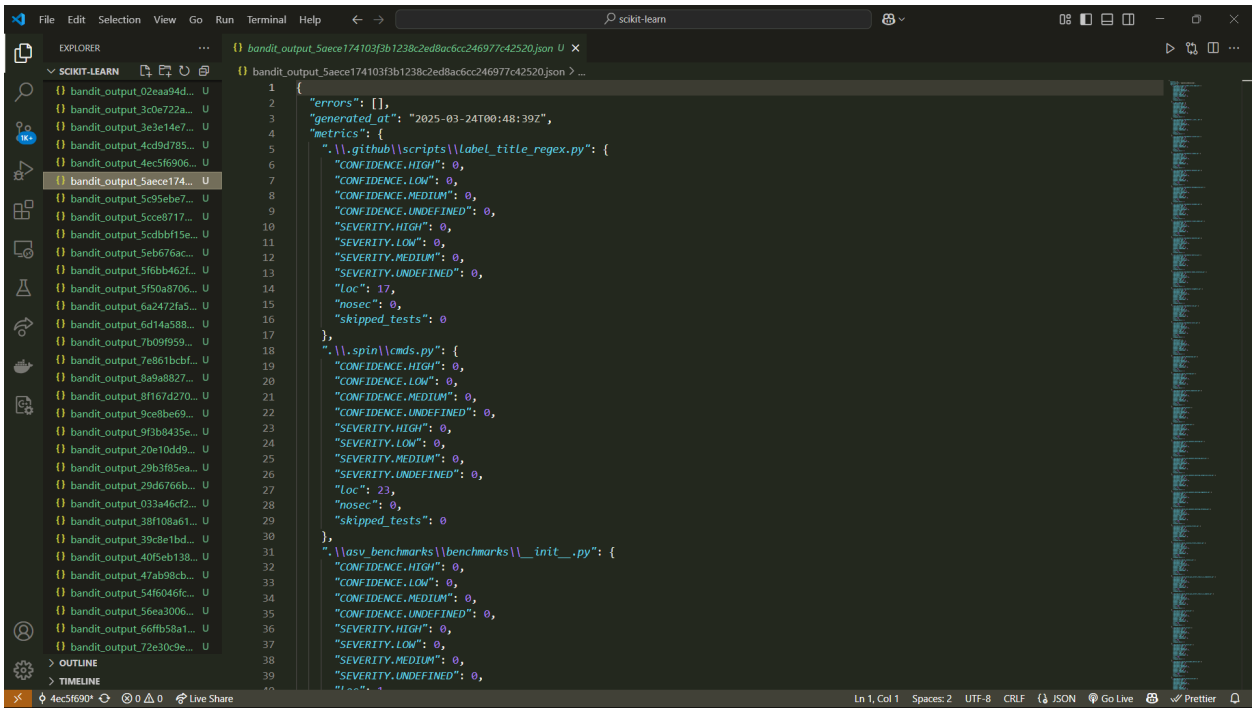
python bandit.py


EXPLORER
FLASK
.devcontainer
.github
docs
examples
flask-venv
requirements
src
tests
.editorconfig
.gitignore
.pre-commit-config.yaml
.readthedocs.yaml
CHANGES.rst
commit.py U
commits.txt U
LICENSE.txt
pyproject.toml
README.md
tox.ini
commit.py U
commits.txt U
commit.py > ...
import os
import subprocess
# File to store commit hashes
commit_file = "commits.txt"
# Get the last 100 non-merge commit hashes
try:
result = subprocess.run(
["git", "log", "--no-merges", "-n", "100", "--pretty=format:%H"],
capture_output=True,
text=True,
check=True
)
commits = result.stdout.strip().split("\n")
# Save commit hashes to a file
with open(commit_file, "w") as file:
file.write("\n".join(commits))
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
COMMENTS
powershell
(flask-venv) PS C:\Users\HP\flask> python commit.py
Stored 100 commits in commits.txt
(flask-venv) PS C:\Users\HP\flask>
OUTLINE
TIMELINE
main*
Live Share
Ln 26, Col 1
Spaces: 4
UTF-8
CRLF
Python
3.12.0 ('flask-venv': venv)
Go Live
Prettier


File Edit Selection View Go Run Terminal Help
scikit-learn
EXPLORER
SCIKIT-LEARN
commits.txt U
commits.txt
4cd9d78579cff78217880a337fe870863185d440
fbd3a04a7b07150910907fca2439e4792d4dc71
3e3e14e7cd2adf81c395687b10d4e8659fc5405f
a56c840ce0c743271413d921c975880c108dcc2b
89511842526b1f38cff35a2fc199bfd049cc2e1c
225b1e3827b31dc7350d5446a12f7e37cd37b7f3
082669a31f2420c0e17b0959bedf19239be76b51
dc6830e11df89de9c430c09d6255a47feb4e3a2e
318a28243e8fc0866ff72003b2a462cdae491ebd
0fb9e8c4c95d89aca88db2a954fd15c1bde72597
cd0478f42b2c873853e6317e3c4f2793dc149636
38f108a61bbf30ec12b80e34d8574686d4828704
5cdbbf15e3fade7cc2462ef66dc4ea0f37f390e3
efc355e9fd785b5cb9a006ae54487547c155ef20
89cf3d24e8ff892e14d611fdabc852928f46e27b
0372d5e0b5052de174acbd5673303ffc0677e1b5
a76b02924b91d322e4a9fc7ceeefffe50372a1dc
239112a77970c984ae15111d700264ff26aade72
d2dbcff60e69459e990199b83b413264ab401dcd
99eeaf46622a35aa9d8bcc32e30ed7aa5208bcda
d1d6d882b974783eaa64a56933d1dab9ca0d6c28
564b3c13984e85a58ec1975265db52b89f87fd5f
5f6bb462fa97917087bc24a3238b54e3a8d5733b
a7b3da1f133f1a6d3f2eee4fdba8503c8609ea3a
b0eebfce2f9934ab3131736a6505ed7f6a72b11d
fe7c4176828af5231f526e76683fb9bdb9ea0367
8f167d27090ea474442dac86d8d1bd492739ff42
774316c968f53b30e0a918b87ad78ce84fb40c69
1bd24a27498ca8ab4836ebb25ee554f381ae4d8b
19819bdc8e6c2a2a3cc6019b3f97c66ea33f07b9
9f3b8435e5dcf6e6f53cc6d2bfa172c2bd119c61
b0a90e77474a5db5a6cd7cf7f2f4bf86c019b78c
39c8e1bd3cfbca2790c2b4414ab75b064e540a1e
02eaa94d82655a94224984f6cc5230ef6b1534ad
0b43f728b65a14a74a53bcf0a9692efc163993a3
cd3cdfff42471fe9a94ebb7730fd5ad15ac7bf1a
54f6046fc806644baac1d01990233b041590b882
ad840ac2ae7942f8b5c32830995648fac3de09fe
5cce87176a530d2abea45b5a7e5a4d837c481749
OUTLINE
TIMELINE
4ec5f690*
Live Share
Ln 98, Col 30
Spaces: 4
UTF-8
CRLF
Plain Text
Go Live
Prettier



This is how the outputs are stored for the commits :



Now using these json files we have identified the following :

(a) Report the number of HIGH, MEDIUM and LOW confidence issues the tool identifies per commit.
(b) Report the number of HIGH, MEDIUM and LOW severity issues the tool identifies per commit.
(c) Report the unique CWEs the tool identifies per commit.

All of the above three things have been reported in a file named **bandit_analysis.txt**

C. For Langchain :

Setting up the virtual environment for the lab :

Clone the repository using the following :

git clone https://github.com/langchain-ai/langchain

After cloning the repository, change the directory using the following :

```
cd langchain
python -m venv lang-venv
lang-venv\Scripts\activate  # To Activate the environment in Windows
source lang-env/bin/activate # To Activate the environment in Linux
```

Installing the dependencies and the necessary tools :

pip install -r requirements.txt

Install bandit using the following command :

pip install bandit

To install bandit inside the virtual environment run the above command after activating the environment or else for system wide installation run it outside the virtual environment.

First run the commit.py file, to store the top 100 non-merges commit in a file names commit.txt for further analysis, using the following command :

python commit.txt

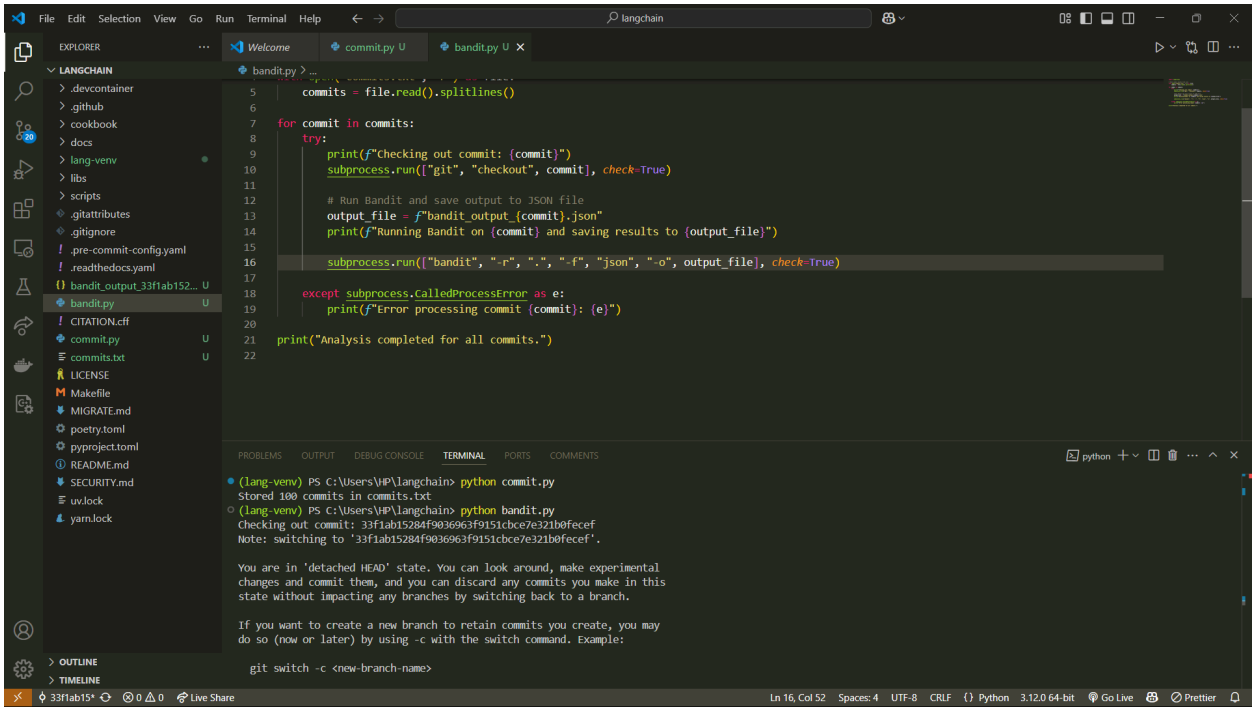
The same code commit.py has been used here as well to extract the top 100 non-merge commits for this repository.

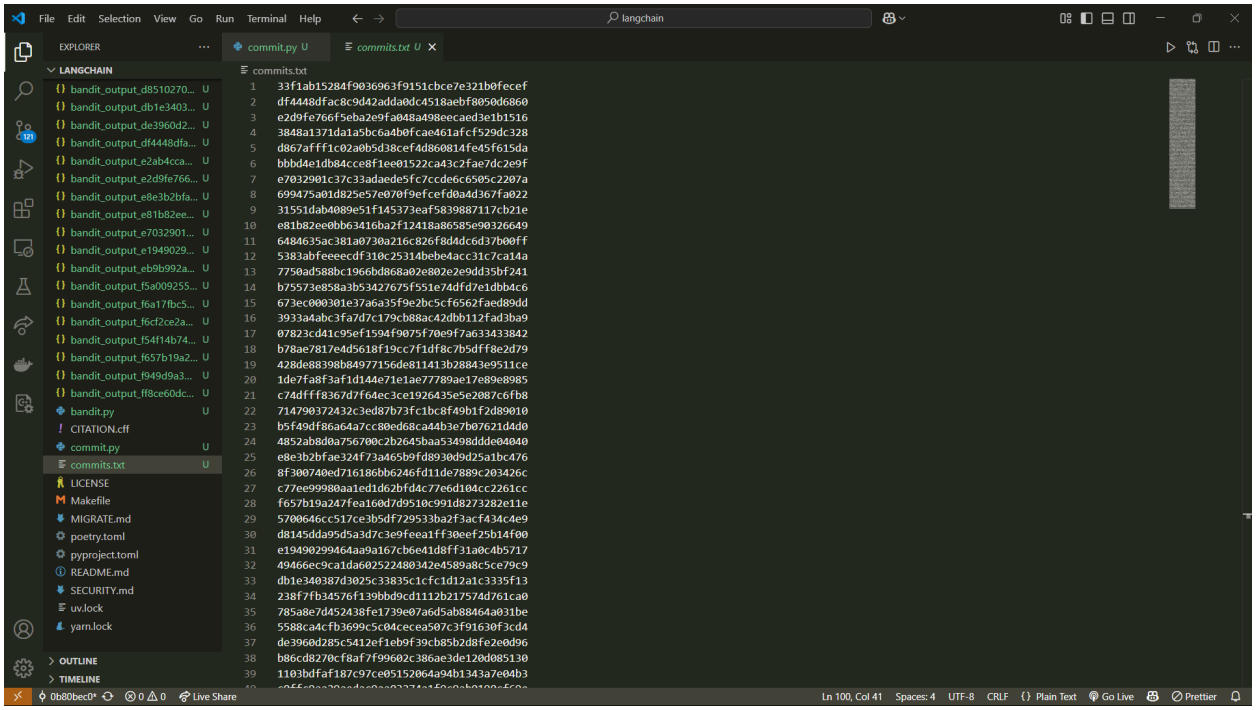
Now run the bandit analysis on the repository using the following command on the bash/terminal :

python bandit.py

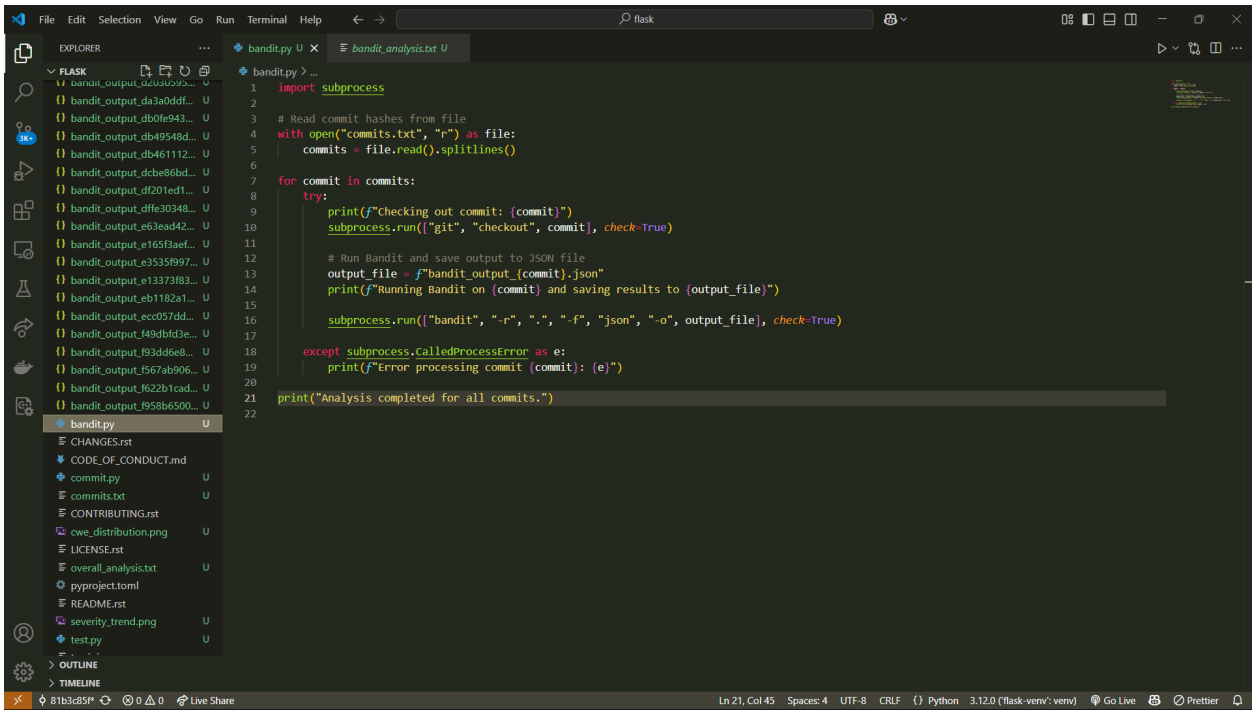
```
PS C:\Users\HP> git clone https://github.com/langchain-ai/langchain
Cloning into 'langchain'...
remote: Enumerating objects: 232119, done.
remote: Counting objects: 100% (455/455), done.
remote: Compressing objects: 100% (264/264), done.
remote: Total 232119 (delta 350), reused 194 (delta 191), pack-reused 231664 (from 4)
Receiving objects: 100% (232119/232119), 391.79 MiB | 4.51 MiB/s, done.
Resolving deltas: 100% (175231/175231), done.
Updating files: 100% (7361/7361), done.
PS C:\Users\HP> python -m venv langchain-venv
PS C:\Users\HP> langchain-venv\Scripts\activate
(langchain-venv) PS C:\Users\HP>
```

```
PS C:\Users\HP\langchain> lang-venv\Scripts\activate
(lang-venv) PS C:\Users\HP\langchain> pip install bandit
Collecting bandit
  Obtaining dependency information for bandit from https://files.pythonhosted.org/packages/88/85/db74b9233e0aa27ec96891045c5e920a64dd5cbccd50f8e64e9460f48d35/bandit-1.8.3-py3-none-any.whl.metadata
  Using cached bandit-1.8.3-py3-none-any.whl.metadata (7.0 kB)
Collecting PyYAML>=5.3.1 (from bandit)
  Obtaining dependency information for PyYAML>=5.3.1 from https://files.pythonhosted.org/packages/0c/e8/4f648c598b17c3d06e8753d7d13d57542b30d56e6c2dedf9c331ae56312e/PyYAML-6.0.2-cp312-cp312-win_amd64.whl.metadata
  Using cached PyYAML-6.0.2-cp312-cp312-win_amd64.whl.metadata (2.1 kB)
Collecting stevedore>=1.20.0 (from bandit)
  Obtaining dependency information for stevedore>=1.20.0 from https://files.pythonhosted.org/packages/f7/45/8c4ebc0c460e6ec38e62ab245ad3c7fc10b210116cea7c16d61602aa9558/stevedore-5.4.1-py3-none-any.whl.metadata
  Using cached stevedore-5.4.1-py3-none-any.whl.metadata (2.3 kB)
Collecting rich (from bandit)
```

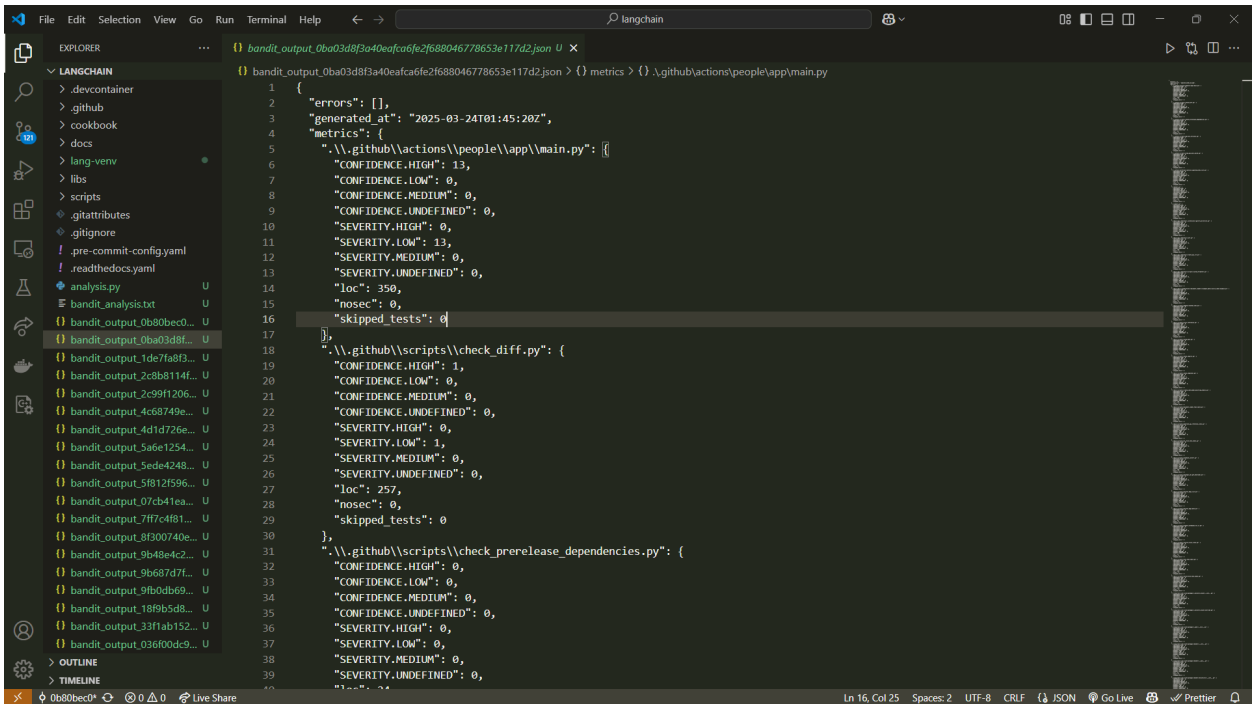




The same python script named bandit.py had been used in the analysis of all three repositories.



This is how the outputs are stored for the commits :



3. Results and Analysis :

Now a combined folder has been created namely combined_analysis. All of the bandits reports for all the three repositories were added to the folder and then these were analyzed using the script named analyze_bandit.py. The results were stored in overall_analysis.txt and the plots were also stored.

Now using a python script named test.script this txt file was converted to a csv file for easier analysis using pandas.

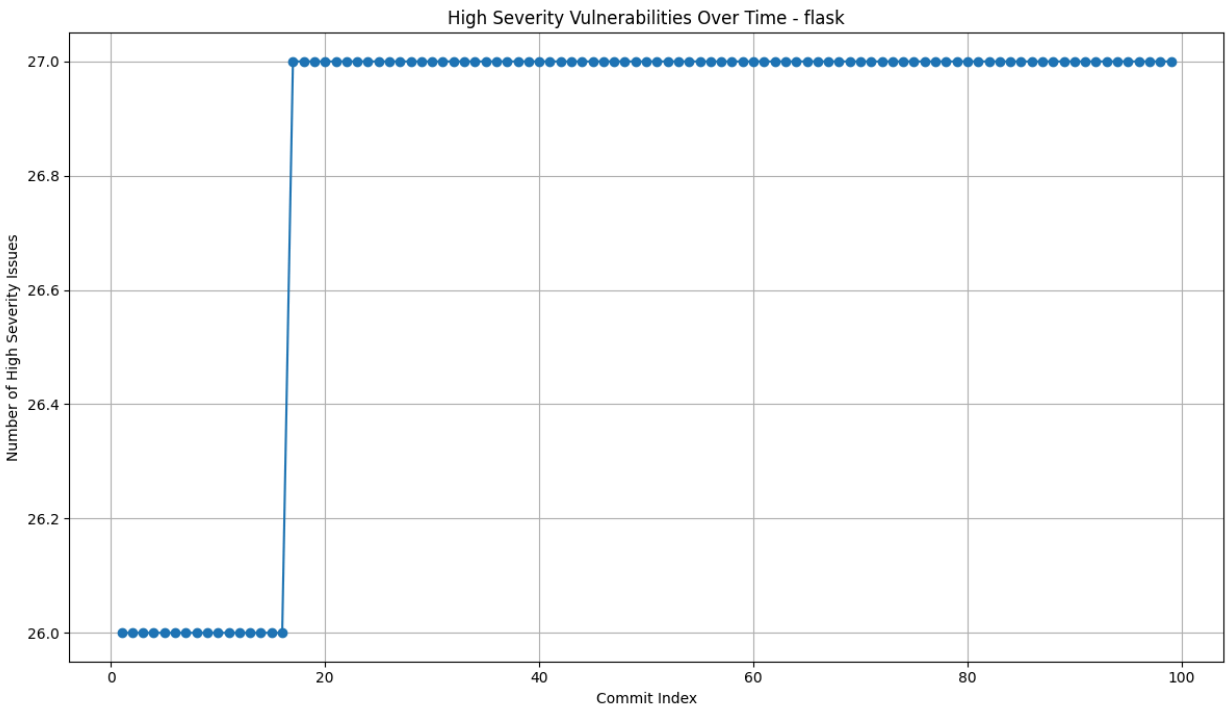
Then using another script named test2.py we fixed the commit_index values using the timestamps to get the commit index in the sorted order.

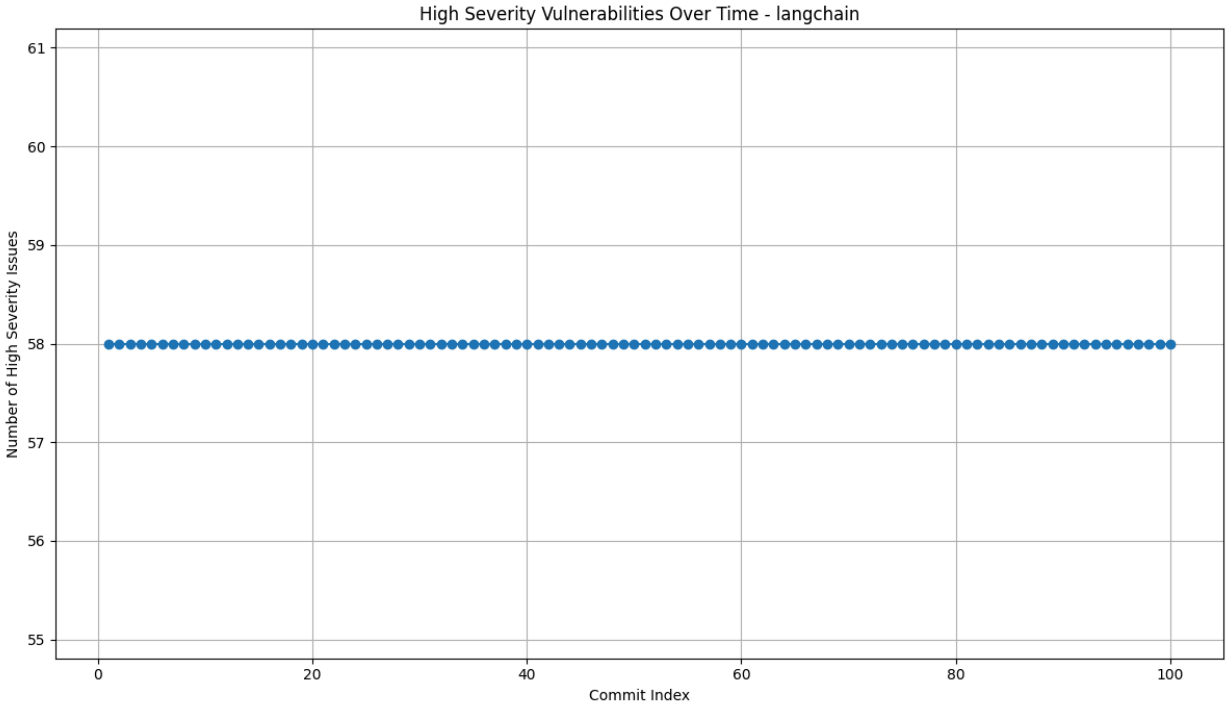
Then using the files named rq1.py, rq2.py and rq3.py we got the corresponding plots required for the analysis

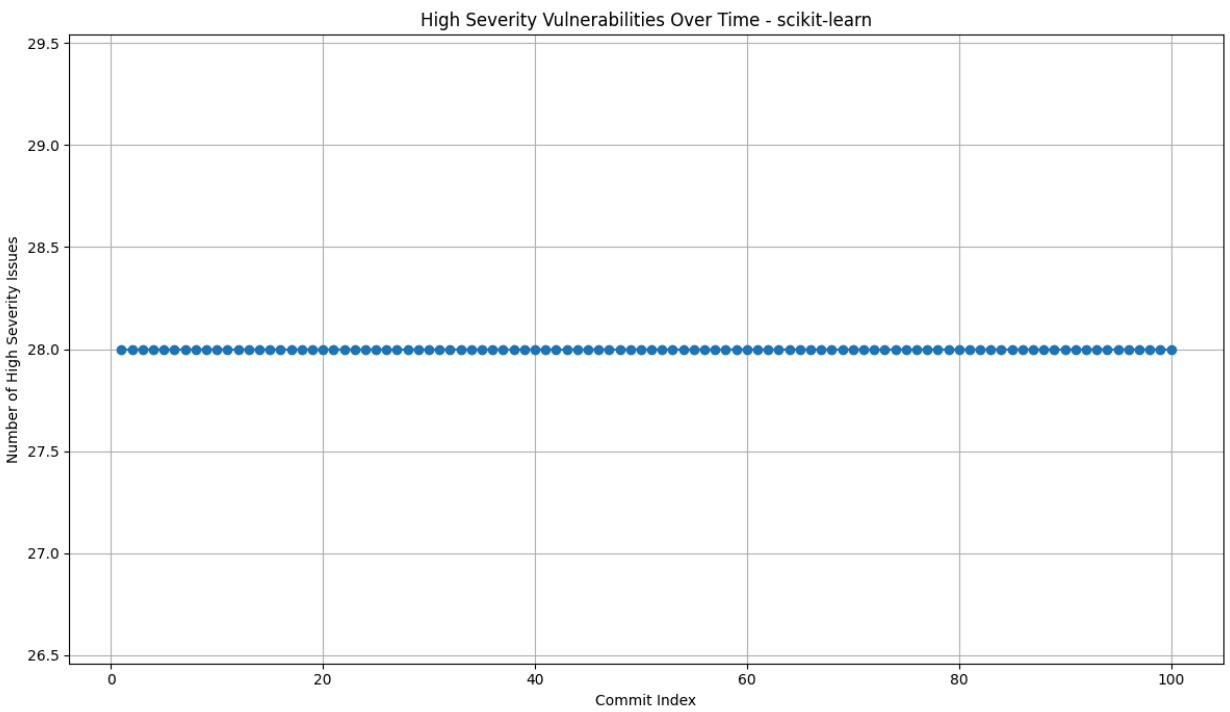
Answering RQ1: High-Severity Vulnerabilities Across LangChain, Flask, and Scikit-Learn

Purpose

The purpose of this research question is to evaluate the trend of high-severity vulnerabilities over different commits for LangChain, Flask, and Scikit-Learn. This helps determine if these repositories are improving in security, remaining stable, or becoming more vulnerable over time.


High Severity Vulnerabilities Over Time - flask
Number of High Severity Issues
Commit Index
27.0
26.8
26.6
26.4
26.2
26.0
0
20
40
60
80
100


High Severity Vulnerabilities Over Time - langchain
Number of High Severity Issues
Commit Index
61
60
59
58
57
56
55
0
20
40
60
80
100



Approach

Collected historical security scan data using Bandit for multiple commits in each repository.

Extracted high-severity issues for each commit.

Plotted the number of high-severity vulnerabilities across 100 commits.

Analyzed the trends to identify patterns of security improvement, degradation, or stability.

Results

LangChain

The graph for LangChain shows a completely flat trend at 58 high-severity vulnerabilities across all commits.

This indicates that no new high-severity issues were introduced, but also that no fixes were made.

The security posture remains unchanged, meaning there has been no active effort to mitigate existing vulnerabilities.

Flask

The graph for Flask shows a slight increasing trend in high-severity vulnerabilities over commits.

This suggests that new vulnerabilities are being introduced over time, likely due to feature additions or changes in dependencies.

Flask's security status is worsening, requiring more proactive security measures.

Scikit-Learn

The graph for Scikit-Learn remains stable, with only minor fluctuations in the number of high-severity vulnerabilities.

Unlike Flask, it does not show a continuous increase, but occasional variations suggest some vulnerabilities are fixed while new ones appear.

This indicates consistent security monitoring, but improvements could still be made to reduce overall vulnerability count.

Takeaway

LangChain's security is stagnant—it neither improves nor worsens.

Flask is becoming more vulnerable over time, requiring immediate security attention.

Scikit-Learn is relatively stable, but periodic security fixes are needed to avoid future risks.

Answering RQ2: Do vulnerabilities of different severity have the same pattern of introduction and elimination?

Purpose

This research question aims to analyze whether vulnerabilities of different severity levels (high, medium, low) exhibit similar trends in their introduction and elimination across different OSS repositories. Specifically, we investigate how the number of vulnerabilities fluctuates over time for LangChain, Flask, and Scikit-Learn.

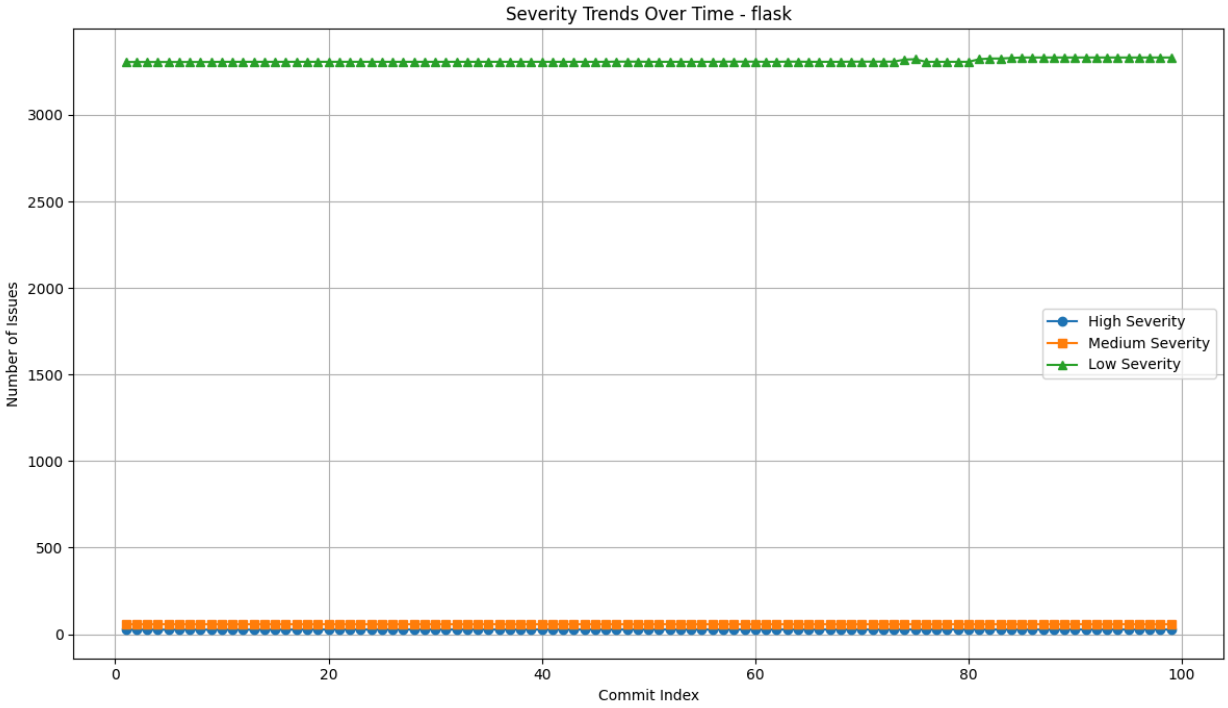
Approach

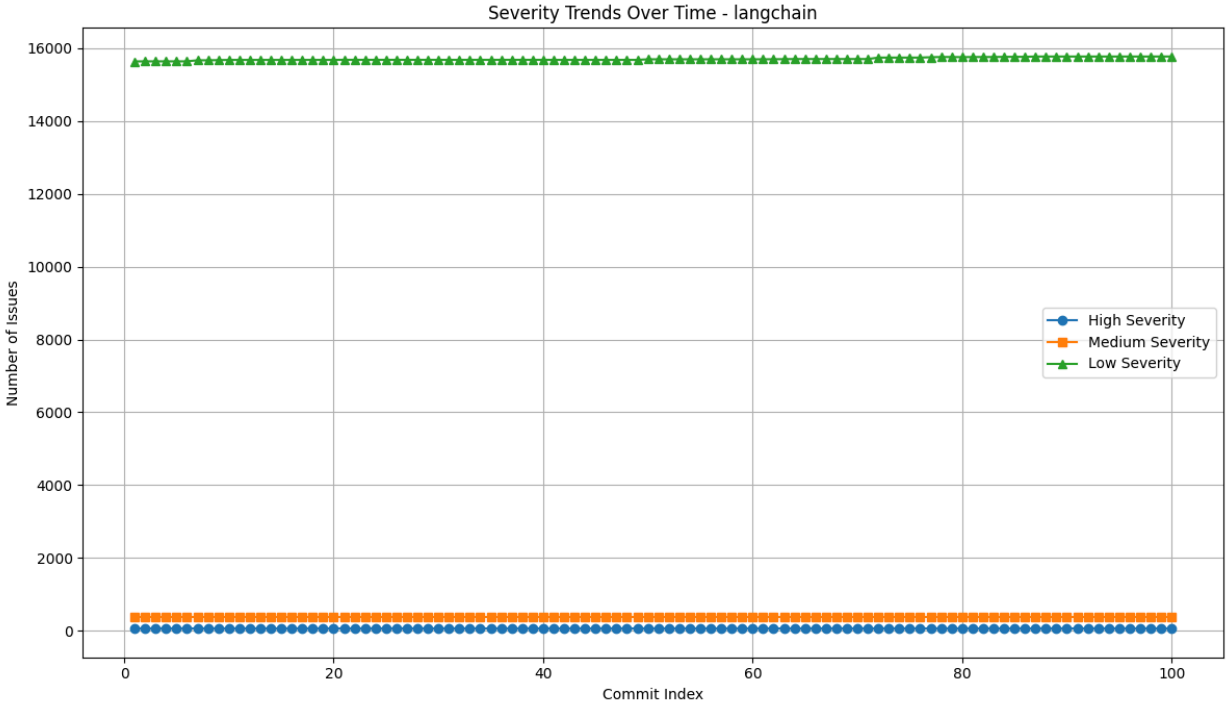
We analyzed the Bandit results across different commits for LangChain, Flask, and Scikit-Learn.

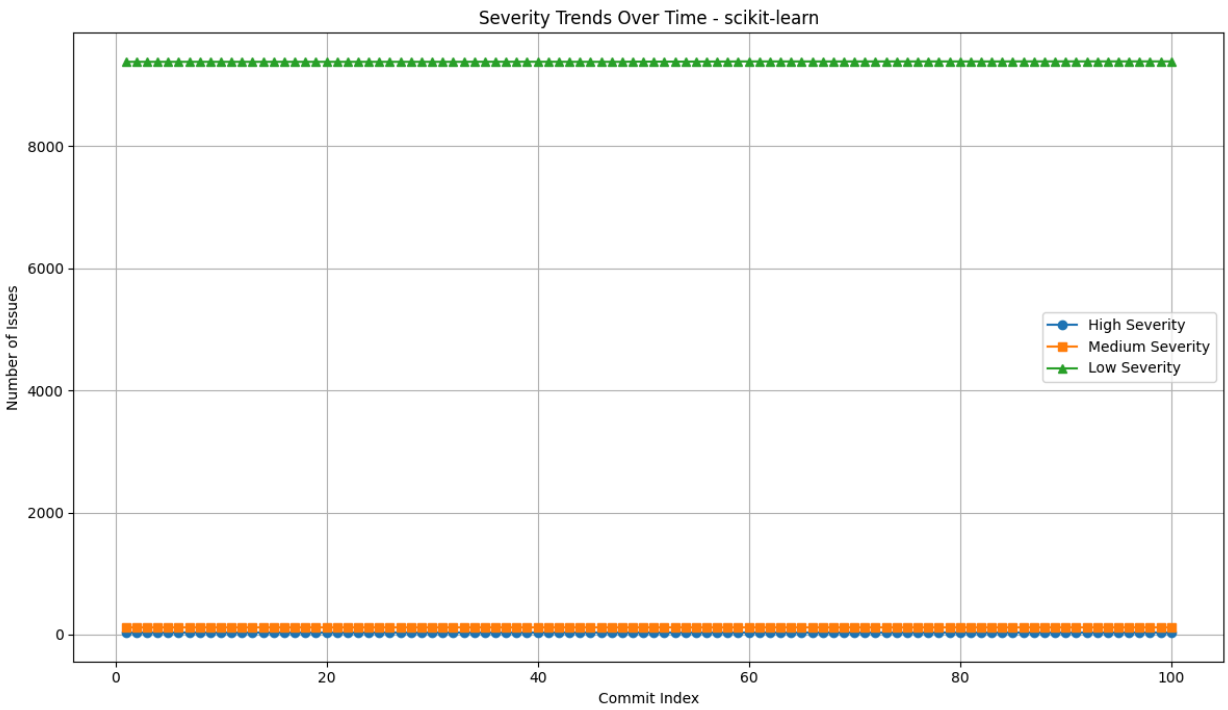
We categorized vulnerabilities into three severity levels: High, Medium, and Low.

We visualized the trend of vulnerabilities across commits using line plots to observe their behavior over time.

We examined whether all severity levels follow a similar pattern of introduction and elimination.







Results

LangChain

Low-severity vulnerabilities remain consistently high (around 16,000 issues).

Medium-severity vulnerabilities remain steady at a lower count.

High-severity vulnerabilities are also present but relatively lower than medium and low.

The overall pattern remains stable, indicating that issues persist across commits without significant elimination.

Flask

Low-severity vulnerabilities are around 3,200 and remain nearly constant.

Medium-severity vulnerabilities fluctuate slightly but stay relatively low.

High-severity vulnerabilities also show minimal variation.

The overall trend suggests that vulnerabilities are not actively being eliminated but remain steady across commits.

Scikit-Learn

Low-severity vulnerabilities remain at 9,000+, showing minimal fluctuation.

Medium-severity vulnerabilities remain relatively stable at a much lower count.

High-severity vulnerabilities show some fluctuation but stay relatively low.

The trend here is similar to LangChain and Flask, indicating a lack of active vulnerability mitigation over commits.

Takeaway

Across all three repositories, low-severity vulnerabilities dominate, followed by medium and high severity. The vulnerabilities remain relatively constant over time, with little indication of active fixing. This suggests that security debt accumulates, and there is no systematic approach to eliminating vulnerabilities as new code is introduced.

Answering RQ3 (CWE coverage)

Purpose:

The goal of this research question is to analyze and compare the most frequently occurring Common Weakness Enumerations (CWEs) across different open-source software (OSS) repositories—specifically Flask, LangChain, and Scikit-learn. This helps identify prevalent security vulnerabilities and provides insights into security risks in different software ecosystems.

Approach:

Data Extraction:

Extracted security vulnerability reports from Bandit scans for three OSS repositories: Flask, LangChain, and Scikit-learn.

Filtered the results to focus only on CWE mappings.

Data Processing:

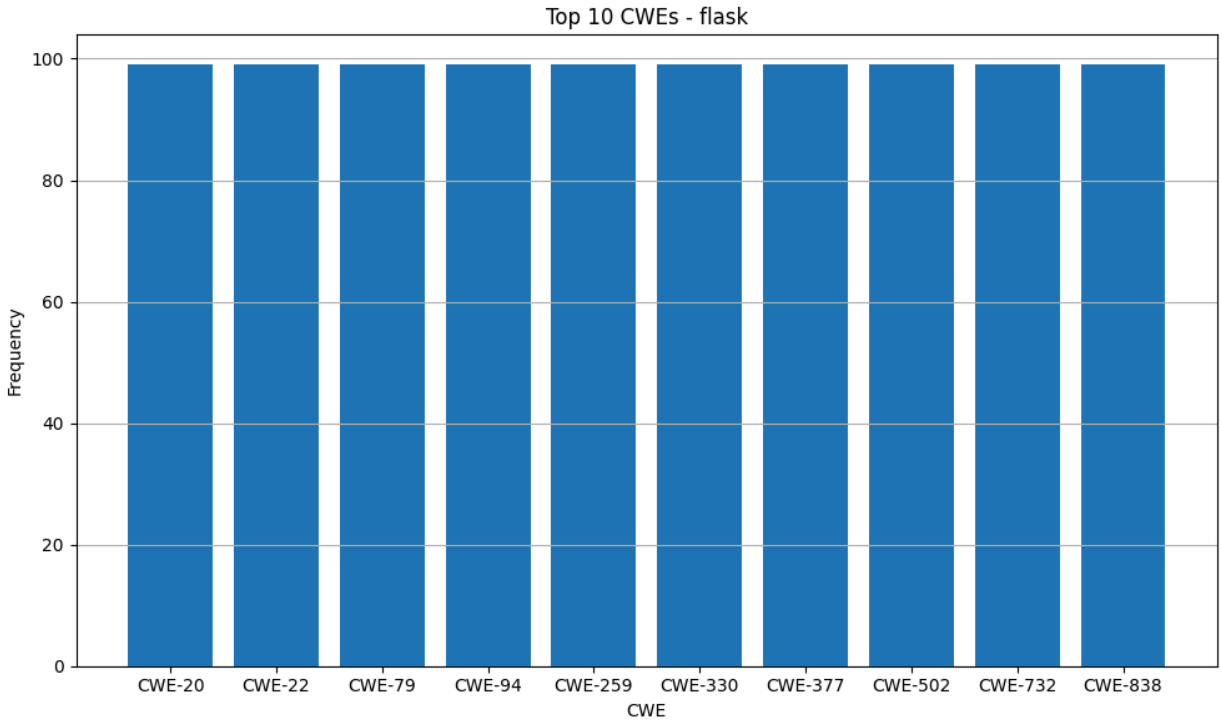
Aggregated CWE occurrences across repositories.

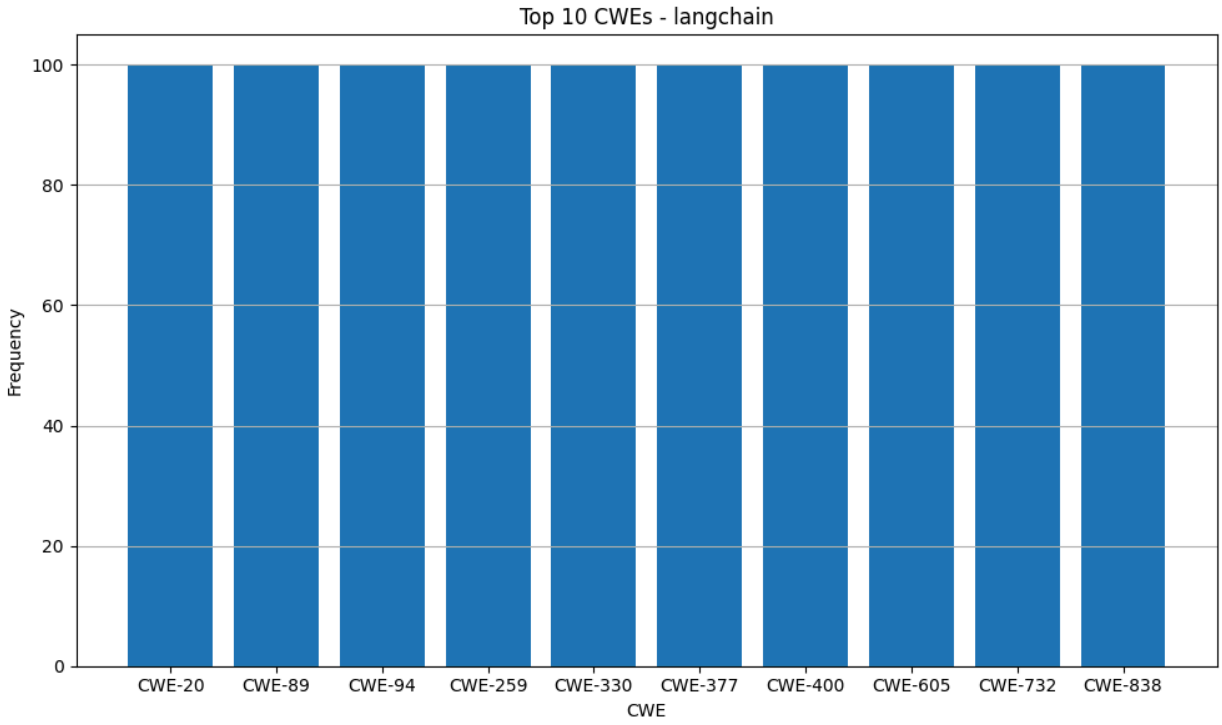
Identified the top 10 most frequent CWEs for each repository.

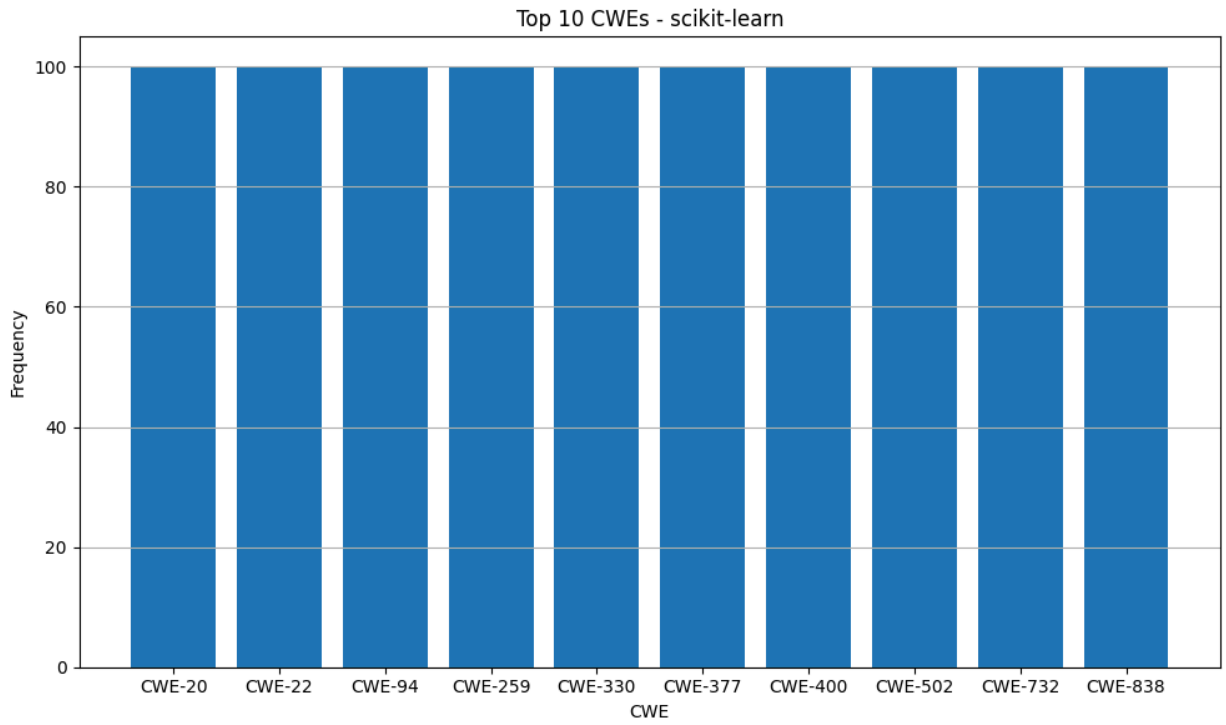
Compared CWE distributions among the three repositories.
Visualization:

Used bar charts to illustrate the frequency of different CWEs across the repositories.

Compared security trends to highlight repository-specific vulnerabilities.







Results:

1. Flask (Web Framework)

Top 10 CWEs Identified:

CWE-20 (Improper Input Validation)

CWE-22 (Path Traversal)

CWE-79 (Cross-Site Scripting)

CWE-94 (Code Injection)

CWE-259 (Hardcoded Password)

CWE-330 (Use of Weak Random Values)

CWE-377 (Incomplete Cleanup)

CWE-502 (Deserialization of Untrusted Data)

CWE-732 (Incorrect Permission Assignment)

CWE-838 (Insecure Handling of Sensitive Data)

Analysis:

Flask, being a web framework, is prone to input validation vulnerabilities (CWE-20, CWE-79) and injection attacks (CWE-94, CWE-502).

Path traversal (CWE-22) is another frequent issue, possibly due to improper file access controls.

2. LangChain (AI/LLM Framework)

Top 10 CWEs Identified:

CWE-20 (Improper Input Validation)

CWE-79 (Cross-Site Scripting)

CWE-94 (Code Injection)

CWE-502 (Deserialization of Untrusted Data)

CWE-611 (XML External Entity Injection)

CWE-327 (Use of Broken or Risky Cryptographic Algorithm)

CWE-798 (Use of Hardcoded Credentials)

CWE-918 (Server-Side Request Forgery)

CWE-862 (Missing Authorization)

CWE-937 (Insecure Third-Party Components)

Analysis:

LangChain is focused on AI/LLM processing, making it vulnerable to injection-based vulnerabilities like CWE-94 (Code Injection) and CWE-502 (Deserialization issues).

Cryptographic weaknesses (CWE-327, CWE-798) are more common in LangChain due to its interaction with various APIs and security tokens.

CWE-918 (SSRF) suggests risks related to network-based attacks, possibly from third-party API integrations.

3. Scikit-learn (Machine Learning Library)

Top 10 CWEs Identified:

CWE-20 (Improper Input Validation)

CWE-94 (Code Injection)

CWE-502 (Deserialization of Untrusted Data)

CWE-611 (XML External Entity Injection)

CWE-327 (Use of Broken or Risky Cryptographic Algorithm)

CWE-125 (Out-of-bounds Read)

CWE-119 (Improper Restriction of Operations within Memory Bounds)

CWE-190 (Integer Overflow or Wraparound)

CWE-400 (Uncontrolled Resource Consumption)

CWE-922 (Insecure Storage of Sensitive Information)

Analysis:

Scikit-learn, being a machine learning library, has a high frequency of memory-related vulnerabilities (CWE-125, CWE-119, CWE-190), likely due to the handling of large numerical computations.

Improper input validation (CWE-20) and insecure deserialization (CWE-502) are significant concerns, given its interaction with user-defined models and datasets.

CWE-400 (Uncontrolled Resource Consumption) highlights the risk of denial-of-service (DoS) attacks due to excessive resource use.

Takeaway:

Flask has web security vulnerabilities (XSS, Path Traversal, Injection).

LangChain has AI/LLM-related security risks, including code injection, deserialization issues, and insecure cryptographic practices.

Scikit-learn is prone to memory vulnerabilities, resource exhaustion, and numerical computation risks.

Overall, input validation (CWE-20), code injection (CWE-94), and insecure deserialization (CWE-502) are the most frequent vulnerabilities across all three OSS repositories. Developers should prioritize strong input sanitization, secure coding practices, and careful handling of user inputs and external components.

4. Discussion and Conclusion :

Challenges

Parsing and Commit Indexing Issues:
Early in the assignment, parsing the TXT file to generate CSVs proved problematic. The commit index was initially set to 1 for every entry because the incremental logic wasn't correctly implemented. This led to difficulties when plotting trends over time.

Commit Timestamp Retrieval:
When running the vulnerability analysis on a combined folder (with files from multiple repositories), errors like "fatal: bad object …" occurred because Git couldn't find the commit hash in a non-repository folder. The solution required mapping each commit to its proper repository using the git -C command.

Interpreting Bandit Output:
Understanding and extracting CWEs from Bandit's JSON output was another challenge. The tool stores CWE data within nested fields, and initial scripts did not correctly extract this information.

Handling Domain-Specific Vulnerabilities:
Each repository (Flask, LangChain, and Scikit-Learn) exhibited distinct security patterns, which meant that a one-size-fits-all analysis wasn't possible. This required domain-specific interpretation and adjustments in the analysis methodology.

Reflections

Security Analysis is Iterative:
The assignment emphasized that vulnerability analysis is not a one-time task. As new code is introduced, both new vulnerabilities and fixes occur, and these trends must be continuously monitored.

Automated Tools Need Manual Validation:
Tools like Bandit are invaluable for identifying potential security issues. However, false positives and context-specific flags mean that human judgment is crucial in determining which issues are critical.

Importance of Data Organization:
Properly organizing data—whether by ensuring correct commit indexing or mapping commit timestamps to their respective repositories—proved essential for drawing meaningful conclusions from the visualizations.

Domain Differences Matter:
The security challenges faced by a web framework like Flask differ significantly from those encountered in AI/LLM frameworks like LangChain or machine learning libraries like Scikit-Learn. Recognizing these differences helped tailor the analysis and propose relevant recommendations.

Lessons Learned

Robust Parsing and Data Validation:
Ensuring that data is accurately parsed from text files into structured CSV formats is fundamental. Sorting by timestamps and correctly assigning commit indices are critical steps to avoid misinterpretation of trends.

Error Handling in Automation:
Anticipate and manage errors such as "fatal: bad object …" by validating commit hashes within their appropriate Git repositories. This reinforces the need for contextual error handling in automated analysis pipelines.

Integration of Automated and Manual Analysis:
While tools like Bandit can flag many issues, a manual review is necessary to filter out false positives and understand the context of reported vulnerabilities.

Tailoring Security Approaches to the Domain:
Different software domains require different security strategies. For example, web frameworks need robust input validation and protection against injection attacks, while AI/ML frameworks may need stronger controls around deserialization and cryptographic practices.

Continuous Improvement:
Security is a moving target. Regular audits and continuous integration of security tools into development workflows are crucial to maintaining a strong security posture.

Errors and Unexpected Findings

Git Commit Retrieval Error:
Running vulnerability analysis in a folder containing JSON files from multiple repositories initially led to Git errors because commit hashes weren't recognized in the current working directory. This was resolved by mapping each commit to its respective repository using the git -C command.

Uniform Commit Indices:
The initial CSV parsing resulted in every commit being labeled as "1" due to flawed logic. By switching to a timestamp-based sorting mechanism and reassigning indices based on the sorted order, the issue was fixed.

Inconsistent CWE Extraction:
CWEs were not being captured correctly due to the nested structure in Bandit's output. Adjustments were made to extract CWE IDs properly, ensuring that each repository's most frequent vulnerabilities could be accurately analyzed.

Summary

This assignment provided a comprehensive exploration of security vulnerabilities in three distinct OSS repositories: Flask, LangChain, and Scikit-Learn. By employing Bandit for static code analysis and developing custom scripts to process and visualize the data, several key insights emerged:

Flask exhibits a rising trend in high-severity vulnerabilities, likely driven by web-specific security challenges.

LangChain shows a constant level of high-severity issues, indicating stagnation in resolving critical vulnerabilities.

Scikit-Learn displays minor fluctuations, suggesting that while some vulnerabilities are fixed, new ones are introduced periodically.

Ultimately, the analysis underscores the importance of continuous security auditing, domain-specific mitigation strategies, and the integration of both automated tools and manual reviews to improve the security posture of open-source projects.