

CS202 - Lab Assignment 5

- Hitesh Kumar

- 22110098

Commit Hash - cad4754bc71742c2d6fcbdb3b92ae74834d359844

Setup for Pytest

1. Clone the GitHub repository *keon/algorithms*:

```
$ git clone https://github.com/keon/algorithms
```

2. Install the *test_requirements* and dependencies after setting up venv:

```
$ python3 -m venv venv
```

```
$ source venv/bin/activate
```

```
$ cd algorithms
```

```
$ pip install -r test_requirements
```

```
$ pip install -r docs/requirements
```

```
$ pip install -e .
```

Running pytest for Existing Tests (Test Suite A)

1. Configure *pytest.ini*

[pytest]

- **addopts**: Specifies additional command-line options for pytest.

Here, it includes:

- **--cov=algorithms**: Enables code coverage measurement for the algorithms module.

```
[pytest]
addopts = --cov=algorithms --cov-report=html:testAreport
testpaths = tests/
```

- `--cov-report=html:testAreport`: Generates an HTML report of the coverage and saves it in the testAreport directory.
- `testpaths`: Specifies the directories to search for tests. Here, it is set to the tests/ directory.

2. Configure `.coveragerc`

```
1 [report]
2 omit =
3     */python?.?/*
4     */site-packages/nose/*
5     *__init__*
```

```
7 [run]
8 source = algorithms
9 omit =
10     *tests/*
```

[report] Section: (This part was already there in the `.coveragerc`)

- `omit`: Specifies patterns for files to exclude from the coverage report.
 - `*/python?.?/*`: Excludes files in directories matching `python?.?` (e.g., `python3.8`).
 - `*/site-packages/nose/*`: Excludes files in the nose package.
 - `*__init__*`: Excludes `__init__` files.

[run] Section

- `source`: Specifies the source directory to measure coverage for, in this case, the algorithms directory.
- `omit`: Specifies patterns for files to exclude during the coverage run.
 - `*tests/*`: Excludes files in the tests directory.

3. Running `pytest`

Simply run `pytest` as the `pytest.ini` file has already been configured.

```
$ pytest
```

After running `pytest`, a folder `testAreport` consisting of HTML files for the coverage report will be created. Run the `class_index.html` on the Live Server (as shown in *Fig: Coverage Report*).

Analysis of Coverage Report for Test Suite A

Coverage report: 69%

Files Functions Classes

coverage.py v7.6.10, created at 2025-02-06 10:56 +0530

File	class	statements	missing	excluded	coverage ▲
algorithms/dp/longest_common_subsequence.py	(no class)	12	12	0	0%
algorithms/graph/clone_graph.py	UndirectedGraphNode	4	4	0	0%
algorithms/graph/clone_graph.py	(no class)	52	52	0	0%
algorithms/graph/find_all_cliques.py	(no class)	22	22	0	0%
algorithms/graph/markov_chain.py	(no class)	16	16	0	0%
algorithms/graph/minimum_spanning_tree.py	Edge	3	3	0	0%
algorithms/graph/minimum_spanning_tree.py	DisjointSet	14	14	0	0%
algorithms/graph/minimum_spanning_tree.py	(no class)	32	32	0	0%
algorithms/graph/satisfiability.py	(no class)	70	70	0	0%
algorithms/graph/transitive_closure_dfs.py	Graph	13	13	0	0%
algorithms/graph/transitive_closure_dfs.py	(no class)	5	5	0	0%
algorithms/graph/traversal.py	(no class)	28	28	0	0%
algorithms/heap/merge_sorted_k_lists.py	ListNode	2	2	0	0%
algorithms/linkedlist/add_two_numbers.py	Node	2	2	0	0%
algorithms/linkedlist/add_two_numbers.py	TestSuite	34	34	0	0%
algorithms/linkedlist/add_two_numbers.py	(no class)	47	47	0	0%
algorithms/linkedlist/delete_node.py	Node	2	2	0	0%
algorithms/linkedlist/delete_node.py	TestSuite	19	19	0	0%

Fig: Coverage Report

- **Overall Coverage:** The codebase is at a 69% coverage level, leaving about 31% of the code untested.
- **Consistent vs. Inconsistent Coverage:** Simple modules are thoroughly tested (100%), but many complex modules (e.g., tree, graph, dynamic programming, and polynomial algorithms) show low or even zero coverage.
- **Testing Focus:** The uneven distribution of tests suggests a need to focus on adding tests for modules with complex conditions and branches.

Now, we will generate the automated test cases using `pynguin` for the `tree` module as many files in the `tree` module have 0% coverage with an overall coverage of only 37%.


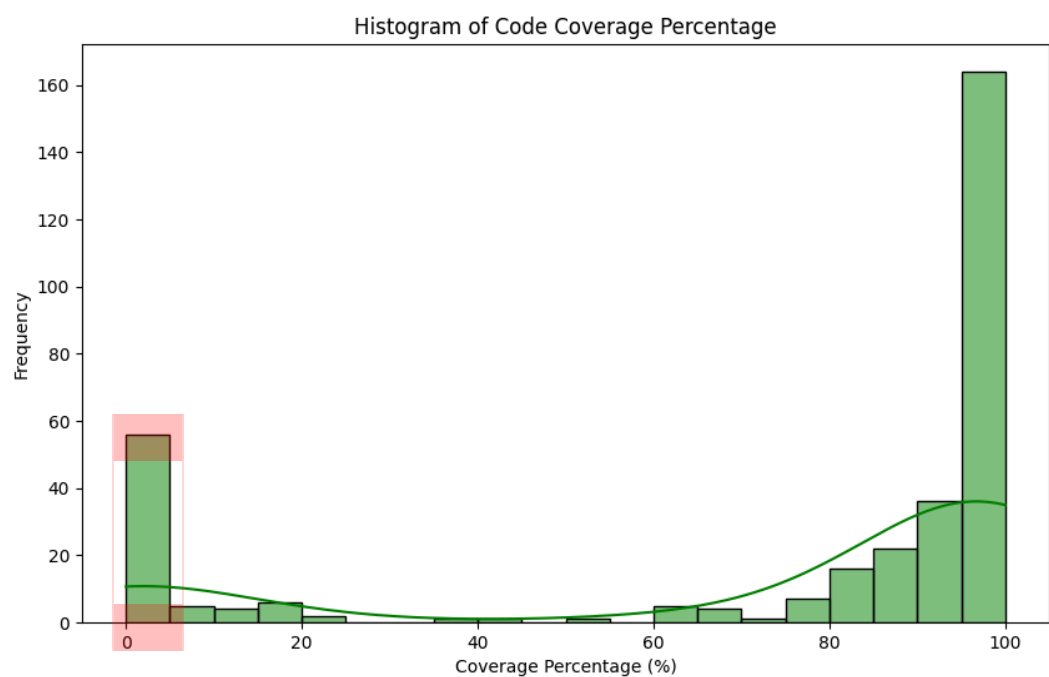
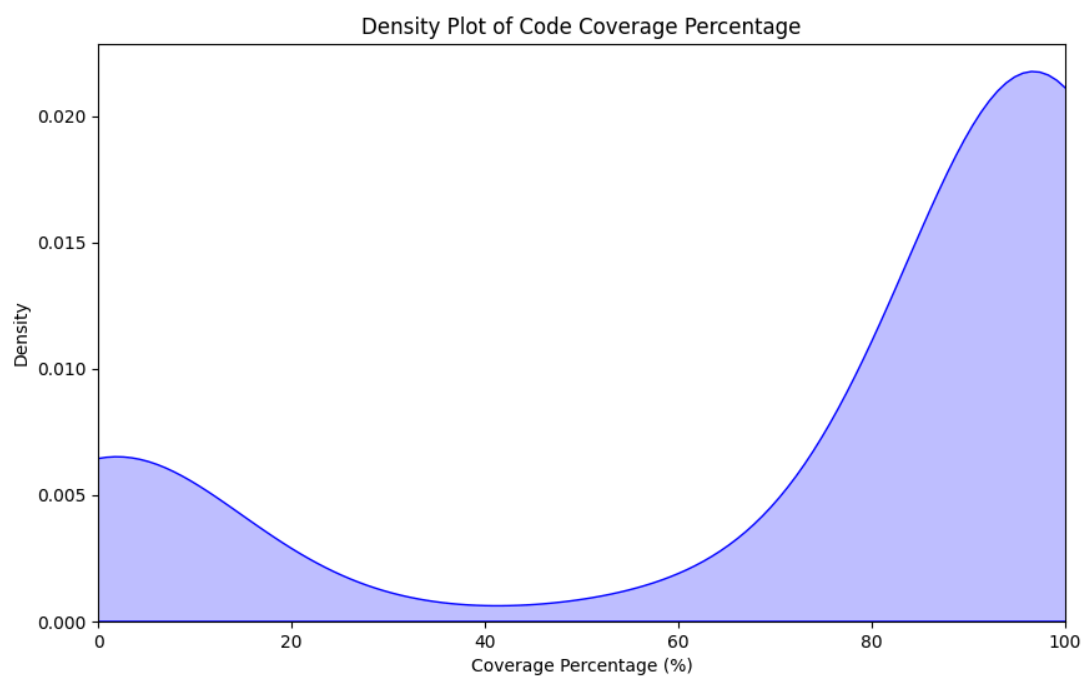
File	class	statements	missing	excluded	coverage 
algorithms/tree/avl/avl.py	AvlTree	67	67	0	0%
algorithms/tree/avl/avl.py	(no class)	10	10	0	0%
algorithms/tree/bin_tree_to_list.py	(no class)	28	28	0	0%
algorithms/tree/binary_tree_paths.py	(no class)	13	13	0	0%
algorithms/tree/deepest_left.py	DeepestLeft	2	2	0	0%
algorithms/tree/deepest_left.py	(no class)	23	23	0	0%
algorithms/tree/invert_tree.py	(no class)	8	8	0	0%
algorithms/tree/is_balanced.py	(no class)	12	12	0	0%
algorithms/tree/is_subtree.py	(no class)	19	19	0	0%
algorithms/tree/is_symmetric.py	(no class)	25	25	0	0%
algorithms/tree/longest_consecutive.py	(no class)	15	15	0	0%
algorithms/tree/lowest_common_ancestor.py	(no class)	8	8	0	0%
algorithms/tree/max_height.py	(no class)	33	33	0	0%
algorithms/tree/max_path_sum.py	(no class)	11	11	0	0%
algorithms/tree/min_height.py	(no class)	40	40	0	0%
algorithms/tree/path_sum.py	(no class)	35	35	0	0%
algorithms/tree/path_sum2.py	(no class)	42	42	0	0%
algorithms/tree/pretty_print.py	(no class)	10	10	0	0%
algorithms/tree/same_tree.py	(no class)	6	6	0	0%
algorithms/tree/traversal/inorder.py	Node	3	3	0	0%
algorithms/tree/traversal/level_order.py	(no class)	17	17	0	0%
algorithms/tree/traversal/postorder.py	Node	3	3	0	0%
algorithms/tree/traversal/preorder.py	Node	3	3	0	0%
algorithms/tree/traversal/zigzag.py	(no class)	19	19	0	0%
algorithms/tree/tree.py	TreeNode	3	3	0	0%
algorithms/tree/tree.py	(no class)	2	2	0	0%
algorithms/tree/traversal/inorder.py	(no class)	37	13	0	65%
algorithms/tree/b_tree.py	Node	4	1	0	75%
algorithms/tree/construct_tree_postorder_preorder.py	(no class)	39	7	0	82%
algorithms/tree/b_tree.py	BTree	125	17	0	86%
algorithms/tree/traversal/postorder.py	(no class)	28	1	0	96%
algorithms/tree/traversal/preorder.py	(no class)	25	1	0	96%
algorithms/tree/b_tree.py	(no class)	22	0	0	100%
algorithms/tree/construct_tree_postorder_preorder.py	TreeNode	3	0	0	100%
algorithms/tree/fenwick_tree/fenwick_tree.py	Fenwick_Tree	16	0	0	100%
algorithms/tree/fenwick_tree/fenwick_tree.py	(no class)	5	0	0	100%
algorithms/tree/segment_tree/iterative_segment_tree.py	SegmentTree	20	0	0	100%
algorithms/tree/segment_tree/iterative_segment_tree.py	(no class)	5	0	0	100%
Total		786	497	0	37%

Fig: Coverage for *tree* module



Setting Up for Running Pynguin

1. Install pynguin

```
$ pip install pynguin
```

2. Script

```
1  #!/bin/bash
2
3  # Base directories
4  MODULE_DIR="algorithms/$1"
5  OUTPUT_DIR="testsB"
6
7  # Ensure the output directory exists
8  mkdir -p $OUTPUT_DIR
9
10 # Iterate over each Python file in the module directory
11 find $MODULE_DIR -type f -name "*.py" | while read -r file; do
12     # Extract the module name and path
13     module_path=$(dirname "$file")
14     module_name=$(basename "$file" .py)
15
16     # Run Pynguin
17     echo "Running Pynguin for module $module_name"
18     pynguin --project-path="$module_path" --module-name="$module_name" --output-path="$OUTPUT_DIR" --export-strategy=PY_TEST
19 done
```

- **Iterate Over Python Files:**
 - `find $MODULE_DIR -type f -name "*.py"` finds all Python files in the specified module directory.
 - The script extracts the module path and names for each Python file found.
 - **Run Pynguin:** The script runs Pynguin with the appropriate parameters to generate unit tests and save them in the output directory for each module.
-

Running Pynguin

Run the given script inside the base folder `algorithms` with the command-line argument, as in our case, is `'tree'`, so the command will be:

```
$ pynguin_script.sh tree
```

Sample test generated by the given script for a file is shown below:

```
# Test cases automatically generated by Pynguin (https://www.pynguin.eu).
# Please check them before you use them.
import pytest
import binary_tree_paths as module_0

def test_case_0():
    none_type_0 = None
    var_0 = module_0.binary_tree_paths(none_type_0)

@pytest.mark.xfail(strict=True)
def test_case_1():
    bool_0 = False
    module_0.binary_tree_paths(bool_0)

@pytest.mark.xfail(strict=True)
def test_case_2():
    bool_0 = True
    module_0.dfs(bool_0, bool_0, bool_0)
```

Error in Generated Tests: In all of the test files generated, there is a common import error in the files shown in the **red** box in the image above. This is because `pynguin` assumes we will run the test file in the respective directory. The error shows that it can't import a given module. This can be fixed by adding `'from algorithms.tree'` at the start of the import statement. Now, it will import `module_0` from the directory `'algorithms/tree'`, and then it will be able to execute the function from the files in that directory. Also, we need to change the name of the test files as the name of the generated test files may overlap with the previously available files. I added `_B` at the end of the file name to prevent that. A simple script can be made to make all these changes (in import and filename) as shown below:

```

1  #!/bin/bash
2
3  # Directory containing the Python files
4  DIRECTORY="testsB"
5
6  find "$DIRECTORY" -name "*.py" | while read -r file; do
7      # Update the import statements
8      sed -i 's/import \(.*\) as module_/from algorithms.tree import \1 as module_/' "$file"
9
10     # Change filename
11     dir=$(dirname "$file")
12     filename=$(basename "$file" .py)
13     mv "$file" "$dir/${filename}_B.py"
14 done

```

Running Pytest On All Tests (TestSuite A + TestSuite B)

Now let's run `pytest` on all the test cases (previous and new ones) as we want to check how coverage will be affected by these `pynguin`-generated test cases (TestSuite B).

1. Store all the test files (old + new) in a single folder

```
$ cp tests/* all_tests/
```

```
$ cp testsB/* all_tests/
```

2. Make changes to the `pytest.ini` and `.coveragerc` files:

```

1  [pytest]
2  addopts = --cov=algorithms --cov-report=html:testBreport
3  testpaths = all_tests/

```

```

[run]
source = algorithms
omit =
    *tests/*
    *testsB/*
    *all_tests/*

```

3. Run `pytest`

```
$ pytest
```

Analysis of Coverage Report

Coverage report: 71%

Files

Functions

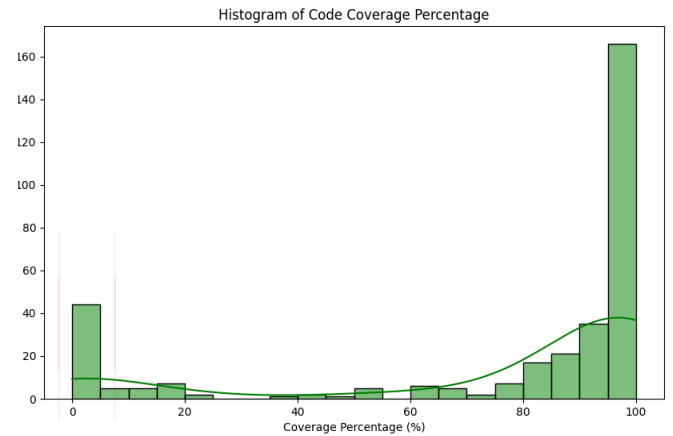
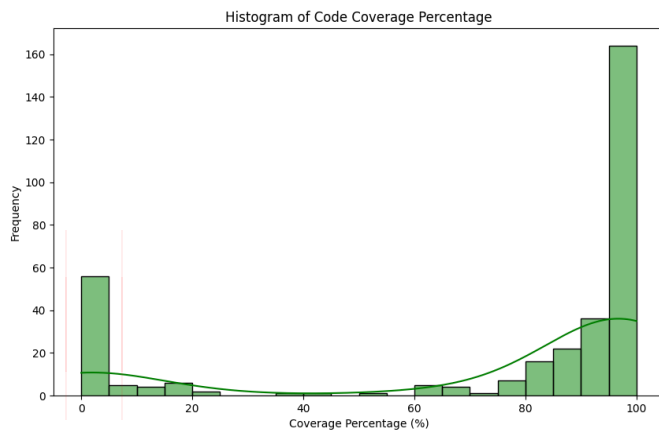
Classes

coverage.py v7.6.10, created at 2025-02-24 14:47 +0530

File	class	statements	missing	excluded	coverage
algorithms/tree/avl/avl.py	AvlTree	67	67	0	0%
algorithms/tree/avl/avl.py	(no class)	10	10	0	0%
algorithms/tree/b_tree.py	Node	4	0	0	100%
algorithms/tree/b_tree.py	BTree	125	4	0	97%
algorithms/tree/b_tree.py	(no class)	22	0	0	100%
algorithms/tree/bin_tree_to_list.py	(no class)	28	28	0	0%
algorithms/tree/binary_tree_paths.py	(no class)	13	6	0	54%
algorithms/tree/construct_tree_postorder_preorder.py	TreeNode	3	0	0	100%
algorithms/tree/construct_tree_postorder_preorder.py	(no class)	39	7	0	82%
algorithms/tree/deepest_left.py	DeepestLeft	2	2	0	0%
algorithms/tree/deepest_left.py	(no class)	23	23	0	0%
algorithms/tree/fenwick_tree/fenwick_tree.py	Fenwick_Tree	16	0	0	100%
algorithms/tree/fenwick_tree/fenwick_tree.py	(no class)	5	0	0	100%
algorithms/tree/invert_tree.py	(no class)	8	4	0	50%
algorithms/tree/is_balanced.py	(no class)	12	4	0	67%
algorithms/tree/is_subtree.py	(no class)	19	5	0	74%
algorithms/tree/is_symmetric.py	(no class)	25	12	0	52%
algorithms/tree/longest_consecutive.py	(no class)	15	15	0	0%
algorithms/tree/lowest_common_ancestor.py	(no class)	8	4	0	50%
algorithms/tree/max_height.py	(no class)	33	33	0	0%
algorithms/tree/max_path_sum.py	(no class)	11	4	0	64%
algorithms/tree/min_height.py	(no class)	40	21	0	48%
algorithms/tree/path_sum.py	(no class)	35	35	0	0%
algorithms/tree/path_sum2.py	(no class)	42	25	0	40%
algorithms/tree/pretty_print.py	(no class)	10	0	0	100%
algorithms/tree/same_tree.py	(no class)	6	1	0	83%
algorithms/tree/segment_tree/iterative_segment_tree.py	SegmentTree	20	0	0	100%
algorithms/tree/segment_tree/iterative_segment_tree.py	(no class)	5	0	0	100%
algorithms/tree/traversal/inorder.py	Node	3	3	0	0%
algorithms/tree/traversal/inorder.py	(no class)	37	13	0	65%
algorithms/tree/traversal/level_order.py	(no class)	17	17	0	0%
algorithms/tree/traversal/postorder.py	Node	3	3	0	0%
algorithms/tree/traversal/postorder.py	(no class)	28	1	0	96%
algorithms/tree/traversal/preorder.py	Node	3	3	0	0%
algorithms/tree/traversal/preorder.py	(no class)	25	1	0	96%
algorithms/tree/traversal/zigzag.py	(no class)	19	19	0	0%
algorithms/tree/tree.py	TreeNode	3	0	0	100%
algorithms/tree/tree.py	(no class)	2	0	0	100%
Total		786	370	0	53%

coverage.py v7.6.10, created at 2025-02-24 14:47 +0530

Penguin-generated test cases helped to increase the overall coverage from 69% to 71%. Specifically, the tree module's coverage increased from 37% to 53%.



Comparing the Coverage Percentage Histogram, we observe a significant reduction in files with 0% coverage. This indicates that Pynguin-generated test cases effectively covered previously untested parts of the code.

Observations

Overall, the automated test generation improved code coverage. This demonstrates Pynguin's ability to enhance test completeness and identify untested scenarios, leading to more reliable software.

CS202 - Lab Assignment 6

- Hitesh Kumar
- 22110098

Commit Hash - cad4754bc71742c2d6fcbdb3b92ae74834d359844

NOTE: The setup for `pytest` is already done in Assignment 5.

Sequential Test Execution

Run `pytest` on 'keon/algorithms' 10 times with the same TestSuite A to remove failing and flaky. Some test cases were found to fail.

```
===== FAILURES =====
TestRemoveDuplicate.test_remove_duplicates

self = <test_array.TestRemoveDuplicate testMethod=test_remove_duplicates>

    def test_remove_duplicates(self):
>     self.assertListEqual(remove_duplicates([1,1,1,2,2,2,3,3,4,4,5,6,7,7,7,8,9,10,10]))
E     TypeError: TestCase.assertListEqual() missing 1 required positional argument: 'list2'

tests/test_array.py:305: TypeError

TestSummaryRanges.test_summarize_ranges

self = <test_array.TestSummaryRanges testMethod=test_summarize_ranges>

    def test_summarize_ranges(self):
>     self.assertListEqual(summarize_ranges([0, 1, 2, 4, 5, 7]),
                             [(0, 2), (4, 5), (7, 7)])
E     AssertionError: Lists differ: ['0-2', '4-5', '7'] != [(0, 2), (4, 5), (7, 7)]
E
E     First differing element 0:
E     '0-2'
E     (0, 2)
E
E     - ['0-2', '4-5', '7']
E     + [(0, 2), (4, 5), (7, 7)]

tests/test_array.py:349: AssertionError

----- coverage: platform linux, python 3.10.16-final-0 -----
Coverage HTML written to dir htmlcov

===== short test summary info =====
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)]
===== 2 failed, 414 passed in 11.95s =====
```

After removing these and the flaky test cases (passing on some runs but failing on others - non-deterministic behaviour), if any are found. Make sure no test cases are failing.

Now, on running `pytest` 5 times to calculate the average sequential execution time on TestSuite A. Let's denote the average sequential execution time as `T_seq`.

Run the following python code `lab6.py` to calculate the `T_seq`.

```
def compute_Tseq(num_runs=5):
    print("\n=== Running Sequential Tests for Timing Measurement (Stable Suite) ===")
    times = []
    for i in range(num_runs):
        print(f"Timing run {i+1}/{num_runs}")
        elapsed, failures, _ = run_pytest(["pytest", "tests/"])
        if failures != 0:
            print("WARNING: Failures detected in a timing run; expected a stable suite.")
        times.append(elapsed)
        print(f"   Time: {elapsed:.2f} sec")
    Tseq = statistics.mean(times)
    print(f"\nAverage Sequential Execution Time (Tseq) over {num_runs} runs: {Tseq:.2f} sec")
    return Tseq
```

Parallel Test Execution

Function for running the parallel execution:

```
def run_parallel_tests():
    print("\n=== Running Parallel Test Executions ===")
    n_values = ["1", "auto"]
    parallel_threads_values = ["1", "auto"]
    dist_modes = ["load", "no"]

    configurations = []
    for n in n_values:
        for pt in parallel_threads_values:
            for dist in dist_modes:
                name = f"xdist -n {n}, --dist {dist} | run-parallel --parallel-threads {pt}"
                cmd = ["pytest", "-n", n, "--dist", dist, "--parallel-threads", pt, "tests/"]
                configurations.append({"name": name, "cmd": cmd})

    repetitions = 3
    results = {}

    for config in configurations:
        print(f"\nConfiguration: {config['name']}")
        times = []
        failure_counts = []
        all_failed_tests = []
        for i in range(repetitions):
            print(f"   Parallel run {i+1}/{repetitions}")
            elapsed, failures, failed_tests = run_pytest(config["cmd"])
            times.append(elapsed)
            failure_counts.append(failures)
            all_failed_tests.extend(failed_tests)
            print(f"\tTime: {elapsed:.2f} sec | Failures: {failures}")
        avg_time = statistics.mean(times)
        results[config["name"]] = {
            "Tpar": avg_time,
            "times": times,
            "failure_counts": failure_counts,
            "failing_tests": list(set(all_failed_tests)) # unique test names
        }
        print(f"   ⇒ Average Time for '{config['name']}': {avg_time:.2f} sec")
    return results
```

The given script executes parallel execution with the defined configurations 3 times each. It calculates the average execution time of those 3 repetitions for each configuration. It also displays the failures in the tests by running them in parallel. Figure given below shows the output of the code.

```
=== Final Summary ===
Average Sequential Time (Tseq): 8.49 sec
xdist -n 1, --dist load | run-parallel --parallel-threads 1: Tpar = 9.28 sec | Speedup = 0.91 | Failure counts per run: [0, 0, 0]
xdist -n 1, --dist no | run-parallel --parallel-threads 1: Tpar = 9.00 sec | Speedup = 0.94 | Failure counts per run: [0, 0, 0]
xdist -n 1, --dist load | run-parallel --parallel-threads auto: Tpar = 93.96 sec | Speedup = 0.09 | Failure counts per run: [3, 4, 3]
Failing tests: tests/test_heap.py, tests/test_linkedlist.py, tests/test_compression.py
xdist -n 1, --dist no | run-parallel --parallel-threads auto: Tpar = 92.21 sec | Speedup = 0.09 | Failure counts per run: [3, 3, 4]
Failing tests: tests/test_heap.py, tests/test_linkedlist.py, tests/test_compression.py
xdist -n auto, --dist load | run-parallel --parallel-threads 1: Tpar = 8.40 sec | Speedup = 1.01 | Failure counts per run: [0, 0, 0]
xdist -n auto, --dist no | run-parallel --parallel-threads 1: Tpar = 8.58 sec | Speedup = 0.99 | Failure counts per run: [0, 0, 0]
xdist -n auto, --dist load | run-parallel --parallel-threads auto: Tpar = 32.77 sec | Speedup = 0.26 | Failure counts per run: [3, 3, 3]
Failing tests: tests/test_heap.py, tests/test_linkedlist.py
xdist -n auto, --dist no | run-parallel --parallel-threads auto: Tpar = 32.73 sec | Speedup = 0.26 | Failure counts per run: [3, 3, 3]
Failing tests: tests/test_heap.py, tests/test_linkedlist.py
=== End of Summary ===
```

Analysis of Results

Below is a matrix detailing parallelisation mode, worker count, average execution time, and failure rates based on the above results.

Below is a Markdown table summarising the eight configurations and their results:

Configuration	Tpar (sec)	Speedup	Failure Counts per Run	Failing Tests (Unique)
xdist -n 1, --dist load run-parallel --parallel-threads 1	9.28	0.91	[0, 0, 0]	None

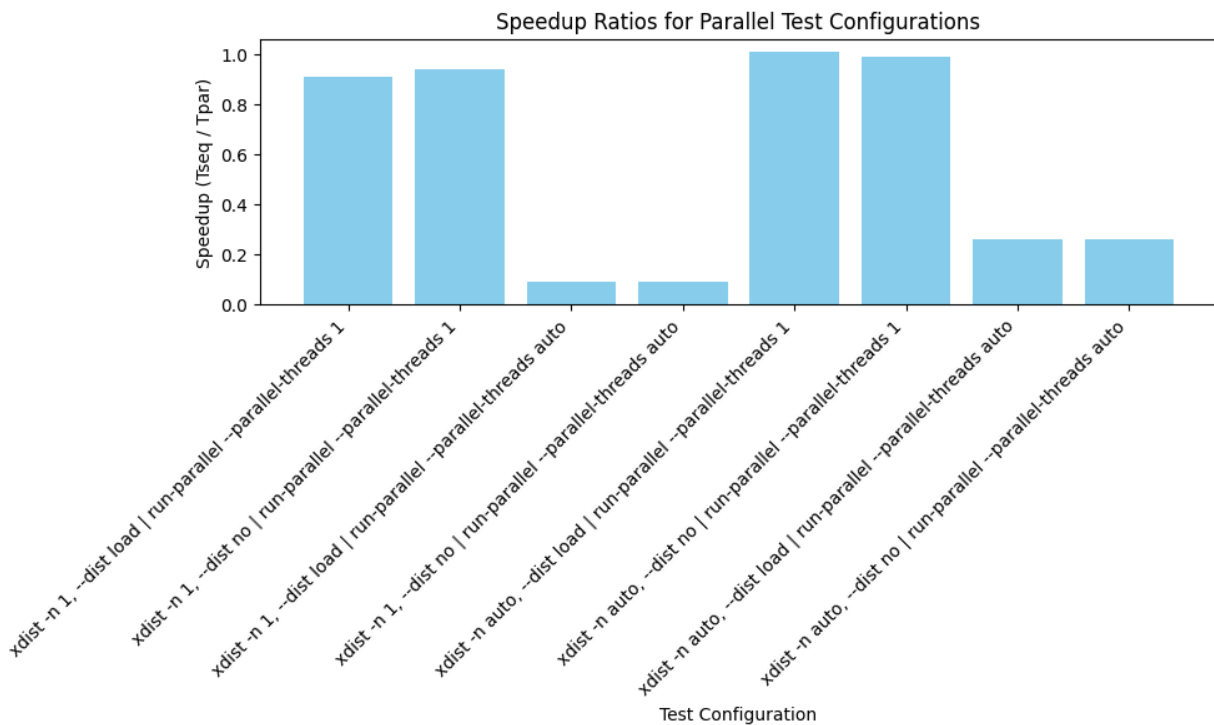
xdist -n 1, --dist no run-parallel --parallel-threads 1	9.00	0.94	[0, 0, 0]	None
xdist -n 1, --dist load run-parallel --parallel-threads auto	93.96	0.09	[3, 4, 3]	tests/test_heap.py, tests/test_linkedlist.py, tests/test_compression.py
xdist -n 1, --dist no run-parallel --parallel-threads auto	92.21	0.09	[3, 3, 4]	tests/test_heap.py, tests/test_linkedlist.py, tests/test_compression.py
xdist -n auto, --dist load run-parallel --parallel-threads 1	8.40	1.01	[0, 0, 0]	None
xdist -n auto, --dist no run-parallel --parallel-threads 1	8.58	0.99	[0, 0, 0]	None

<pre>xdist -n auto, --dist load run-parallel --parallel-threads auto</pre>	32.77	0.26	[3, 3, 3]	tests/test_heap.py, tests/test_linkedlist.py
<pre>xdist -n auto, --dist no run-parallel --parallel-threads auto</pre>	32.73	0.26	[3, 3, 3]	tests/test_heap.py, tests/test_linkedlist.py

Baseline: Average Sequential Time (Tseq): 8.49 sec

Failure: As you can see, when running those tests with `pytest-run-parallel`, 3 test cases failed from the `heap`, `compression` and `linked-list` module. The possible reason for the failure of those test cases could be that the code written for handling `heap` and `linked-list` data structures doesn't support parallelisation (e.g. when two pieces of code try to append a node together, they may end up adding the node to a common node, creating a branched linked-list instead of a linear one.)

The speedup ratios for parallel test configurations are shown in the plot below. The speedup in the case of `xdist` is almost 1 reason being, the overhead for the distribution of load over multiple cores. As the code is not parallelisable, the overhead is not overcome by the faster execution and ends up taking a little more time than sequential execution.



Suggestions

The project is not fully ready for parallel testing, especially with thread-based parallelization, due to concurrency issues in certain tests like `test_heap.py`, `test_compression.py` and `test_linkedList.py`. These failures suggest that the test suite relies on shared state or mutable global resources, which are problematic in a multi-threaded environment.

Refactoring the test suite to ensure thread safety or utilizing process-based parallelization (e.g., `pytest-xdist`) is a nice strategy to improve parallel testing readiness. For pytest developers, better thread safety mechanisms or warning systems around shared resources would help improve detection and isolating tests that modify the global state unexpectedly.

CS202 - Lab Assignment 7-8

- Hitesh Kumar

- 22110098

Setup for Bandit

1. Install **bandit** in a Python virtual environment

```
$ python3 -m venv venv
```

```
$ source venv/bin/activate
```

```
$ pip install bandit
```

You can also install the requirements from each repository in the virtual environment. The requirements will be presented in the **requirements.txt** file.

Repository Selection

- **Language & Application:** The repository must be primarily Python and represent a real-world application.
- **Popularity Metrics:** A minimum of 10,000+ GitHub stars and 2000+ forks.
- **Community Engagement:** The repositories should have at least 200 active contributors and recent commit activity (e.g., within the last 6 months) and a minimum of 10,000 commits.
- **Project Maturity:** Established projects with versioned releases and a sizable codebase.
- **Large Repository:** With at least 1,00,000 code lines.
- **Tool Assistance:** The SEART GitHub Search Engine filters projects based on these criteria.

The 3 selected repositories are:

1. Django

 **django/django** 

Commits: 33380	👁 Watchers: 2294	☆ Stars: 82505	🍴 Forks: 32295
🕒 Total Issues: 0	📄 Total Pull Req: 19149	🌿 Branches: 29	👤 Contributors: 393
🔴 Open Issues: 0	🔴 Open Pull Req: 293	📦 Releases: 0	📦 Size: 259.31 KB
+ Created: 2012-04-28	🕒 Updated: 2025-02-27	↑ Last Push: 2025-02-25	📅 Last Commit: 2025-02-25
<> Code Lines: 774,455	📄 Comment Lines: 73,938	Blank Lines: 202,023	
# Last Commit SHA: 5a1cae3a5675c5733daf5949759476d65aa0e636			

Show More


2. Scikit-learn

 [scikit-learn/scikit-learn](https://github.com/scikit-learn/scikit-learn) 

Commits: 32204	👁 Watchers: 2146	☆ Stars: 61248	🍴 Forks: 25633
🕒 Total Issues: 11397	🔗 Total Pull Req: 18423	🌿 Branches: 35	👤 Contributors: 410
🔴 Open Issues: 1569	🔴 Open Pull Req: 563	📦 Releases: 45	📦 Size: 162.8 KB
+ Created: 2010-08-17	📅 Updated: 2025-02-27	↑ Last Push: 2025-02-26	📅 Last Commit: 2025-02-26
<> Code Lines: 282,371	💬 Comment Lines: 134,380	Blank Lines: 88,442	
# Last Commit SHA: 5eb676ac9afd4a5d90cdda198d174c2c8d2da226			

Show More

3. Matplotlib

 [matplotlib/matplotlib](https://github.com/matplotlib/matplotlib) 

Commits: 51997	👁 Watchers: 589	☆ Stars: 20791	🍴 Forks: 7772
🕒 Total Issues: 10721	🔗 Total Pull Req: 18913	🌿 Branches: 26	👤 Contributors: 420
🔴 Open Issues: 1225	🔴 Open Pull Req: 387	📦 Releases: 91	📦 Size: 449.41 KB
+ Created: 2011-02-19	📅 Updated: 2025-02-25	↑ Last Push: 2025-02-25	📅 Last Commit: 2025-02-25
<> Code Lines: 606,650	💬 Comment Lines: 98,724	Blank Lines: 75,333	
# Last Commit SHA: 633bb6cb5b8bd9b2ba978e6d89f0a8f0a5c43c91			

Show More

Running Bandit on Selected Repositories

First of all, clone the selected repositories:

```
$ git clone https://github.com/django/django.git
```

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

```
$ git clone https://github.com/matplotlib/matplotlib
```

This task includes collecting the repository's last 100 **non-merge** commits (from older to newer, i.e. in chronological order) and running **bandit** on each commit to get the vulnerability report. We also have to prevent bandit from running on the tests folder, for which we can use **-x** or **--exclude** argument in bandit. The bash script given below does precisely that. It extracts the last 500 non-merge commits of the repository and iteratively **git checkout** to these commits to run bandit for each and store the output in **JSON** format in a folder named **bandit_reports**.

```

bandit.sh
1  #!/bin/bash
2
3  repos=("django" "matplotlib" "scikit-learn")
4
5  for repo in "${repos[@]"; do
6      echo "Processing repository: $repo"
7
8      cd "$repo" || { echo "Failed to enter $repo"; continue; }
9
10     rm -rf bandit_reports
11     mkdir -p bandit_reports
12
13     git log --first-parent --no-merges -n 100 --pretty=format:"%H" --reverse | awk '1; END {print ""}' > commits.txt
14
15     counter=1
16
17     while read commit; do
18         [ -z "$commit" ] && continue
19         [ ${#commit} -ne 40 ] && continue
20
21         git checkout -f "$commit"
22
23         printf -v seq_num "%03d" $counter
24
25         bandit -r . -f json -x ./tests -o "bandit_reports/bandit_${seq_num}_${commit:0:7}.json"
26
27         ((counter++))
28     done < commits.txt
29
30     # Clean up
31     cd ..
32
33     echo "Completed processing: $repo"
34     echo "-----"
35 done

```

Repository Level Analysis

First, we will extract the valuable information from the **JSON** files and make a single **CSV** file containing the **confidence counts**, **severity counts**, and **unique CWE** information. You can use the code in the **to_csv.py** file.

```

import json
import csv
import os
from glob import glob
import argparse

def process_bandit_report(filepath):
    with open(filepath, 'r') as f:
        data = json.load(f)

    # Count confidence levels from each issue in the "results" list.
    results = data.get("results", [])
    conf_high = sum(1 for issue in results if issue.get("issue_confidence", "").upper() == "HIGH")
    conf_med = sum(1 for issue in results if issue.get("issue_confidence", "").upper() == "MEDIUM")
    conf_low = sum(1 for issue in results if issue.get("issue_confidence", "").upper() == "LOW")

    # Count severity levels.
    sev_high = sum(1 for issue in results if issue.get("issue_severity", "").upper() == "HIGH")
    sev_med = sum(1 for issue in results if issue.get("issue_severity", "").upper() == "MEDIUM")
    sev_low = sum(1 for issue in results if issue.get("issue_severity", "").upper() == "LOW")

```

```

# Collect unique CWE IDs.
cwe_set = set()
for issue in results:
    cwe_info = issue.get("issue_cwe")
    if cwe_info and "id" in cwe_info:
        cwe_set.add(f"CWE-{cwe_info['id']}")
unique_cwes = ", ".join(sorted(cwe_set))

# Extract commit hash from the filename.
# Assuming filename pattern: bandit_NUM_COMMIT.json (e.g., bandit_100_136a1e8.json)
filename = os.path.basename(filepath)
parts = filename.split('_')
commit = parts[2].split('.')[0] if len(parts) ≥ 3 else ""

return {
    "commit": commit,
    "conf_high": conf_high,
    "conf_medium": conf_med,
    "conf_low": conf_low,
    "sev_high": sev_high,
    "sev_medium": sev_med,
    "sev_low": sev_low,
    "unique_cwes": unique_cwes,
}

```

We can proceed with the analysis now that we have a **CSV** file containing all the required information. But let's first understand what **confidence** and **severity** mean in the context of **bandit**

Severity: Indicates the potential impact level of a detected security issue on the system.

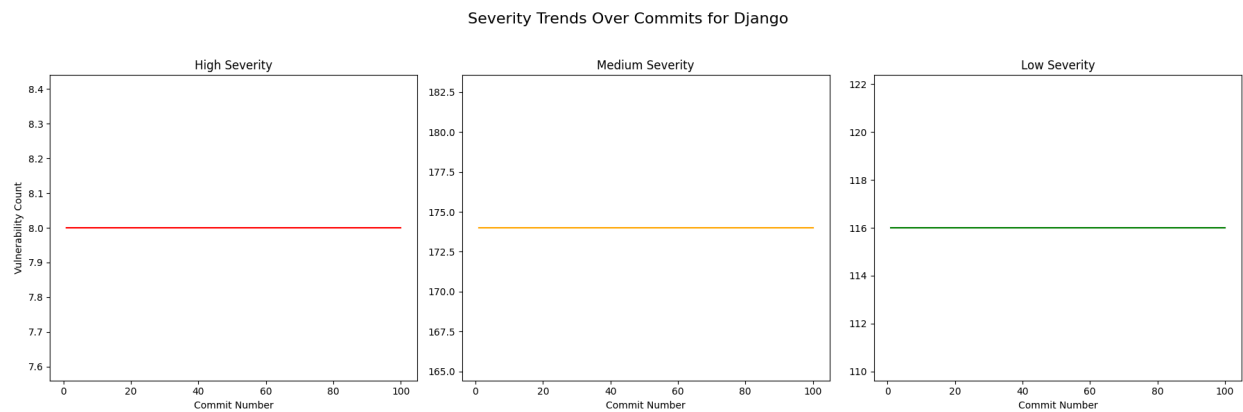
- **High:** Issues that could lead to significant security vulnerabilities if exploited.
- **Medium:** Issues that pose moderate security risks.
- **Low:** Issues with minimal security impact.

Confidence: Reflects the likelihood that a reported issue is a true positive.

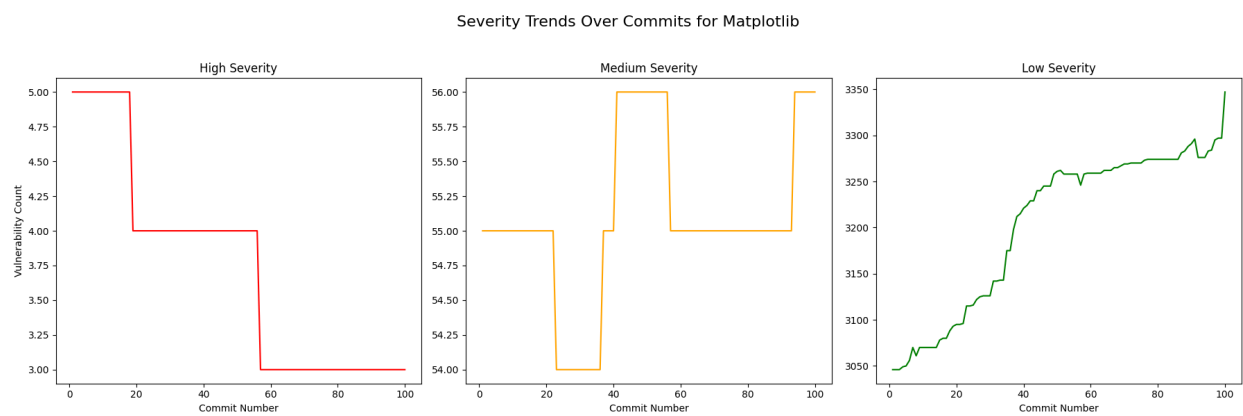
- **High:** The issue is a genuine security concern.
- **Medium:** The issue is likely a security concern, but there's some uncertainty.
- **Low:** The issue might be a false positive; further investigation is needed.

The graphs below show the number of **HIGH**, **MEDIUM** and **LOW** severity issues in the codebase the **bandit** identifies for each commit.

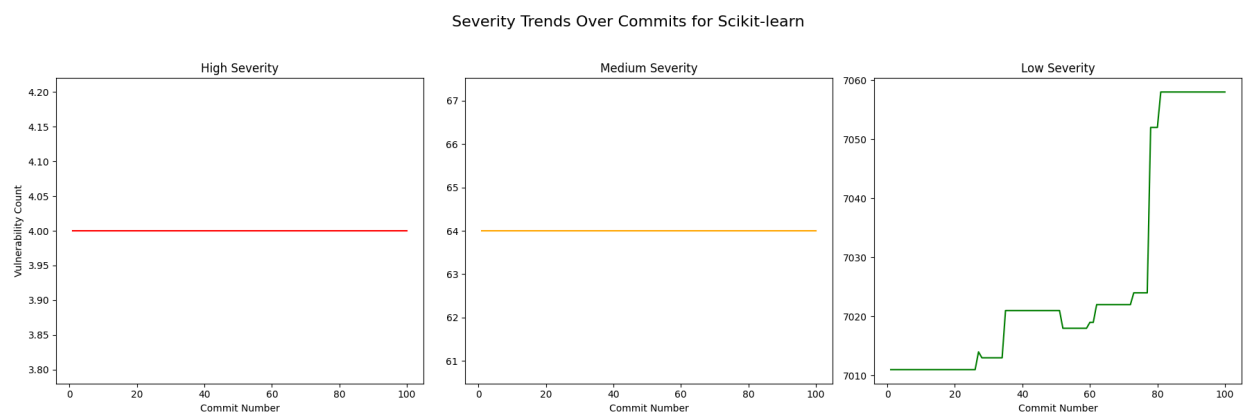
Django



Matplotlib



Scikit-learn



→ All the 100 commits in the **Django** Repository contain the following CWEs: CWE-20, CWE-259, CWE-327, CWE-330, CWE-400, CWE-502, CWE-605, CWE-703, CWE-78, CWE-79, CWE-80, CWE-838, CWE-89.

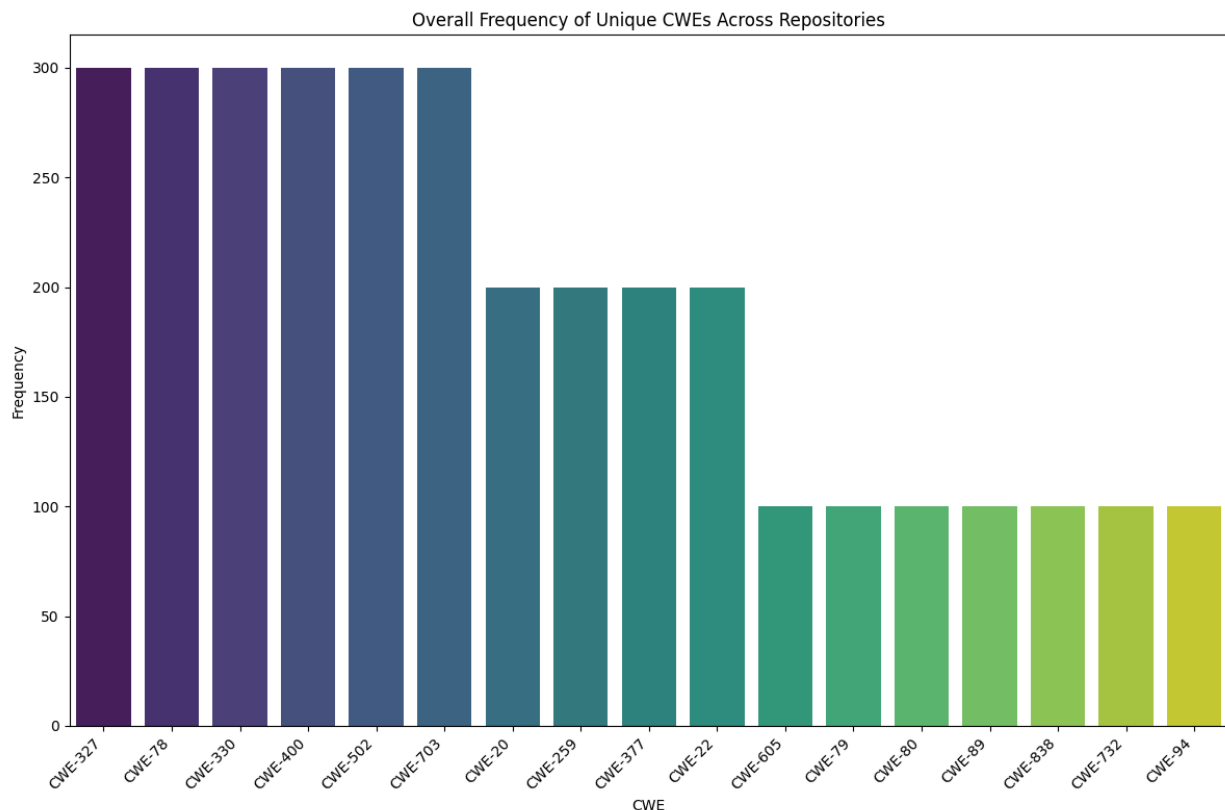
→ All the 100 commits in the **Matplotlib** Repository contains CWE-20, CWE-22, CWE-327, CWE-330, CWE-377, CWE-400, CWE-502, CWE-703, CWE-732, CWE-78.

→ All the commits in the **Scikit-learn** Repository contains CWE-22, CWE-259, CWE-327, CWE-330, CWE-377, CWE-400, CWE-502, CWE-703, CWE-78, CWE-94.

This result is seen because of the short span of 100 commits and similar types of weaknesses in the coding style.

Overall Dataset Level Analysis

If we look at the count of all the unique CWEs across the three repositories, we can estimate which weaknesses and vulnerabilities will most likely be introduced by the developers in the large codebase projects.



Below are some of the most frequently introduced CWEs by the developers while coding for large codebases:

1. **CWE-337 (Missing Critical Cryptographic Step)**: This vulnerability occurs when an application fails to perform an essential cryptographic operation—such as encryption, digital signing, or message authentication—at a point where it is needed to protect sensitive data.
2. **CWE-400 (Uncontrolled Resource Consumption)**: Allows attackers to exhaust system resources (e.g., memory, CPU), often leading to denial of service.
3. **CWE-78 (OS Command Injection)**: Lets attackers run arbitrary operating system commands on the target machine.
4. **CWE-330 (Use of Insufficiently Random Values)**: Involves using predictable or weak random number generation, making tokens and secrets easier for attackers to guess.
5. **CWE-703 (Improper Check or Handling of Exceptional Conditions)**: Fails to correctly handle error or exceptional states, potentially leaving the system in an unsafe condition or disclosing sensitive information.
6. **CWE-502 (Deserialization of Untrusted Data)**: Risks remote code execution or attacks when deserializing data from untrusted sources.

Q: How is a vulnerability fixed?

Ans: In this analysis, we define a vulnerability as fixed when a commit that previously contained a reported security issue no longer shows that issue in its Bandit report. I.e., a vulnerability present in previous commits is no longer present in subsequent commits.

RQ1: High Severity Vulnerabilities

- **Purpose:** Identify when critical (high severity) vulnerabilities are introduced and fixed along the development timeline.

- **Approach:** Analyze commit-level Bandit reports to track high-severity issue counts over sequential commits, marking increases as introductions and decreases as fixes.
- **Results:** There is not much change in the **high**-severity issues in the short span of 100 commits, but we can observe that the **high**-severity issues stayed constant or the developers are resolving them over time (e.g. `matplotlib`)
- **Takeaway:** Critical vulnerabilities are introduced during early development and remediated as the project matures.

RQ2: Patterns by Severity

- **Purpose:** Determine if vulnerabilities of different severities (high, medium, low) follow similar introduction and elimination trends.
- **Approach:** Plot the counts of high, medium, and low severity vulnerabilities against the commit order for each repository.
- **Results:** High-severity issues are fixed promptly, while medium and low-severity issues fluctuate over time. Also, the count of high-severity issues is negligible compared to the low and medium-severity issues.
- **Takeaway:** Remediation efforts are prioritized based on risk, with critical issues addressed faster than lower-risk ones.

RQ3: CWE Coverage Across Repositories

- **Purpose:** Find out which Common Weakness Enumerations (CWEs) are most frequently reported across the repositories.
 - **Approach:** Aggregate and count CWE identifiers from each commit's Bandit report and visualize the top frequencies using a bar chart.
 - **Results:** CWEs such as 400, 78, 330, 703, and 502 consistently appear as the most common vulnerabilities.
 - **Takeaway:** Frequent CWEs indicate recurring security issues.
-