Lab Assignment Report 05

CS202: Software Tools and Techniques for CSE

Roll Number: 22110087
Name: Parth Govale

Contents

1 Introduction, Setup, and Tools **2**
1.1 Overview . . . 2

2 Environment Setup **2**
2.1 System Requirements . . . 2
2.2 Repository Setup . . . 2

3 Methodology and Execution **3**
3.1 Test Suite A Execution . . . 3
3.2 Test Suite A Coverage Analysis . . . 3
3.3 Unit Test Generation (Test Suite B) . . . 5
3.4 Test Suite B Coverage Analysis . . . 6
3.5 Test Effectiveness and Coverage Comparison . . . 8

4 Results and Analysis **8**
4.1 Test Coverage Metrics . . . 8
4.2 Comparison of Test Suites . . . 8

5 Discussion and Conclusion **8**
5.1 Challenges and Reflections . . . 8
5.2 Lessons Learned . . . 9
5.3 Summary . . . 9

6 Appendix **9**
6.1 Tools and Resources . . . 9

1 Introduction, Setup, and Tools

1.1 Overview

This lab report aims to analyze and measure different types of code coverage, including Line (Statement) Coverage, Branch Coverage, and Function Coverage, and generate unit test cases using automated testing tools. The lab also focuses on evaluating the effectiveness of these test cases using various testing metrics.

2 Environment Setup

2.1 System Requirements

- **Operating System:** Ubuntu 20.04 (Windows 11 VM SET-IITGN-VM)
- **Python Version:** 3.10.11
- **Required Tools:**
  - pytest (v8.3.4)
  - pytest-cov (v6.0.0)
  - pytest-func-cov (v0.2.3)
  - coverage (v7.6.12)
  - pynguin (0.40.0)

2.2 Repository Setup

- Clone the `keon/algorithms` repository.

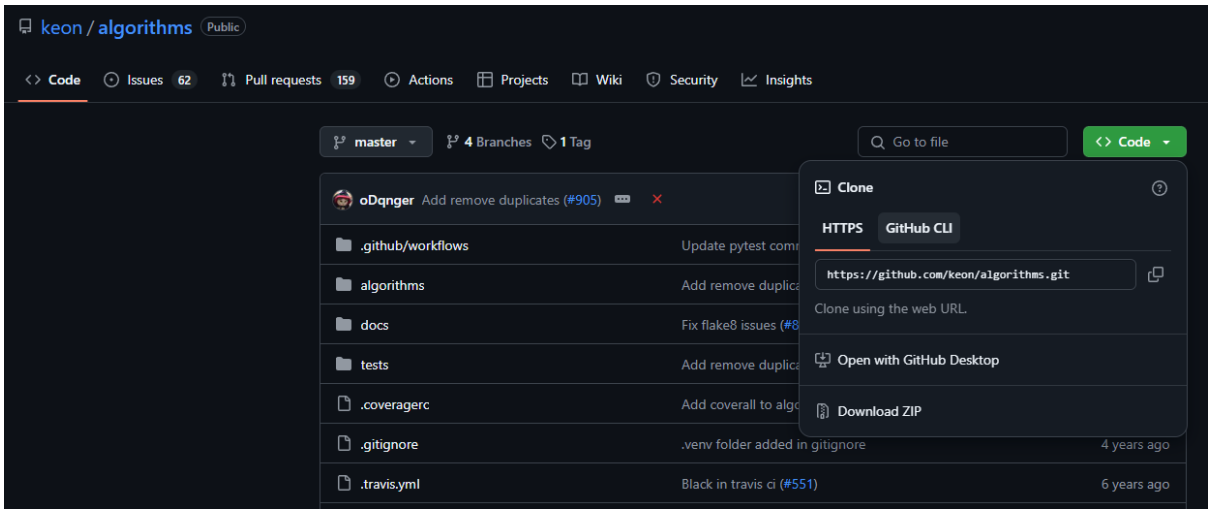


Figure 1: Keon/algorithm GitHub Repository

- Current Commit Hash: `cad4754bc71742c2d6fcbd3b92ae74834d359844`

3 Methodology and Execution

3.1 Test Suite A Execution

- Execute existing test cases (test suite A):

```
python3 -m pytest tests
```

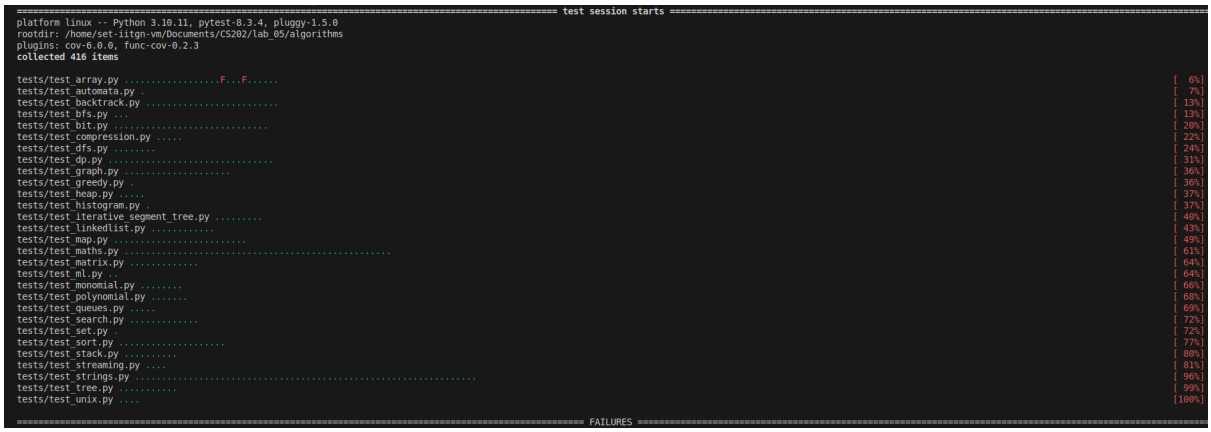


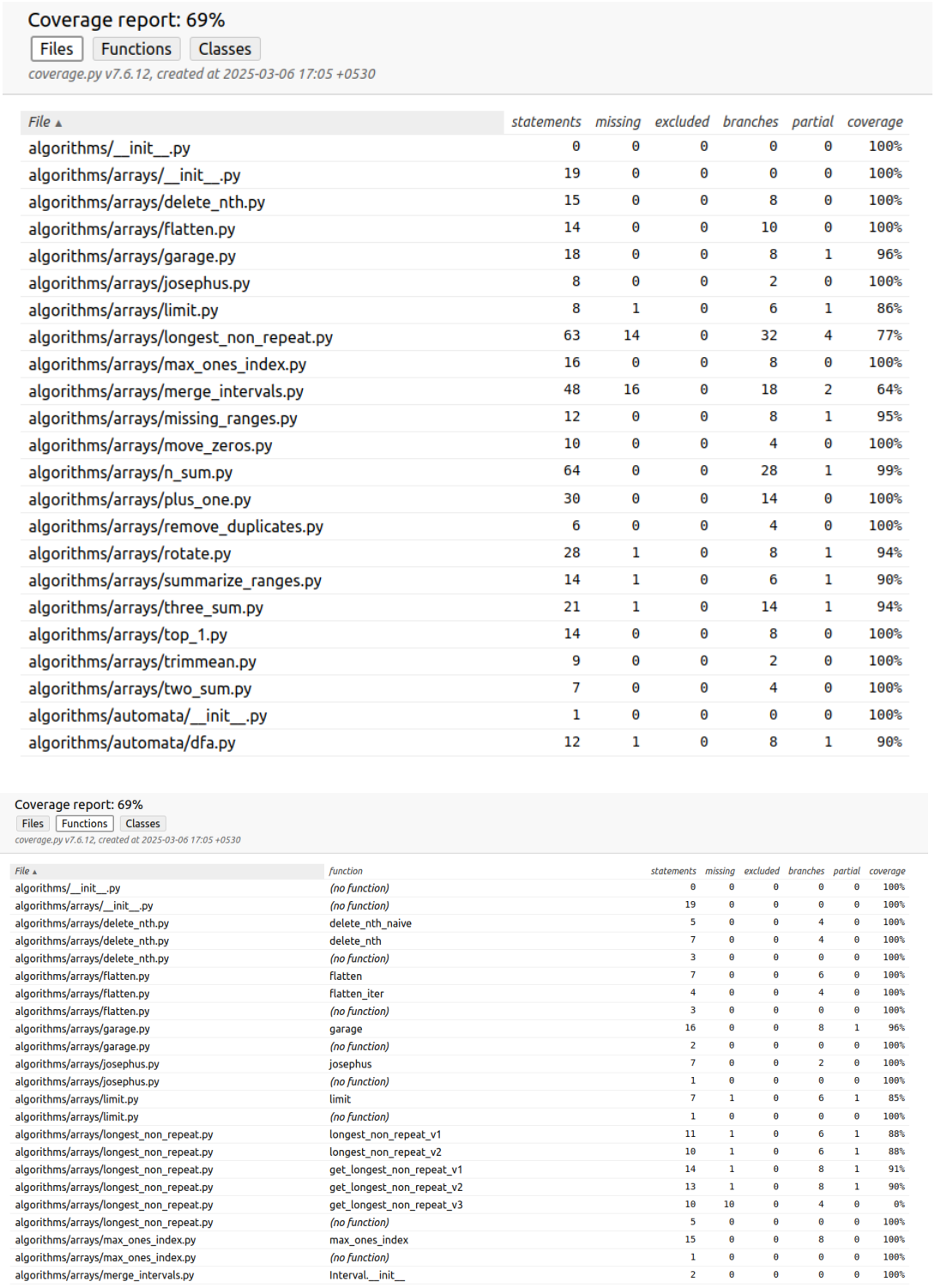
Figure 2: Results of Executing Repository Tests

3.2 Test Suite A Coverage Analysis

- Analyze the coverage for:
  - Line Coverage
  - Branch Coverage
  - Function Coverage

```
coverage run --branch --source=algorithms -m pytest
coverage report -m
coverage html
```

- Recorded the coverage metrics for the test cases:



Coverage report: 69%

Files | Functions | Classes

coverage.py v7.6.12, created at 2025-03-06 17:05 +0530

File ▲	statements	missing	excluded	branches	partial	coverage
algorithms/__init__.py	0	0	0	0	0	100%
algorithms/arrays/__init__.py	19	0	0	0	0	100%
algorithms/arrays/delete_nth.py	15	0	0	8	0	100%
algorithms/arrays/flatten.py	14	0	0	10	0	100%
algorithms/arrays/garage.py	18	0	0	8	1	96%
algorithms/arrays/josephus.py	8	0	0	2	0	100%
algorithms/arrays/limit.py	8	1	0	6	1	86%
algorithms/arrays/longest_non_repeat.py	63	14	0	32	4	77%
algorithms/arrays/max_ones_index.py	16	0	0	8	0	100%
algorithms/arrays/merge_intervals.py	48	16	0	18	2	64%
algorithms/arrays/missing_ranges.py	12	0	0	8	1	95%
algorithms/arrays/move_zeros.py	10	0	0	4	0	100%
algorithms/arrays/n_sum.py	64	0	0	28	1	99%
algorithms/arrays/plus_one.py	30	0	0	14	0	100%
algorithms/arrays/remove_duplicates.py	6	0	0	4	0	100%
algorithms/arrays/rotate.py	28	1	0	8	1	94%
algorithms/arrays/summarize_ranges.py	14	1	0	6	1	90%
algorithms/arrays/three_sum.py	21	1	0	14	1	94%
algorithms/arrays/top_1.py	14	0	0	8	0	100%
algorithms/arrays/trimmean.py	9	0	0	2	0	100%
algorithms/arrays/two_sum.py	7	0	0	4	0	100%
algorithms/automata/__init__.py	1	0	0	0	0	100%
algorithms/automata/dfa.py	12	1	0	8	1	90%

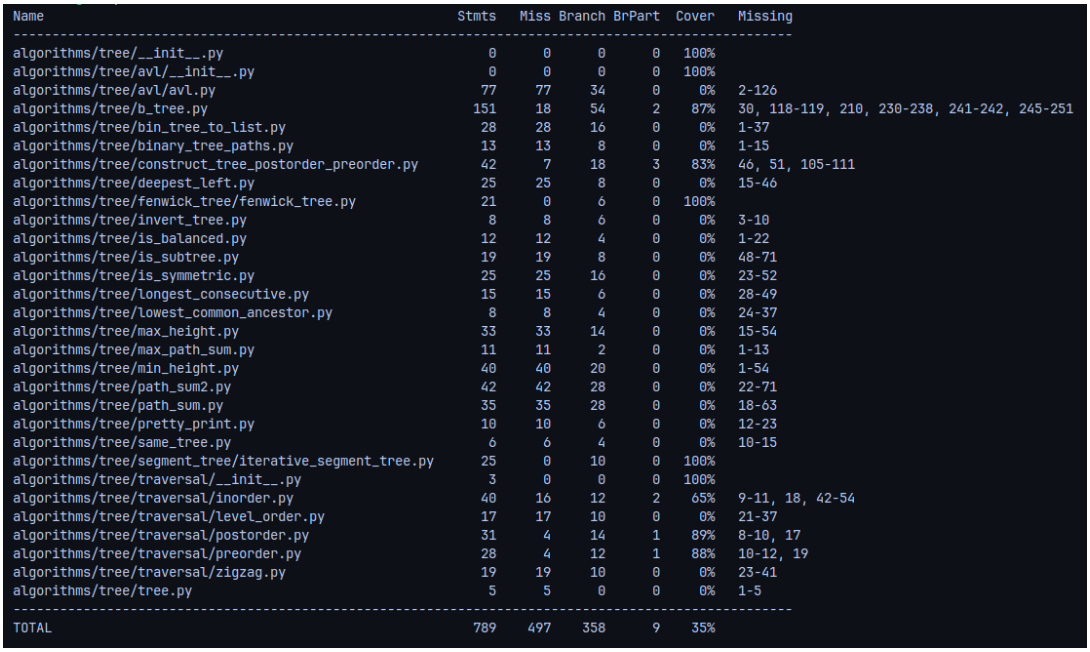
Coverage report: 69%

Files | Functions | Classes

coverage.py v7.6.12, created at 2025-03-06 17:05 +0530

File ▲	function	statements	missing	excluded	branches	partial	coverage
algorithms/__init__.py	(no function)	0	0	0	0	0	100%
algorithms/arrays/__init__.py	(no function)	19	0	0	0	0	100%
algorithms/arrays/delete_nth.py	delete_nth_naive	5	0	0	4	0	100%
algorithms/arrays/delete_nth.py	delete_nth	7	0	0	4	0	100%
algorithms/arrays/delete_nth.py	(no function)	3	0	0	0	0	100%
algorithms/arrays/flatten.py	flatten	7	0	0	6	0	100%
algorithms/arrays/flatten.py	flatten_iter	4	0	0	4	0	100%
algorithms/arrays/flatten.py	(no function)	3	0	0	0	0	100%
algorithms/arrays/garage.py	garage	16	0	0	8	1	96%
algorithms/arrays/garage.py	(no function)	2	0	0	0	0	100%
algorithms/arrays/josephus.py	josephus	7	0	0	2	0	100%
algorithms/arrays/josephus.py	(no function)	1	0	0	0	0	100%
algorithms/arrays/limit.py	limit	7	1	0	6	1	85%
algorithms/arrays/limit.py	(no function)	1	0	0	0	0	100%
algorithms/arrays/longest_non_repeat.py	longest_non_repeat_v1	11	1	0	6	1	88%
algorithms/arrays/longest_non_repeat.py	longest_non_repeat_v2	10	1	0	6	1	88%
algorithms/arrays/longest_non_repeat.py	get_longest_non_repeat_v1	14	1	0	8	1	91%
algorithms/arrays/longest_non_repeat.py	get_longest_non_repeat_v2	13	1	0	8	1	90%
algorithms/arrays/longest_non_repeat.py	get_longest_non_repeat_v3	10	10	0	4	0	0%
algorithms/arrays/longest_non_repeat.py	(no function)	5	0	0	0	0	100%
algorithms/arrays/max_ones_index.py	max_ones_index	15	0	0	8	0	100%
algorithms/arrays/max_ones_index.py	(no function)	1	0	0	0	0	100%
algorithms/arrays/merge_intervals.py	Interval.__init__	2	0	0	0	0	100%

Figure 3: Coverage Results for the Repository



Name	Stmts	Miss	Branch	BrPart	Cover	Missing
algorithms/tree/__init__.py	0	0	0	0	100%	
algorithms/tree/avl/__init__.py	0	0	0	0	100%	
algorithms/tree/avl/avl.py	77	77	34	0	0%	2-126
algorithms/tree/b_tree.py	151	18	54	2	87%	30, 118-119, 210, 230-238, 241-242, 245-251
algorithms/tree/bin_tree_to_list.py	28	28	16	0	0%	1-37
algorithms/tree/binary_tree_paths.py	13	13	8	0	0%	1-15
algorithms/tree/construct_tree_postorder_preorder.py	42	7	18	3	83%	46, 51, 105-111
algorithms/tree/deepest_left.py	25	25	8	0	0%	15-46
algorithms/tree/fenwick_tree/fenwick_tree.py	21	0	6	0	100%	
algorithms/tree/invert_tree.py	8	8	6	0	0%	3-10
algorithms/tree/is_balanced.py	12	12	4	0	0%	1-22
algorithms/tree/is_subtree.py	19	19	8	0	0%	48-71
algorithms/tree/is_symmetric.py	25	25	16	0	0%	23-52
algorithms/tree/longest_consecutive.py	15	15	6	0	0%	28-49
algorithms/tree/lowest_common_ancestor.py	8	8	4	0	0%	24-37
algorithms/tree/max_height.py	33	33	14	0	0%	15-54
algorithms/tree/max_path_sum.py	11	11	2	0	0%	1-13
algorithms/tree/min_height.py	40	40	20	0	0%	1-54
algorithms/tree/path_sum2.py	42	42	28	0	0%	22-71
algorithms/tree/path_sum.py	35	35	28	0	0%	18-63
algorithms/tree/pretty_print.py	10	10	6	0	0%	12-23
algorithms/tree/same_tree.py	6	6	4	0	0%	10-15
algorithms/tree/segment_tree/iterative_segment_tree.py	25	0	10	0	100%	
algorithms/tree/traversal/__init__.py	3	0	0	0	100%	
algorithms/tree/traversal/inorder.py	40	16	12	2	65%	9-11, 18, 42-54
algorithms/tree/traversal/level_order.py	17	17	10	0	0%	21-37
algorithms/tree/traversal/postorder.py	31	4	14	1	89%	8-10, 17
algorithms/tree/traversal/preorder.py	28	4	12	1	88%	10-12, 19
algorithms/tree/traversal/zigzag.py	19	19	10	0	0%	23-41
algorithms/tree/tree.py	5	5	0	0	0%	1-5
TOTAL	789	497	358	9	35%	

Figure 4: Coverage Results for the Repository in tree module

3.3 Unit Test Generation (Test Suite B)

- Generate new test cases using `pynguin`:

```
# Iterate over each Python file in all subdirectories of the module directory
find $MODULE_DIR -type f -name "*.py" | while read -r file; do
    # Skip __init__.py files
    if [[ $(basename "$file") == "__init__.py" ]]; then
        continue
    fi

    # Extract the module name and path
    module_path=$(dirname "$file")
    module_name=$(basename "$file" .py)
    folder_name=$(basename "$module_path")

    # Ensure the output subdirectory exists
    mkdir -p "$OUTPUT_DIR/$folder_name"

    # Run Pynguin
    echo "Running Pynguin for module $module_name in folder $folder_name with
    ↪  module path $module_path"
    pynguin --project-path="$module_path" --module-name="$module_name"
    ↪  --output-path="$OUTPUT_DIR/$folder_name" --export-strategy=PY_TEST
done
```

- Execute `pynguin` generated test cases (test suite B):

```
    python3 -m pytest tests
```

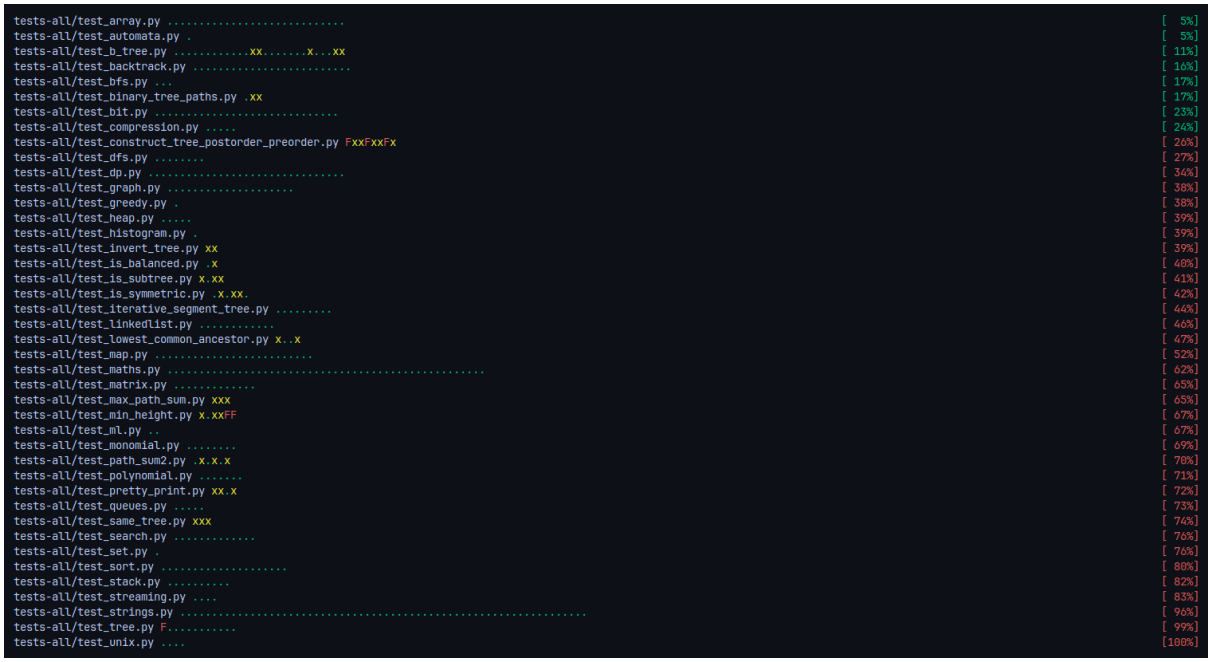


Figure 5: Results of Executing Repository Tests

3.4 Test Suite B Coverage Analysis

- Analyze the coverage for:
  - Line Coverage
  - Branch Coverage
  - Function Coverage

```
coverage run --branch --source=algorithms -m pytest
coverage report -m
coverage html
```

- Recorded the coverage metrics for the test cases:

Coverage report: 70%

Files | Functions | Classes

coverage.py v7.6.12, created at 2025-03-06 17:35 +0530

File ▲	statements	missing	excluded	branches	partial	coverage
algorithms/__init__.py	0	0	0	0	0	100%
algorithms/arrays/__init__.py	19	0	0	0	0	100%
algorithms/arrays/delete_nth.py	15	0	0	8	0	100%
algorithms/arrays/flatten.py	14	0	0	10	0	100%
algorithms/arrays/garage.py	18	0	0	8	1	96%
algorithms/arrays/josephus.py	8	0	0	2	0	100%
algorithms/arrays/limit.py	8	1	0	6	1	86%
algorithms/arrays/longest_non_repeat.py	63	14	0	32	4	77%
algorithms/arrays/max_ones_index.py	16	0	0	8	0	100%
algorithms/arrays/merge_intervals.py	48	16	0	18	2	64%
algorithms/arrays/missing_ranges.py	12	0	0	8	1	95%
algorithms/arrays/move_zeros.py	10	0	0	4	0	100%
algorithms/arrays/n_sum.py	64	0	0	28	1	99%
algorithms/arrays/plus_one.py	30	0	0	14	0	100%
algorithms/arrays/remove_duplicates.py	6	0	0	4	0	100%
algorithms/arrays/rotate.py	28	1	0	8	1	94%
algorithms/arrays/summarize_ranges.py	14	12	0	6	0	10%
algorithms/arrays/three_sum.py	21	1	0	14	1	94%
algorithms/arrays/top_1.py	14	0	0	8	0	100%
algorithms/arrays/trimmean.py	9	0	0	2	0	100%
algorithms/arrays/two_sum.py	7	0	0	4	0	100%
algorithms/automata/__init__.py	1	0	0	0	0	100%
algorithms/automata/dfa.py	12	1	0	8	1	90%
algorithms/backtrack/__init__.py	15	0	0	0	0	100%
algorithms/backtrack/add_operators.py	20	1	0	12	1	94%
algorithms/backtrack/anagram.py	10	0	0	4	0	100%
algorithms/backtrack/array_sum_combinations.py	47	0	0	22	0	100%

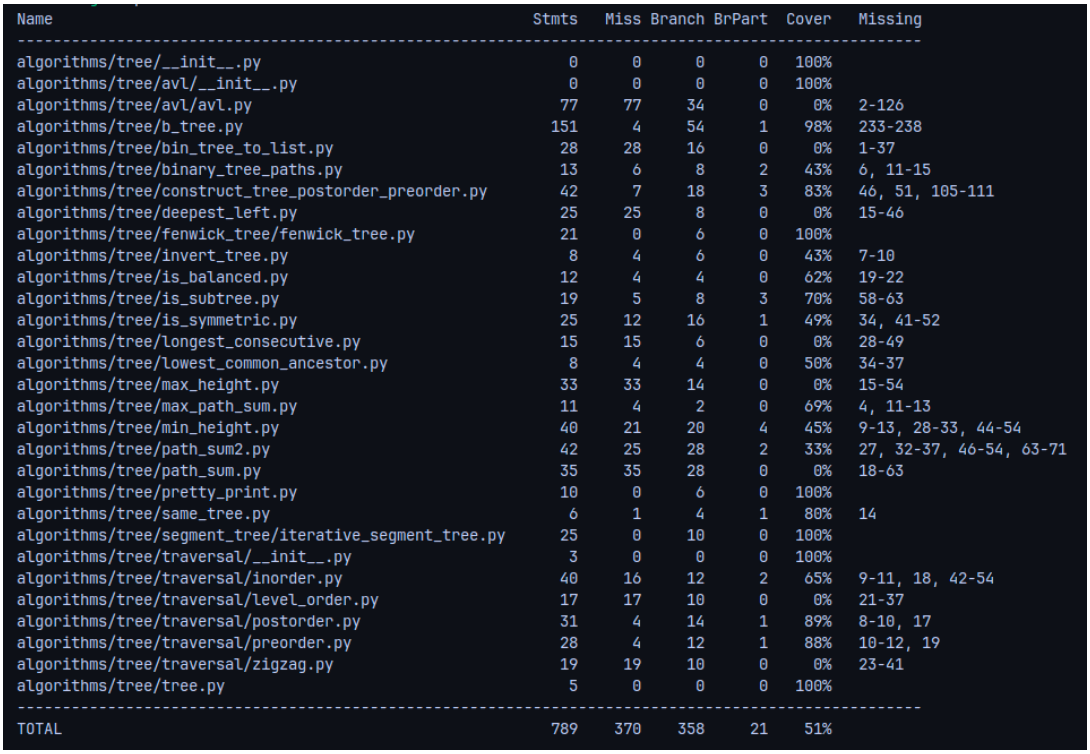
Coverage report: 70%

Files | Functions | Classes

coverage.py v7.6.12, created at 2025-03-06 17:35 +0530

File ▲	function	statements	missing	excluded	branches	partial	coverage
algorithms/__init__.py	*(no function)*	0	0	0	0	0	100%
algorithms/arrays/__init__.py	*(no function)*	19	0	0	0	0	100%
algorithms/arrays/delete_nth.py	delete_nth_naive	5	0	0	4	0	100%
algorithms/arrays/delete_nth.py	delete_nth	7	0	0	4	0	100%
algorithms/arrays/delete_nth.py	*(no function)*	3	0	0	0	0	100%
algorithms/arrays/flatten.py	flatten	7	0	0	6	0	100%
algorithms/arrays/flatten.py	flatten_iter	4	0	0	4	0	100%
algorithms/arrays/flatten.py	*(no function)*	3	0	0	0	0	100%
algorithms/arrays/garage.py	garage	16	0	0	8	1	96%
algorithms/arrays/garage.py	*(no function)*	2	0	0	0	0	100%
algorithms/arrays/josephus.py	josephus	7	0	0	2	0	100%
algorithms/arrays/josephus.py	*(no function)*	1	0	0	0	0	100%
algorithms/arrays/limit.py	limit	7	1	0	6	1	85%
algorithms/arrays/limit.py	*(no function)*	1	0	0	0	0	100%
algorithms/arrays/longest_non_repeat.py	longest_non_repeat_v1	11	1	0	6	1	88%
algorithms/arrays/longest_non_repeat.py	longest_non_repeat_v2	10	1	0	6	1	88%
algorithms/arrays/longest_non_repeat.py	get_longest_non_repeat_v1	14	1	0	8	1	91%
algorithms/arrays/longest_non_repeat.py	get_longest_non_repeat_v2	13	1	0	8	1	90%
algorithms/arrays/longest_non_repeat.py	get_longest_non_repeat_v3	10	10	0	4	0	0%
algorithms/arrays/longest_non_repeat.py	*(no function)*	5	0	0	0	0	100%
algorithms/arrays/max_ones_index.py	max_ones_index	15	0	0	8	0	100%
algorithms/arrays/max_ones_index.py	*(no function)*	1	0	0	0	0	100%
algorithms/arrays/merge_intervals.py	Interval.__init__	2	0	0	0	0	100%
algorithms/arrays/merge_intervals.py	Interval.__repr__	1	1	0	0	0	0%
algorithms/arrays/merge_intervals.py	Interval.__iter__	1	1	0	0	0	0%
algorithms/arrays/merge_intervals.py	Interval.__getitem__	3	3	0	2	0	0%
algorithms/arrays/merge_intervals.py	Interval.__len__	1	1	0	0	0	0%

Figure 6: Coverage Results for the Repository



Name	Stmts	Miss	Branch	BrPart	Cover	Missing
algorithms/tree/__init__.py	0	0	0	0	100%	
algorithms/tree/avl/__init__.py	0	0	0	0	100%	
algorithms/tree/avl/avl.py	77	77	34	0	0%	2-126
algorithms/tree/b_tree.py	151	4	54	1	98%	233-238
algorithms/tree/bin_tree_to_list.py	28	28	16	0	0%	1-37
algorithms/tree/binary_tree_paths.py	13	6	8	2	43%	6, 11-15
algorithms/tree/construct_tree_postorder_preorder.py	42	7	18	3	83%	46, 51, 105-111
algorithms/tree/deepest_left.py	25	25	8	0	0%	15-46
algorithms/tree/fenwick_tree/fenwick_tree.py	21	0	6	0	100%	
algorithms/tree/invert_tree.py	8	4	6	0	43%	7-10
algorithms/tree/is_balanced.py	12	4	4	0	62%	19-22
algorithms/tree/is_subtree.py	19	5	8	3	70%	58-63
algorithms/tree/is_symmetric.py	25	12	16	1	49%	34, 41-52
algorithms/tree/longest_consecutive.py	15	15	6	0	0%	28-49
algorithms/tree/lowest_common_ancestor.py	8	4	4	0	50%	34-37
algorithms/tree/max_height.py	33	33	14	0	0%	15-54
algorithms/tree/max_path_sum.py	11	4	2	0	69%	4, 11-13
algorithms/tree/min_height.py	40	21	20	4	45%	9-13, 28-33, 44-54
algorithms/tree/path_sum2.py	42	25	28	2	33%	27, 32-37, 46-54, 63-71
algorithms/tree/path_sum.py	35	35	28	0	0%	18-63
algorithms/tree/pretty_print.py	10	0	6	0	100%	
algorithms/tree/same_tree.py	6	1	4	1	80%	14
algorithms/tree/segment_tree/iterative_segment_tree.py	25	0	10	0	100%	
algorithms/tree/traversal/__init__.py	3	0	0	0	100%	
algorithms/tree/traversal/inorder.py	40	16	12	2	65%	9-11, 18, 42-54
algorithms/tree/traversal/level_order.py	17	17	10	0	0%	21-37
algorithms/tree/traversal/postorder.py	31	4	14	1	89%	8-10, 17
algorithms/tree/traversal/preorder.py	28	4	12	1	88%	10-12, 19
algorithms/tree/traversal/zigzag.py	19	19	10	0	0%	23-41
algorithms/tree/tree.py	5	0	0	0	100%	
TOTAL	789	370	358	21	51%	

Figure 7: Coverage Results for the Repository in tree module

We can observe that for the module tree the coverage has increased from 35% to 51%

3.5 Test Effectiveness and Coverage Comparison

- We can observe that on generating new test cases using pynguin, we were able to increase the coverage of the repository by 1 % and the coverage of tree module by 16 %
- The new test cases also revealed failures in the current implementation and also the addition of xfailure test cases have provided more insights into improving the code quality.

4 Results and Analysis

4.1 Test Coverage Metrics

The test coverage reports are available here: Coverage HTML Report and Test Suites

4.2 Comparison of Test Suites

Test suite B is able to provide better coverage as compared to the repository test suite (Test suite A). The test suite B also includes test cases which expect the function to fail while running these tests which was missing in Test suite A.

5 Discussion and Conclusion

5.1 Challenges and Reflections

The challenges faced included the initial setup of pytest and generating the coverage report. Additionally, running Pynguin to generate unit test cases for each module took a significant amount of time, which slowed down the entire process considerably.

5.2 Lessons Learned

The importance of understanding various methods of code coverage to evaluate how thoroughly a program is tested. Gained practical experience with tools like pytest and coverage.py, which helped in assessing the extent of automated testing. While generating unit tests with Pynguin was effective in improving coverage, it was time-consuming for larger modules. Analyzing coverage reports allowed to identify areas of the code that were not covered by tests.

5.3 Summary

This lab helped us understand the importance of automated testing and coverage tools to improve the testing of code which is helpful while maintaining large repositories. And also the use of Pynguin which generates automated test cases for our code to validate.

6 Appendix

6.1 Tools and Resources

Provide links to any resources or tools that were used for this lab, such as:

- **Pynguin:** Pynguin
- **Coverage:** Coverage Documentation
- **pytest:** pytest Documentation
- **pytest-cov:** pytest-cov GitHub
- **pytest-func-cov:** pytest-func-cov PyPI

Lab Assignment Report 06

CS202: Software Tools and Techniques for CSE

Roll Number: 22110087
Name: Parth Govale

Contents

1 Introduction, Setup, and Tools **2**
1.1 Overview . . . 2

2 Environment Setup **2**
2.1 System Requirements . . . 2
2.2 Repository Setup . . . 2

3 Methodology and Execution **2**
3.1 Sequential Test Execution . . . 2
3.2 Parallel Test Execution . . . 3
3.3 Performance Metrics and Analysis . . . 6

4 Results and Analysis **8**
4.1 Test Execution Times and Speedup . . . 8

5 Discussion and Conclusion **9**
5.1 Challenges and Reflections . . . 9
5.2 Lessons Learned . . . 9
5.3 Summary . . . 9

6 Appendix **9**
6.1 Tools and Resources . . . 9

1 Introduction, Setup, and Tools

1.1 Overview

This lab focuses on exploring the challenges of test parallelization in Python. The primary objectives are to understand different parallelization modes available in `pytest-xdist` and `pytest-run-parallel`, identify flaky tests that emerge under parallel execution, and evaluate the overall speedup achieved compared to sequential test runs.

2 Environment Setup

2.1 System Requirements

- **Operating System:** Ubuntu 20.04 (Windows 11 VM SET-IITGN-VM)
- **Python Version:** 3.10.11
- **Required Tools:**
  - `pytest` (v8.3.4)
  - `pytest-xdist` (v3.6.1)
  - `pytest-run-parallel` (v0.3.1)

2.2 Repository Setup

- Clone the `keon/algorithms` repository.

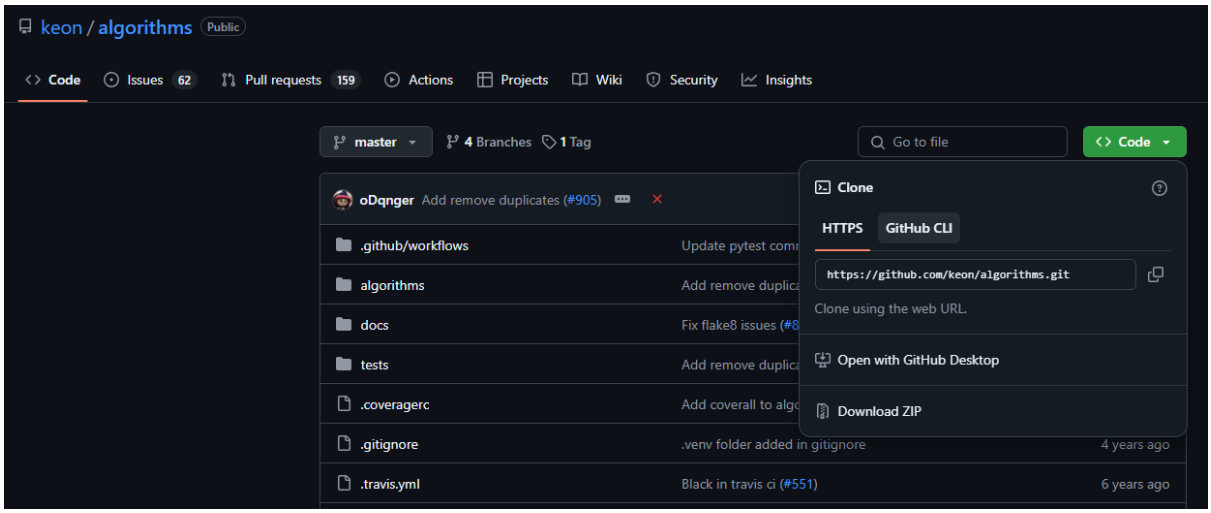


Figure 1: Keon/algorithm GitHub Repository

- Current Commit Hash: `cad4754bc71742c2d6fcbd3b92ae74834d359844`

3 Methodology and Execution

3.1 Sequential Test Execution

- Execute the full test suite sequentially ten times to identify any failing or flaky tests. This is the `pytest.ini`:

```
[pytest]
testpaths =
    tests
```

- Removed tests that are non-deterministic or failing.

```
class TestRemoveDuplicate(unittest.TestCase):

    def test_remove_duplicates(self):
        # Test case with mixed data types
        self.assertListEqual(remove_duplicates([1, 1, "hello", "hello", True,
        ↪  False, False]),
                             [1, "hello", True, False])

        # Test case with mixed data types
        self.assertListEqual(remove_duplicates([1, "hello", True, False]),
                             [1, "hello", True, False])

class TestSummaryRanges(unittest.TestCase):

    def test_summarize_ranges(self):

        self.assertListEqual(summarize_ranges([0, 1, 2, 4, 5, 7]),
                             [(0, 2), (4, 5), (7, 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)])
```

- Once the suite is stable, Average sequential execution time T_{seq} was 4.07483 seconds.

3.2 Parallel Test Execution

- Configure parallel test execution using bash script:

```
# Define configurations:
# We use two options for worker/thread counts: "auto" and "1".
# And two pytest-xdist distribution modes: "load" and "no".
for dist_mode in load no; do
    for config in "auto,auto" "auto,1" "1,auto" "1,1"; do
        IFS=',' read workers threads <<< "$config"
        for rep in {1..3}; do
            echo "Parallel run: --dist $dist_mode, -n $workers,
            ↪  --parallel-threads $threads, repetition $rep"

            ↪  log_file="$LOG_DIR/par_${dist_mode}_${workers}_${threads}_rep${rep}.log"
            start=$(date +%s.%N)
            pytest -n "$workers" --dist "$dist_mode" --parallel-threads
            ↪  "$threads" > "$log_file" 2>&1
            end=$(date +%s.%N)
            elapsed=$(echo "$end - $start" | bc)

            # Count number of failures (simple grep; may need adjustment)
            fail_count=$(grep -c "FAILED" "$log_file" || true)
            # Extract names of failed tests (concatenate them using ';' as
            ↪  separator)
```

```
            failed_tests=$(grep "FAILED" "$log_file" | tr '\n' ';' | sed
            ↪  's/;$//')

            # Write the result as a CSV row
            echo
            ↪  "$dist_mode,$workers,$threads,$rep,$elapsed,$fail_count,\"$failed_tests\""
            ↪  >> "$RESULTS_FILE"
        done
    done
done
```

- For each configuration, perform three repetitions and denote the average execution time as T_{par}.

Mode	Workers	Threads	Rep	T_{par} (sec)	Fail_Count
load	auto	auto	1	236.374712367	4
load	auto	auto	2	231.088542135	4
load	auto	auto	3	235.948705086	3
load	auto	1	1	3.748244486	2
load	auto	1	2	3.551589012	2
load	auto	1	3	3.602219318	2
load	1	auto	1	413.817465372	6
load	1	auto	2	401.262074082	5
load	1	auto	3	402.852980239	6
load	1	1	1	4.570110768	0
load	1	1	2	4.467423525	0
load	1	1	3	4.554395546	0
no	auto	auto	1	235.421363010	3
no	auto	auto	2	229.337739487	3
no	auto	auto	3	229.449731435	3
no	auto	1	1	3.655476188	2
no	auto	1	2	3.657080345	2
no	auto	1	3	3.610869259	2
no	1	auto	1	404.875770944	5
no	1	auto	2	408.963581132	6
no	1	auto	3	397.098718692	5
no	1	1	1	4.465625802	0
no	1	1	2	4.469855731	0
no	1	1	3	4.443898522	0

Table 1: Parallel Execution Results.

- Test failures that occur due to parallel execution (i.e., flaky tests that were not present during sequential runs).

Mode	Workers	Threads	Rep	Fail_Count	Failed Tests
load	auto	auto	1	4	• test_heap.py::TestBinaryHeap::test_insert • test_linkedlist.py::TestSuite::test_is_palindrome • test_heap.py::TestBinaryHeap::test_remove_min • test_compression.py::TestHuffmanCoding::test_huffman_coding
load	auto	auto	2	4	• test_linkedlist.py::TestSuite::test_is_palindrome • test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_compression.py::TestHuffmanCoding::test_huffman_coding
load	auto	auto	3	3	• test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_linkedlist.py::TestSuite::test_is_palindrome
load	auto	1	1	2	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges
load	auto	1	2	2	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges
load	auto	1	3	2	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges
load	1	auto	1	6	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges • test_compression.py::TestHuffmanCoding::test_huffman_coding • test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_linkedlist.py::TestSuite::test_is_palindrome
load	1	auto	2	5	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges • test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_linkedlist.py::TestSuite::test_is_palindrome
load	1	auto	3	6	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges • test_compression.py::TestHuffmanCoding::test_huffman_coding • test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_linkedlist.py::TestSuite::test_is_palindrome

Table 2: Detailed Failed Tests for Specific Configurations

Mode	Workers	Threads	Rep	Fail_Count	Failed Tests
no	auto	auto	1	3	• test_heap.py::TestBinaryHeap::test_insert • test_linkedlist.py::TestSuite::test_is_palindrome • test_heap.py::TestBinaryHeap::test_remove_min
no	auto	auto	2	3	• test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_linkedlist.py::TestSuite::test_is_palindrome
no	auto	auto	3	3	• test_heap.py::TestBinaryHeap::test_insert • test_linkedlist.py::TestSuite::test_is_palindrome • test_heap.py::TestBinaryHeap::test_remove_min
no	auto	1	1	2	• test_array.py::TestSummaryRanges::test_summarize_ranges • test_array.py::TestRemoveDuplicate::test_remove_duplicates
no	auto	1	2	2	• test_array.py::TestSummaryRanges::test_summarize_ranges • test_array.py::TestRemoveDuplicate::test_remove_duplicates
no	auto	1	3	2	• test_array.py::TestSummaryRanges::test_summarize_ranges • test_array.py::TestRemoveDuplicate::test_remove_duplicates
no	1	auto	1	5	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges • test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_linkedlist.py::TestSuite::test_is_palindrome
no	1	auto	2	6	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges • test_compression.py::TestHuffmanCoding::test_huffman_coding • test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_linkedlist.py::TestSuite::test_is_palindrome
no	1	auto	3	5	• test_array.py::TestRemoveDuplicate::test_remove_duplicates • test_array.py::TestSummaryRanges::test_summarize_ranges • test_heap.py::TestBinaryHeap::test_insert • test_heap.py::TestBinaryHeap::test_remove_min • test_linkedlist.py::TestSuite::test_is_palindrome

Table 3: Detailed Failed Tests for Specific Configurations

3.3 Performance Metrics and Analysis

- **Speedup Analysis:** The speedup factor is defined as

$$S = \frac{T_{seq}}{T_{par}},$$

where T_{seq} is the sequential execution time and T_{par} is the parallel execution time under a given configuration. In our experiments, the sequential execution time was measured as:

$$T_{seq} = 4.07483 \text{ sec.}$$

The following parallel configurations were tested (with the average T_{par} calculated over three repetitions):

Load Mode:

- **auto/auto:**

$$\bar{T}_{par} \approx \frac{236.37 + 231.09 + 235.95}{3} \approx 234.14 \text{ sec},$$

yielding

$$S \approx \frac{4.07483}{234.14} \approx 0.0174.$$

- **auto/1:**

$$\bar{T}_{par} \approx \frac{3.748 + 3.552 + 3.602}{3} \approx 3.634 \text{ sec},$$

yielding

$$S \approx \frac{4.07483}{3.634} \approx 1.1216.$$

- **1/auto:**

$$\bar{T}_{par} \approx \frac{413.82 + 401.26 + 402.85}{3} \approx 405.31 \text{ sec},$$

yielding

$$S \approx \frac{4.07483}{405.31} \approx 0.0101.$$

- **1/1:**

$$\bar{T}_{par} \approx \frac{4.570 + 4.467 + 4.554}{3} \approx 4.530 \text{ sec},$$

yielding

$$S \approx \frac{4.07483}{4.530} \approx 0.900.$$

No Mode:

- **auto/auto:**

$$\bar{T}_{par} \approx \frac{235.42 + 229.34 + 229.45}{3} \approx 231.07 \text{ sec},$$

yielding

$$S \approx \frac{4.07483}{231.07} \approx 0.0176.$$

- **auto/1:**

$$\bar{T}_{par} \approx \frac{3.655 + 3.657 + 3.611}{3} \approx 3.641 \text{ sec},$$

yielding

$$S \approx \frac{4.07483}{3.641} \approx 1.119.$$

- **1/auto:**

$$\bar{T}_{par} \approx \frac{404.88 + 408.96 + 397.10}{3} \approx 403.31 \text{ sec},$$

yielding

$$S \approx \frac{4.07483}{403.31} \approx 0.0101.$$

- **1/1:**

$$\bar{T}_{par} \approx \frac{4.466 + 4.470 + 4.444}{3} \approx 4.460 \text{ sec},$$

yielding

$$S \approx \frac{4.07483}{4.460} \approx 0.914.$$

Overall Analysis:

Configurations with both workers and threads set to `auto` (i.e., auto/auto) resulted in extremely high T_{par} (over 230 sec), yielding a very low speedup factor (approximately 0.017), indicating that the overhead in these settings dramatically slows down the test execution.

The best performance improvement is observed in the `auto/1` configuration, where T_{par} is approximately 3.63–3.64 sec, leading to a speedup factor slightly above 1 (around 1.12). This indicates that under this configuration, parallel execution is modestly faster than the sequential run.

Configurations with fixed 1 worker (1/auto) show very high T_{par} (around 405 sec), and those with both fixed (1/1) yield T_{par} values near 4.5 sec, which are either close to or slightly worse than the sequential execution time.

- **Flaky Tests Analysis:** The parallel executions, particularly with the `auto` settings for workers and threads, revealed a number of flaky tests. The configurations with `load, auto, auto` consistently showed failures in tests related to heap and linked list implementations. For example, the configuration with `load, auto, auto` for 1 replication showed failures in 4 tests:

```
test_heap.py::TestBinaryHeap::test_insert
test_linkedlist.py::TestSuite::test_is_palindrome
test_heap.py::TestBinaryHeap::test_remove_min
test_compression.py::TestHuffmanCoding::test_huffman_coding
```

  This non-deterministic behavior suggests that these tests may not be properly isolated for concurrent execution. Similarly, when the configuration changed to 2 replications, the failures remained in the same tests. A similar pattern was observed for other configurations like `no, auto, auto`.

- **Shared Resources and Thread-Safety:** Analysis of the failures indicates that several issues may stem from shared resource conflicts and timing discrepancies. Tests involving data structures such as binary heaps and linked lists were particularly affected. For instance, in the configuration `load, auto, auto` with 1 replication, failures occurred in 4 tests:

```
test_heap.py::TestBinaryHeap::test_insert
test_linkedlist.py::TestSuite::test_is_palindrome
test_heap.py::TestBinaryHeap::test_remove_min
test_compression.py::TestHuffmanCoding::test_huffman_coding
```

4 Results and Analysis

4.1 Test Execution Times and Speedup

Configuration	Average Time (sec)	Speedup
Sequential	4.07	1.00
load, auto, auto	234.47	0.0174
load, auto, 1	3.63	1.12
load, 1, auto	405.98	0.0100
load, 1, 1	4.53	0.90
no, auto, auto	231.40	0.0176
no, auto, 1	3.64	1.12
no, 1, auto	403.65	0.0101
no, 1, 1	4.46	0.91

Table 4: Average Execution Times and Speedup Ratios

The test executions with various configurations are available here: Sequential and Parallel Test case execution

5 Discussion and Conclusion

5.1 Challenges and Reflections

Parallel test execution introduced challenges such as non-deterministic behavior in some tests and issues related to shared resource access. These challenges highlight the need for careful test design when leveraging parallelism.

5.2 Lessons Learned

Understanding the intricacies of both process-level and thread-level parallelization is crucial. Eliminating flaky tests is essential for reliable parallel test execution. Proper configuration and resource management can significantly improve test execution times.

5.3 Summary

This lab provided hands-on experience with Python test parallelization. Through sequential and parallel executions, the report highlights the benefits of parallel testing in reducing execution time while also exposing potential downsides such as flaky tests and inconsistent outcomes.

6 Appendix

6.1 Tools and Resources

- **pytest:** pytest Documentation
- **pytest-xdist:** pytest-xdist Documentation
- **pytest-run-parallel:** pytest-run-parallel PyPI

Lab Assignment Report 07 & 08

CS202: Software Tools and Techniques for CSE

Roll Number: 22110087
Name: Parth Govale

Contents

1 Introduction, Setup, and Tools **2**
1.1 Overview . . . 2

2 Environment Setup **2**
2.1 System Requirements . . . 2
2.2 Repository Setup . . . 2

3 Methodology and Execution **5**
3.1 Execution of Bandit . . . 5
3.2 Individual Repository-Level Analyses . . . 6
3.3 Overall Dataset-Level Analyses (Research Questions) . . . 17

4 Results and Analysis **18**
4.1 Individual Repository Analysis . . . 18

5 Discussion and Conclusion **20**
5.1 Challenges and Reflections . . . 20
5.2 Lessons Learned . . . 20
5.3 Summary . . . 20

6 Appendix **21**
6.1 Tools and Resources . . . 21

1 Introduction, Setup, and Tools

1.1 Overview

This combined lab report covers Lab Assignments 07 and 08, conducted on 20th and 27th February 2025. The primary focus of these labs is to perform a vulnerability analysis on open-source software repositories using `bandit`, a static code analysis tool designed to detect security vulnerabilities in Python code.

2 Environment Setup

2.1 System Requirements

- **Operating System:** Ubuntu 20.04 (Windows 11 VM SET-IITGN-VM)
- **Python Version:** 3.10.11
- **Required Tool:** `bandit` (v1.8.3)

2.2 Repository Setup

1. Used the SEART GitHub Search Engine to identify suitable repositories.
2. Selection criteria included:

Number of commits: $\geq 7,000$	Number of stars: $\geq 5,000$
Number of issues: ≥ 100	Number of watchers: ≥ 500
Number of branches: ≥ 3	Number of forks: $\geq 1,000$
Number of code lines: $\geq 50,000$	Number of releases: ≥ 5

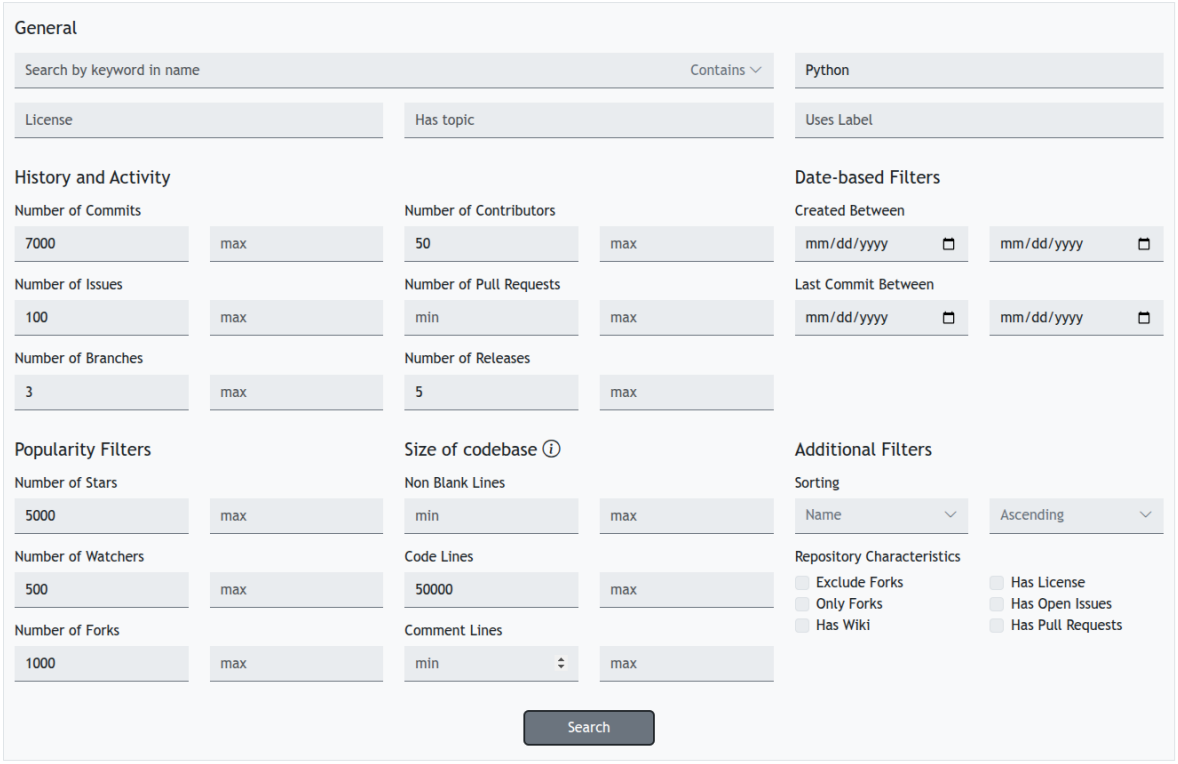


Figure 1: Search parameters in SEART Github Search Engine

3. Selected Repositories:

    1. **Spacy** (GitHub: `https://github.com/explosion/spacy`).

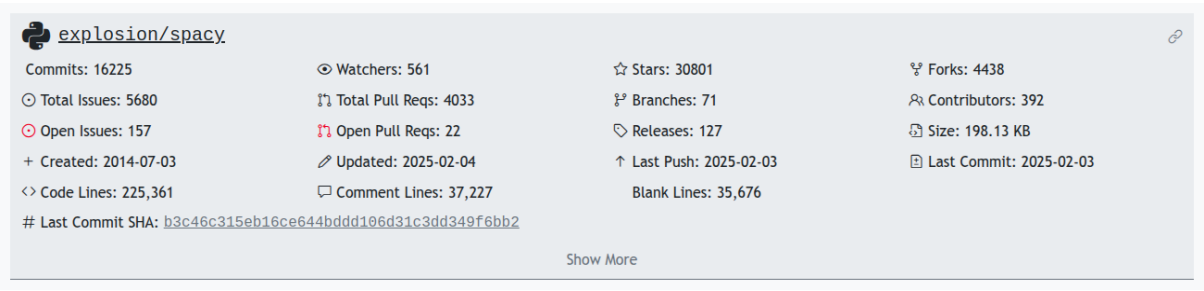


Figure 2: Spacy GitHub Repository

    2. **Transformers** (GitHub: `https://github.com/huggingface/transformers`).

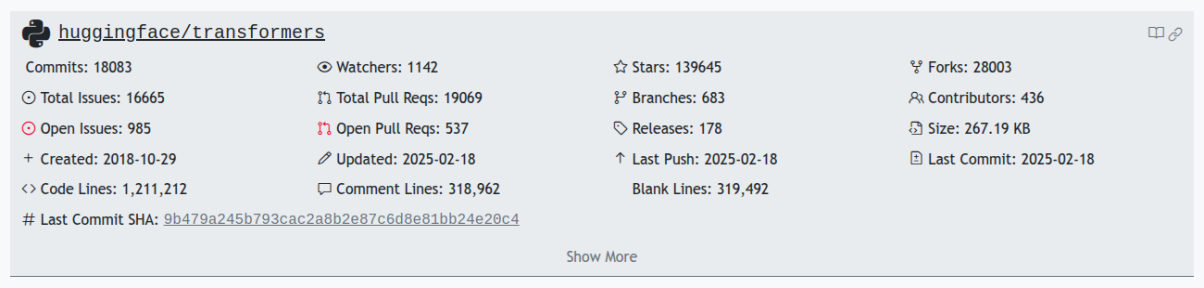


Figure 3: Transformers GitHub Repository

    3. **Keras** (GitHub: `https://github.com/keras-team/keras`).

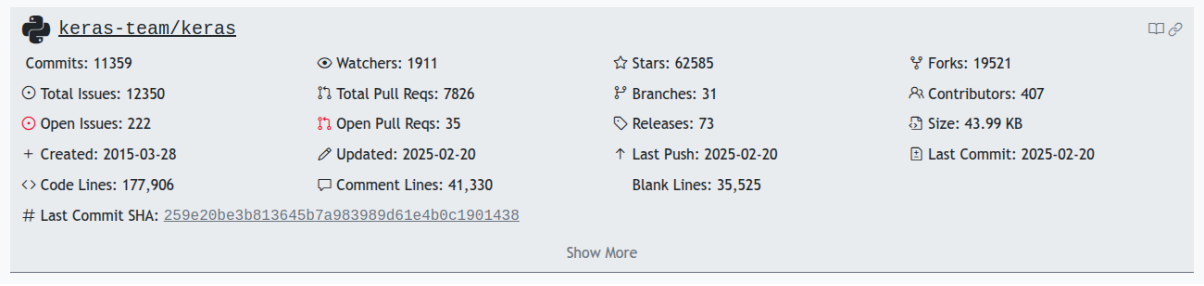


Figure 4: Keras GitHub Repository

    4. **Matplotlib** (GitHub: `https://github.com/matplotlib/matplotlib`).

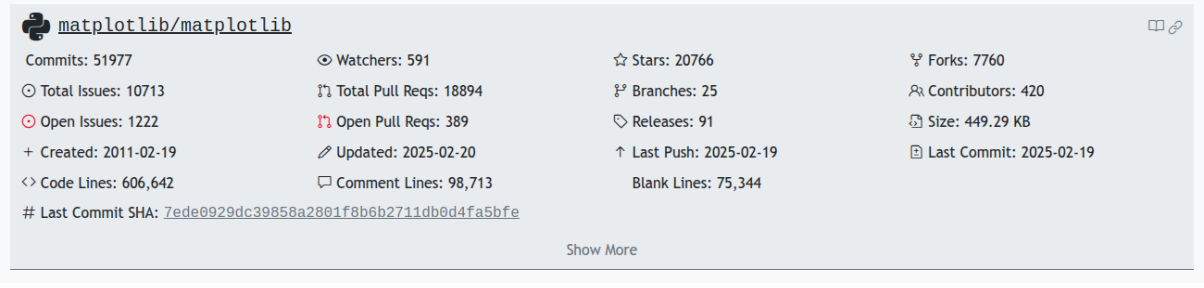


Figure 5: Matplotlib GitHub Repository

5. **Numpy** (GitHub: `https://github.com/numpy/numpy`).

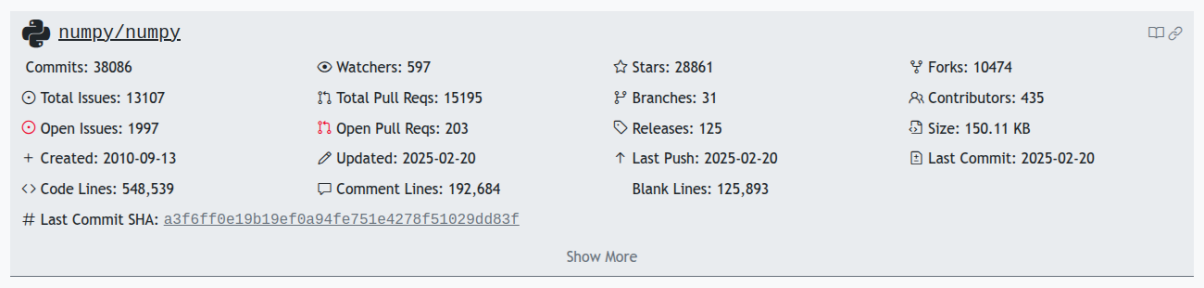


Figure 6: Numpy GitHub Repository

6. **Pandas** (GitHub: `https://github.com/pandas-dev/pandas`).

Figure 7: Pandas GitHub Repository

7. **Pillow** (GitHub: `https://github.com/python-pillow/pillow`).

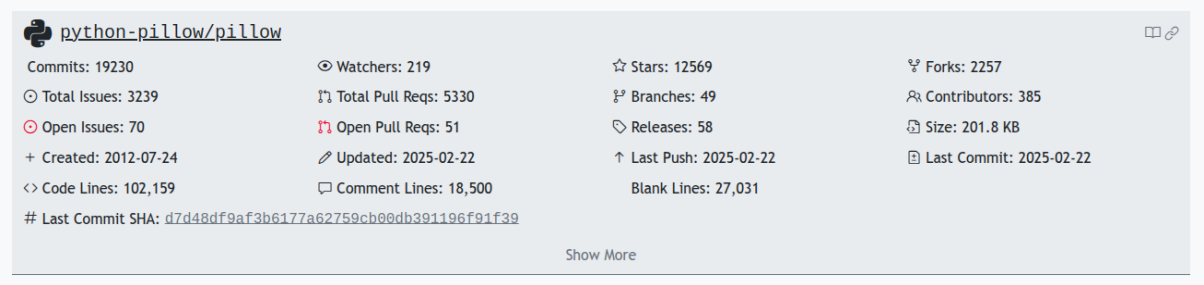


Figure 8: Pillow GitHub Repository

8. **Scikit-learn** (GitHub: `https://github.com/scikit-learn/scikit-learn`).

Figure 9: Scikit-learn GitHub Repository

9. **Sqlmap** (GitHub: `https://github.com/sqlmapproject/sqlmap`).

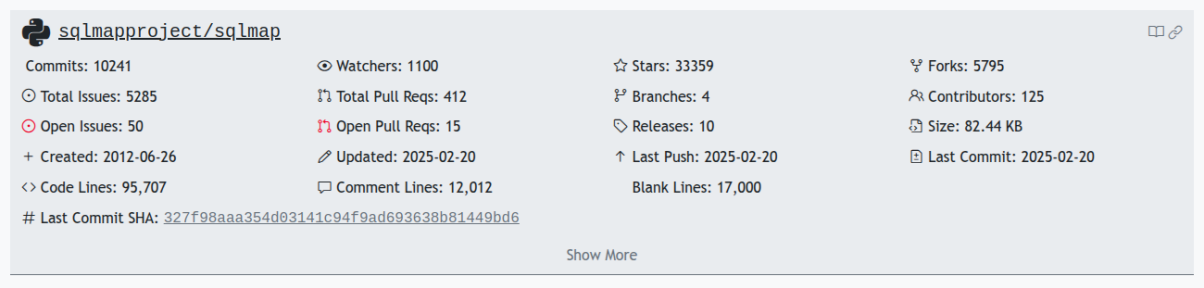


Figure 10: Sqlmap GitHub Repository

10. **Tensorflow Models** (GitHub: `https://github.com/tensorflow/models`).

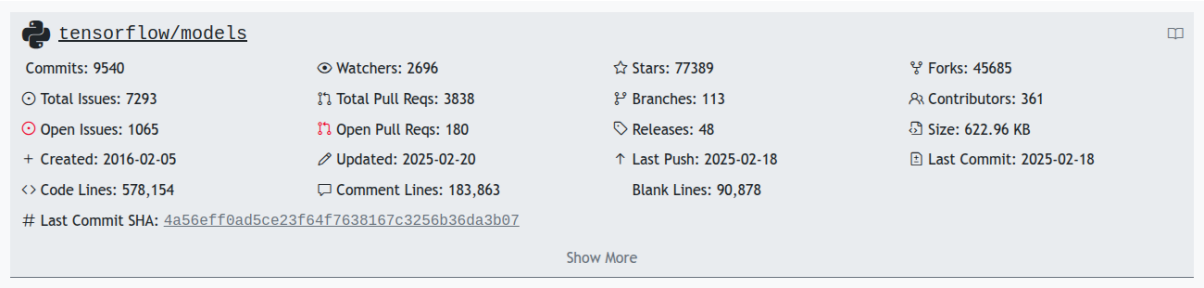


Figure 11: Tensorflow Models GitHub Repository

3 Methodology and Execution

3.1 Execution of Bandit

- For each repository, `bandit` was executed on the last 500 non-merge commits to the main branch.
- This is the execution script:

```
# Loop through each repository
for repo in "${repos[@]}"; do
    echo "Processing repository: $repo"

    # Navigate to repository
    cd "$repo" || { echo "Failed to enter $repo"; continue; }

    rm -rf bandit_reports
    mkdir -p bandit_reports

    # Get last 500 non-merge commits
    git log --first-parent --no-merges -n 500 --pretty=format:"%H" | awk '1;
    ↪  END {print ""}' > commits.txt

    # Process commits with sequential numbering
    counter=1
    while read commit; do
        # Skip empty lines and invalid hashes
        [ -z "$commit" ] && continue
        [ ${#commit} -ne 40 ] && continue

        # Force clean checkout
```

```
        git checkout -f "$commit"

        # Format counter with leading zeros
        printf -v seq_num "%03d" $counter

        # Generate filename
        bandit -r . -f json -o
        ↪  "bandit_reports/bandit_${seq_num}_${commit:0:7}.json"

        ((counter++))
    done < commits.txt

    # Clean up
    cd ..

    echo "Completed processing: $repo"
    echo "-----------------------------------"
done
```

- This process generated reports detailing security vulnerabilities across the Python files.

3.2 Individual Repository-Level Analyses

For every repository, the following metrics were recorded per commit:

```
# Aggregate confidence values (sum across files)
high_conf = sum(file_metrics.get("CONFIDENCE.HIGH", 0) for file_metrics in
↪  metrics.values())
med_conf  = sum(file_metrics.get("CONFIDENCE.MEDIUM", 0) for file_metrics in
↪  metrics.values())
low_conf  = sum(file_metrics.get("CONFIDENCE.LOW", 0) for file_metrics in
↪  metrics.values())

# For severity, compute file-level values and then min, max, avg, and sum.
high_values = [file_metrics.get("SEVERITY.HIGH", 0) for file_metrics in
↪  metrics.values()]
med_values  = [file_metrics.get("SEVERITY.MEDIUM", 0) for file_metrics in
↪  metrics.values()]
low_values  = [file_metrics.get("SEVERITY.LOW", 0) for file_metrics in
↪  metrics.values()]

high_sev_sum = sum(high_values)
med_sev_sum  = sum(med_values)
low_sev_sum  = sum(low_values)

high_sev_min = min(high_values) if high_values else 0
high_sev_max = max(high_values) if high_values else 0
high_sev_avg = sum(high_values)/len(high_values) if high_values else 0

med_sev_min = min(med_values) if med_values else 0
med_sev_max = max(med_values) if med_values else 0
med_sev_avg = sum(med_values)/len(med_values) if med_values else 0

low_sev_min = min(low_values) if low_values else 0
low_sev_max = max(low_values) if low_values else 0
low_sev_avg = sum(low_values)/len(low_values) if low_values else 0
```

1. **Spacy** (GitHub: `https://github.com/explosion/spacy`).

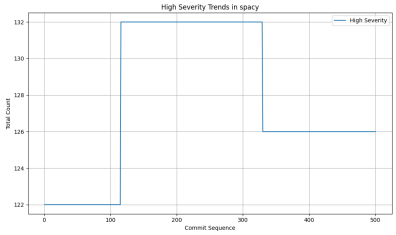


Figure 12: High Severity Trends

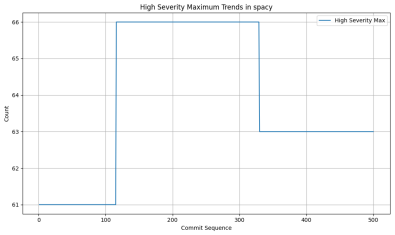


Figure 13: High Severity Max Trends

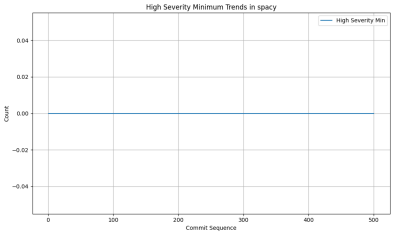


Figure 14: High Severity Min Trends

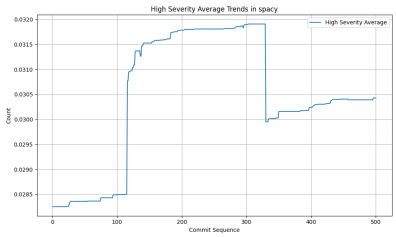


Figure 15: High Severity Avg Trends

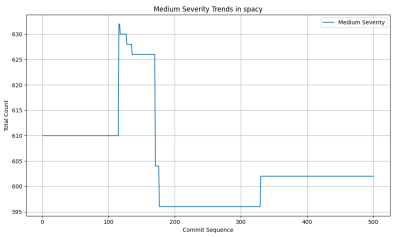


Figure 16: Medium Severity Trends

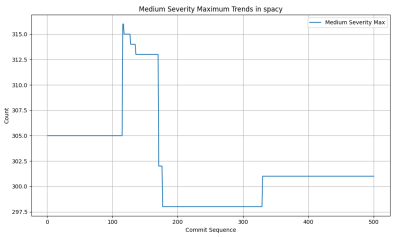


Figure 17: Medium Severity Max Trends

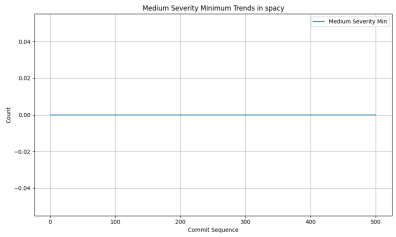


Figure 18: Medium Severity Min Trends

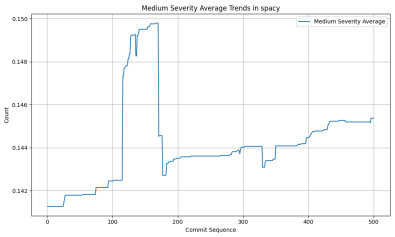


Figure 19: Medium Severity Avg Trends

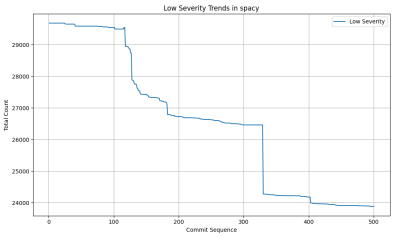


Figure 20: Low Severity Trends

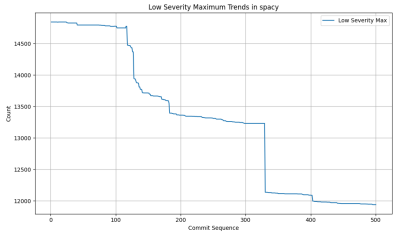


Figure 21: Low Severity Max Trends

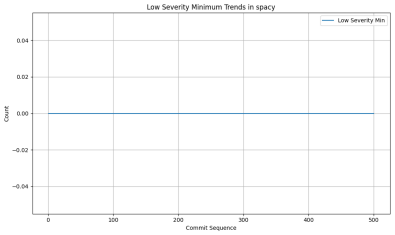


Figure 22: Low Severity Min Trends

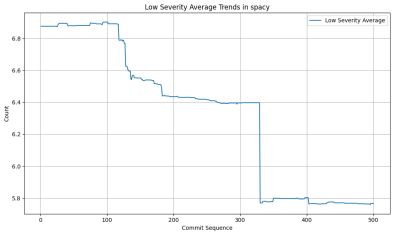


Figure 23: Low Severity Avg Trends

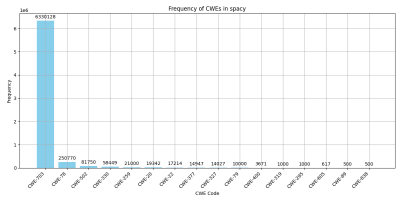


Figure 24: CWE Frequency

Figure 25: Spacy GitHub Repository Severity Analysis

2. **Transformers** (GitHub: https://github.com/huggingface/transformers).

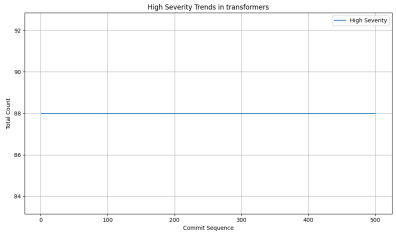


Figure 26: High Severity Trends

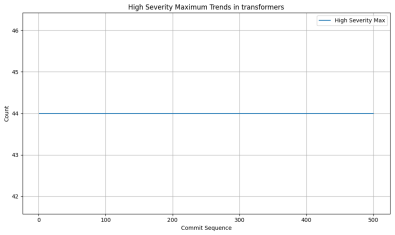


Figure 27: High Severity Max Trends

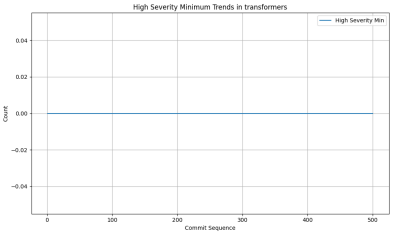


Figure 28: High Severity Min Trends

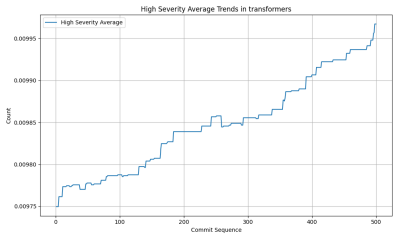


Figure 29: High Severity Avg Trends

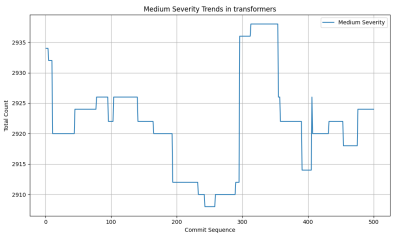


Figure 30: Medium Severity Trends

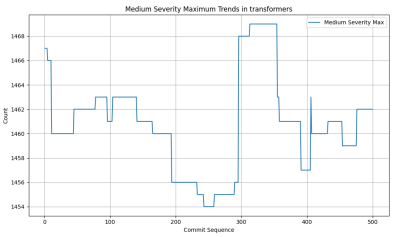


Figure 31: Medium Severity Max Trends

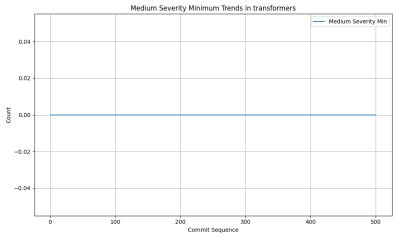


Figure 32: Medium Severity Min Trends

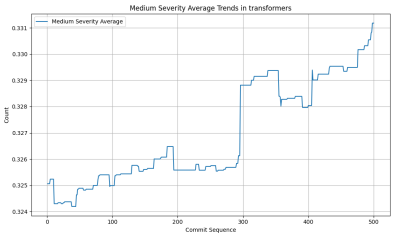


Figure 33: Medium Severity Avg Trends

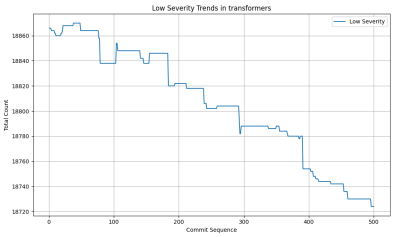


Figure 34: Low Severity Trends

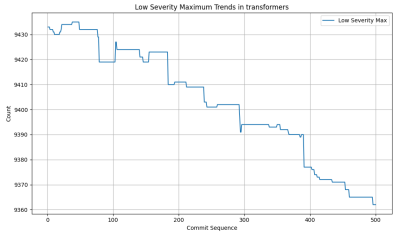


Figure 35: Low Severity Max Trends

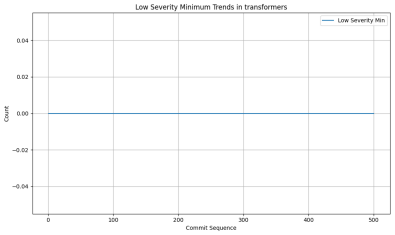


Figure 36: Low Severity Min Trends

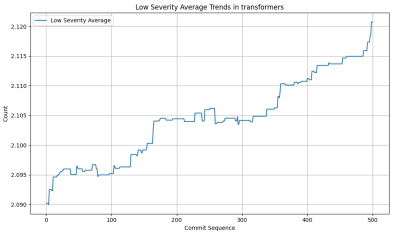


Figure 37: Low Severity Avg Trends

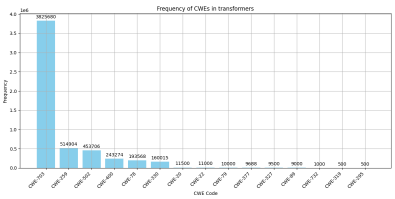


Figure 38: CWE Frequency

Figure 39: Transformers GitHub Repository Severity Analysis

3. **Keras** (GitHub: `https://github.com/keras-team/keras`).

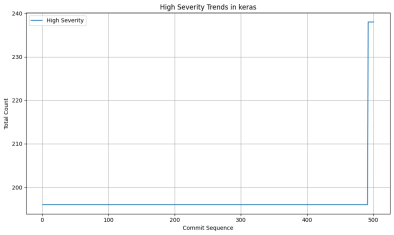


Figure 40: High Severity Trends

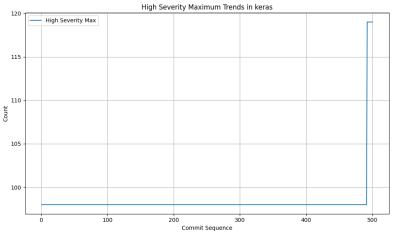


Figure 41: High Severity Max Trends

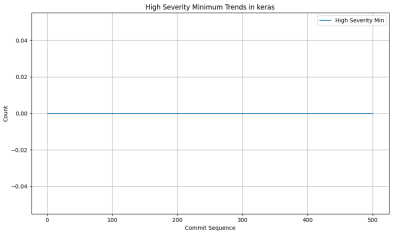


Figure 42: High Severity Min Trends

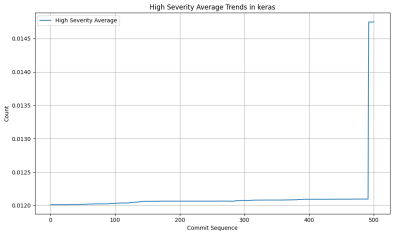


Figure 43: High Severity Avg Trends

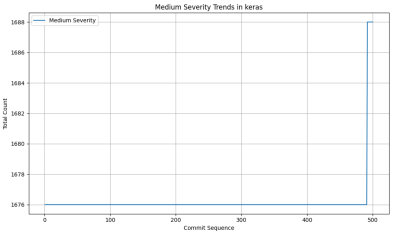


Figure 44: Medium Severity Trends

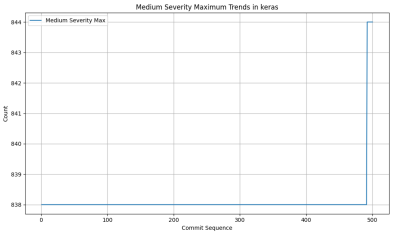


Figure 45: Medium Severity Max Trends

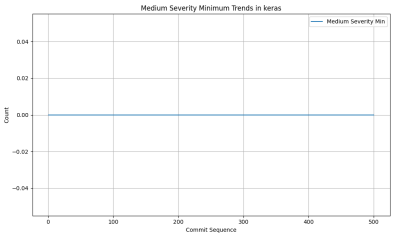


Figure 46: Medium Severity Min Trends

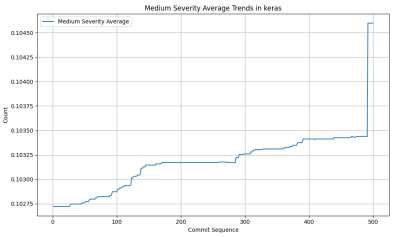


Figure 47: Medium Severity Avg Trends

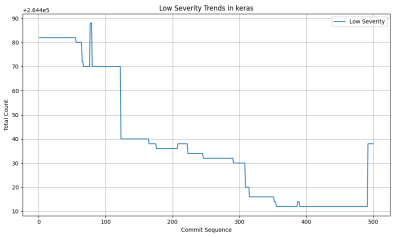


Figure 48: Low Severity Trends

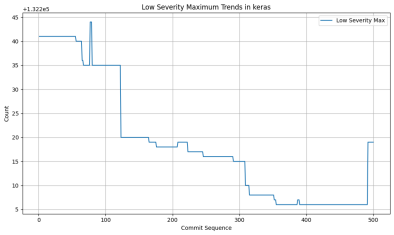


Figure 49: Low Severity Max Trends

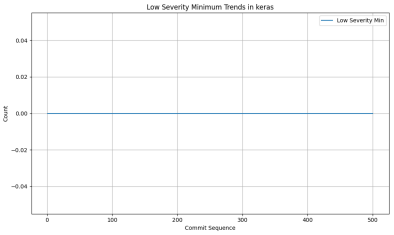


Figure 50: Low Severity Min Trends

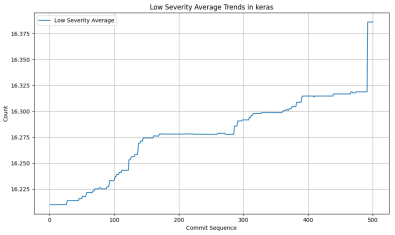


Figure 51: Low Severity Avg Trends

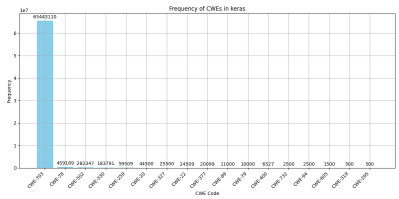


Figure 52: CWE Frequency

Figure 53: Keras GitHub Repository Severity Analysis

4. **Matplotlib** (GitHub: `https://github.com/matplotlib/matplotlib`).

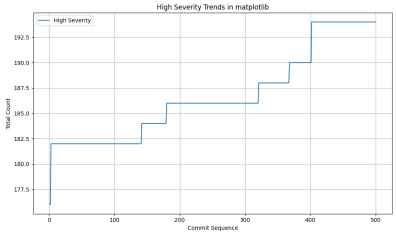


Figure 54: High Severity Trends

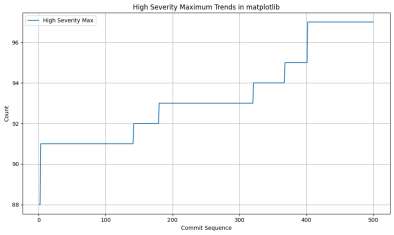


Figure 55: High Severity Max Trends

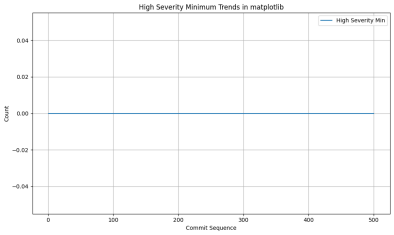


Figure 56: High Severity Min Trends

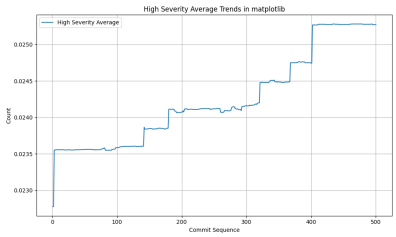


Figure 57: High Severity Avg Trends

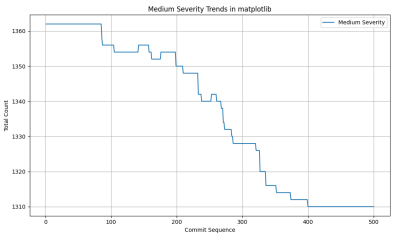


Figure 58: Medium Severity Trends

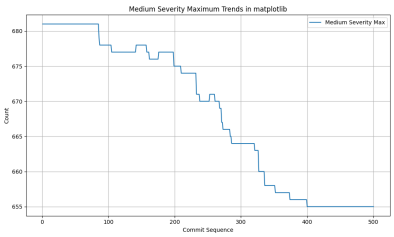


Figure 59: Medium Severity Max Trends

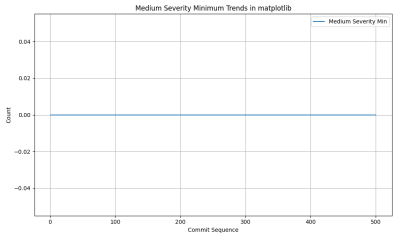


Figure 60: Medium Severity Min Trends

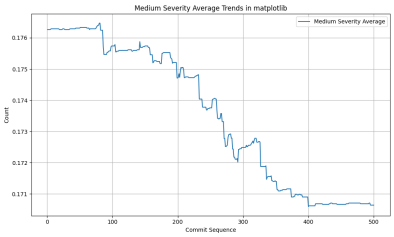


Figure 61: Medium Severity Avg Trends

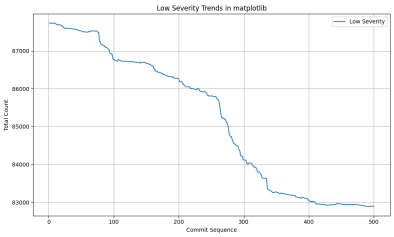


Figure 62: Low Severity Trends

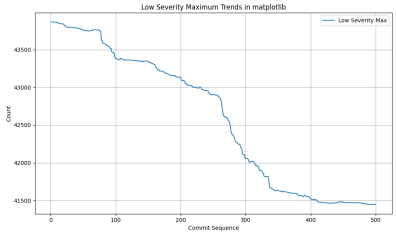


Figure 63: Low Severity Max Trends

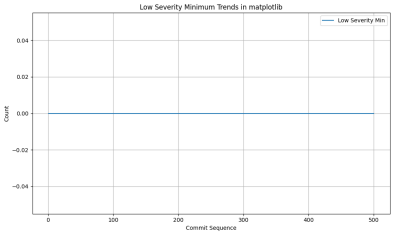


Figure 64: Low Severity Min Trends

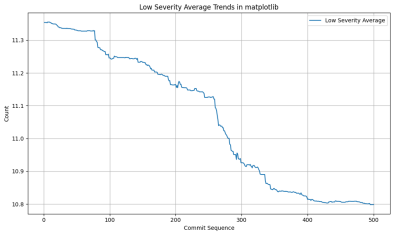


Figure 65: Low Severity Avg Trends

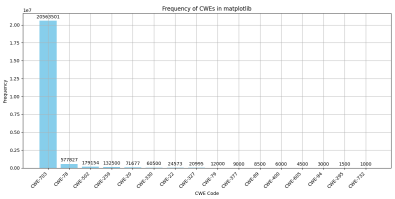


Figure 66: CWE Frequency

Figure 67: Matplotlib GitHub Repository Severity Analysis

5. **Numpy** (GitHub: `https://github.com/numpy/numpy`).

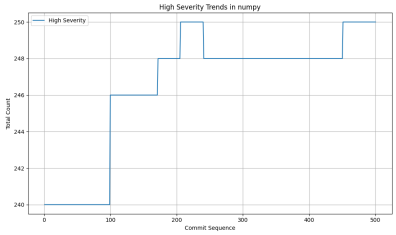


Figure 68: High Severity Trends

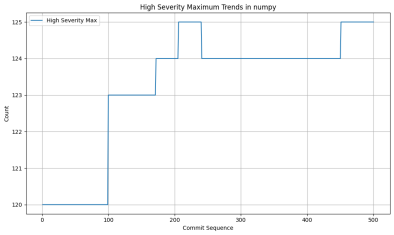


Figure 69: High Severity Max Trends

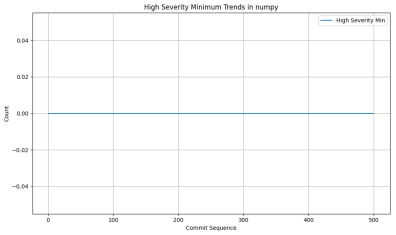


Figure 70: High Severity Min Trends

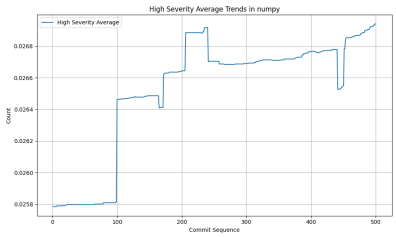


Figure 71: High Severity Avg Trends

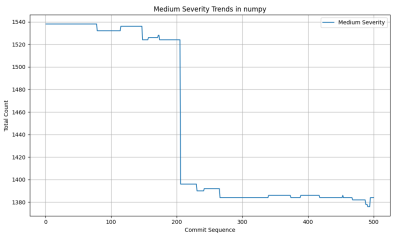


Figure 72: Medium Severity Trends

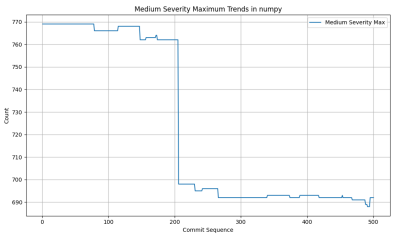


Figure 73: Medium Severity Max Trends

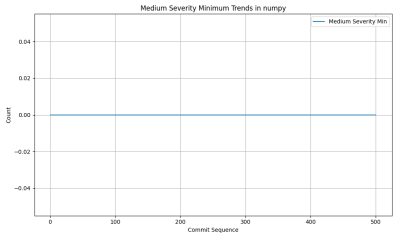


Figure 74: Medium Severity Min Trends

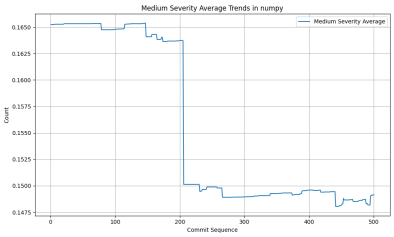


Figure 75: Medium Severity Avg Trends

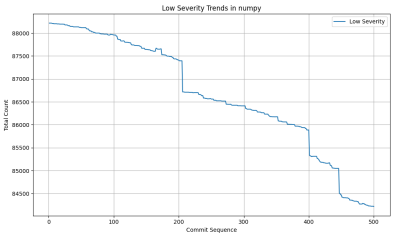


Figure 76: Low Severity Trends

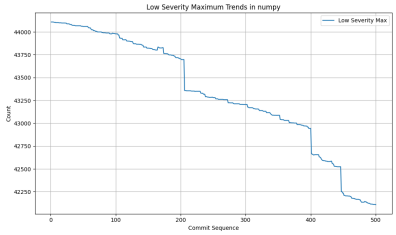


Figure 77: Low Severity Max Trends

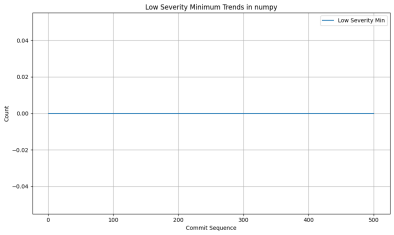


Figure 78: Low Severity Min Trends

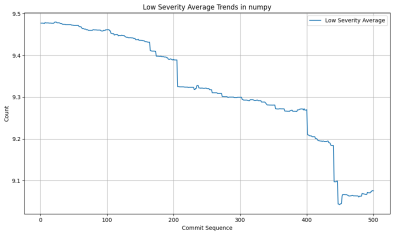


Figure 79: Low Severity Avg Trends

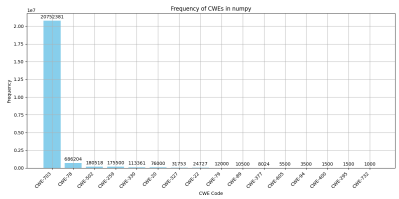


Figure 80: CWE Frequency

Figure 81: Numpy GitHub Repository Severity Analysis

6. **Pandas** (GitHub: `https://github.com/pandas-dev/pandas`).

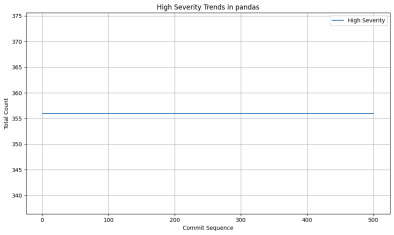


Figure 82: High Severity Trends

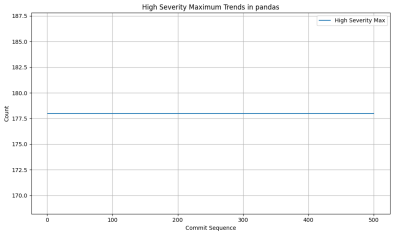


Figure 83: High Severity Max Trends

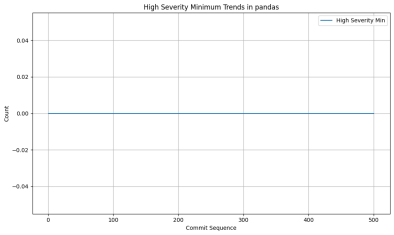


Figure 84: High Severity Min Trends

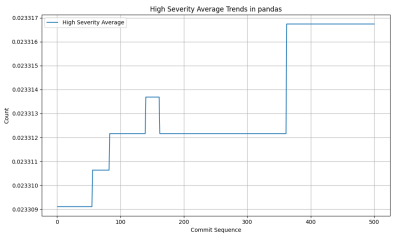


Figure 85: High Severity Avg Trends

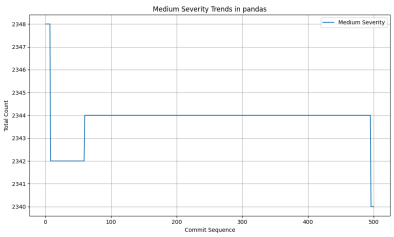


Figure 86: Medium Severity Trends

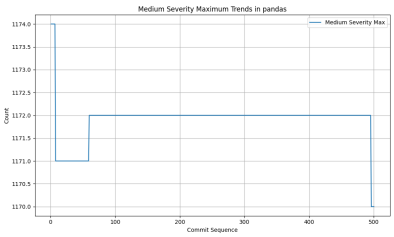


Figure 87: Medium Severity Max Trends

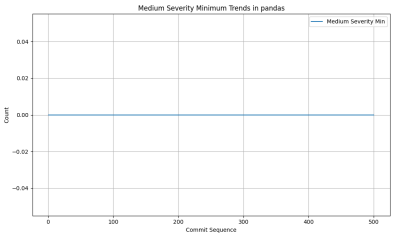


Figure 88: Medium Severity Min Trends

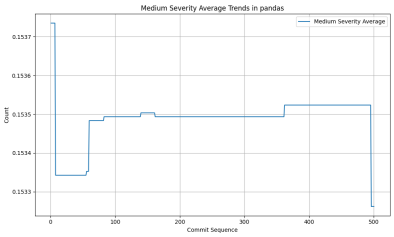


Figure 89: Medium Severity Avg Trends

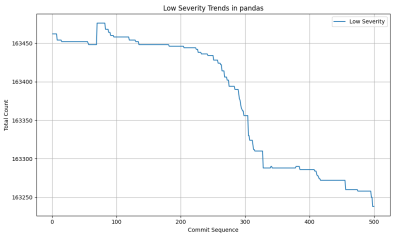


Figure 90: Low Severity Trends

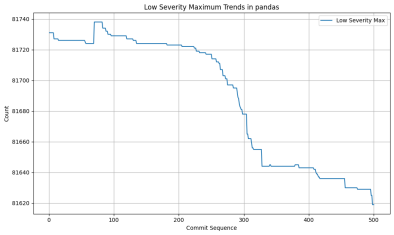


Figure 91: Low Severity Max Trends

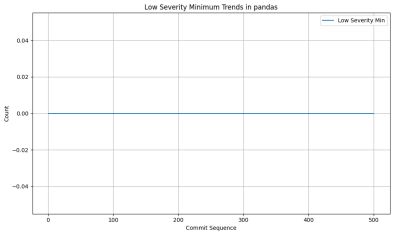


Figure 92: Low Severity Min Trends

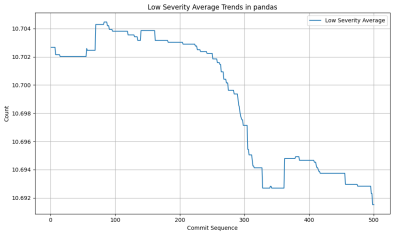


Figure 93: Low Severity Avg Trends

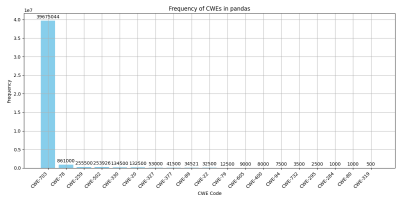


Figure 94: CWE Frequency

Figure 95: Pandas GitHub Repository Severity Analysis

7. **Pillow** (GitHub: `https://github.com/python-pillow/pillow`).

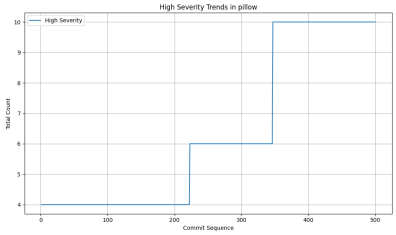


Figure 96: High Severity Trends

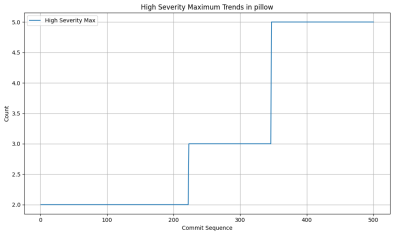


Figure 97: High Severity Max Trends

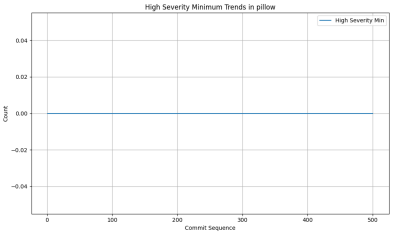


Figure 98: High Severity Min Trends

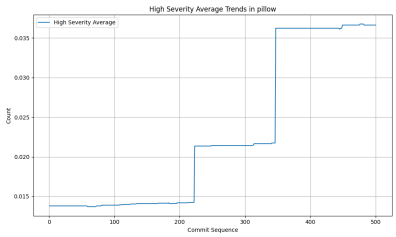


Figure 99: High Severity Avg Trends

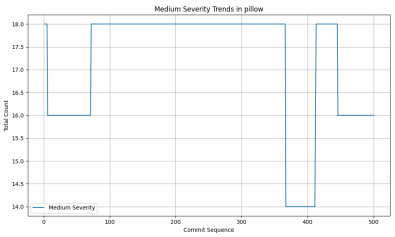


Figure 100: Medium Severity Trends

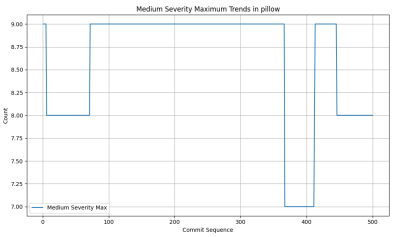


Figure 101: Medium Severity Max Trends

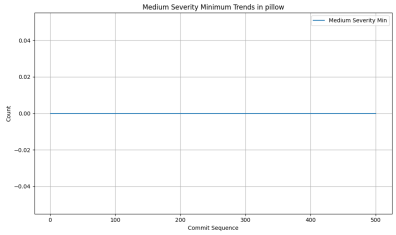


Figure 102: Medium Severity Min Trends

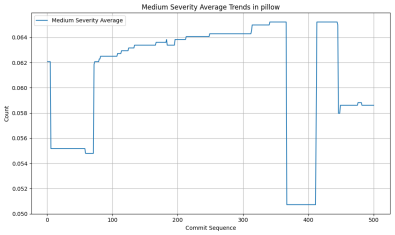


Figure 103: Medium Severity Avg Trends

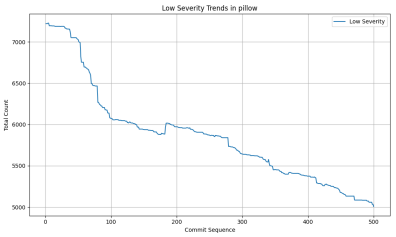


Figure 104: Low Severity Trends

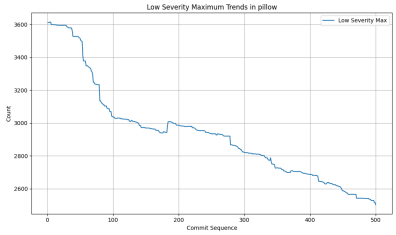


Figure 105: Low Severity Max Trends

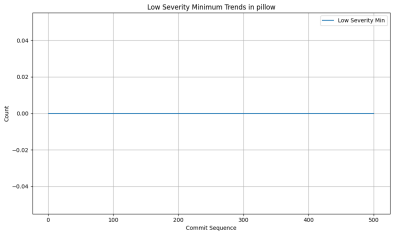


Figure 106: Low Severity Min Trends

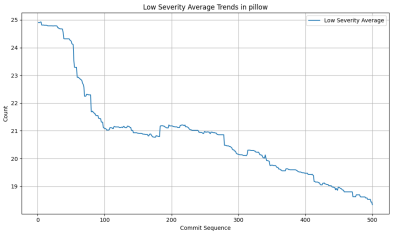


Figure 107: Low Severity Avg Trends

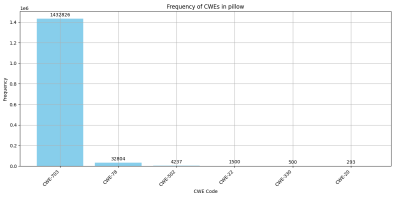


Figure 108: CWE Frequency

Figure 109: Pillow GitHub Repository Severity Analysis

8. **Scikit-learn** (GitHub: `https://github.com/scikit-learn/scikit-learn`).

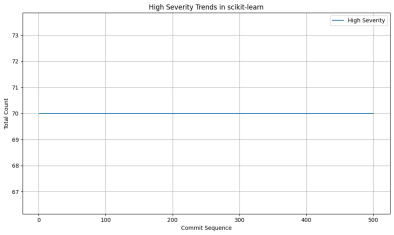


Figure 110: High Severity Trends

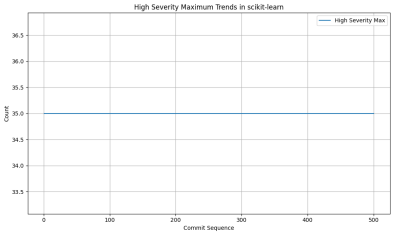


Figure 111: High Severity Max Trends

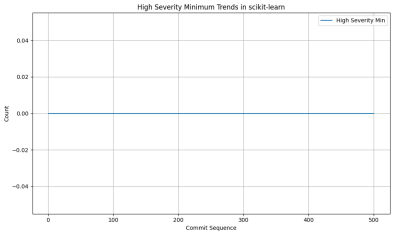


Figure 112: High Severity Min Trends

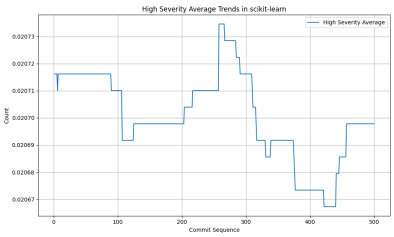


Figure 113: High Severity Avg Trends

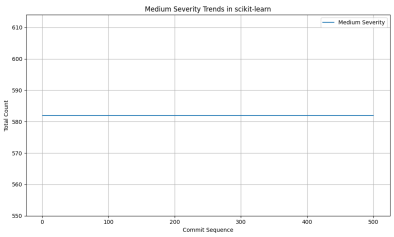


Figure 114: Medium Severity Trends

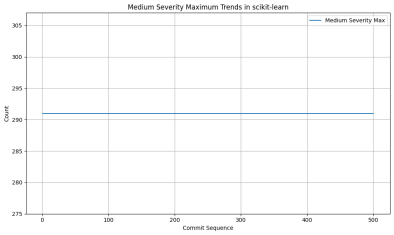


Figure 115: Medium Severity Max Trends

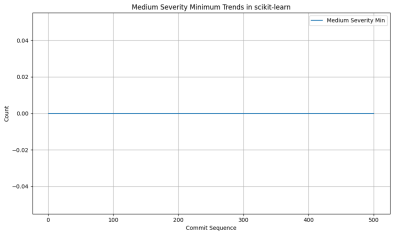


Figure 116: Medium Severity Min Trends

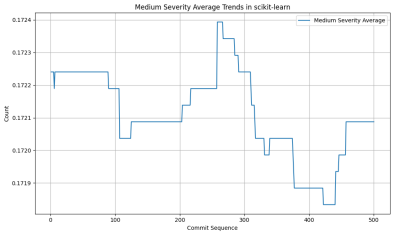


Figure 117: Medium Severity Avg Trends

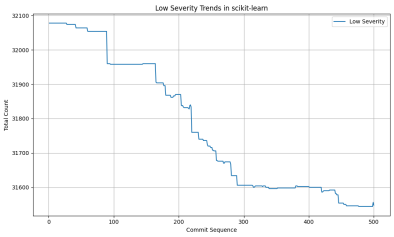


Figure 118: Low Severity Trends

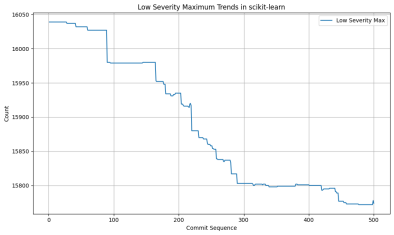


Figure 119: Low Severity Max Trends

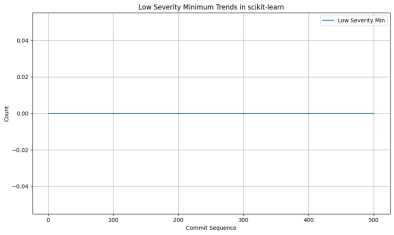


Figure 120: Low Severity Min Trends

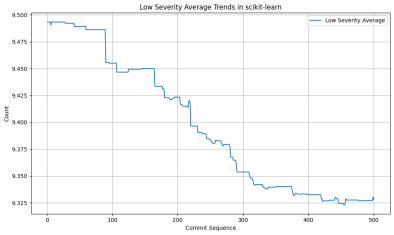


Figure 121: Low Severity Avg Trends

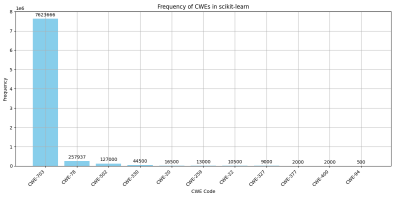


Figure 122: CWE Frequency

Figure 123: Scikit-learn GitHub Repository Severity Analysis

9. **Sqlmap** (GitHub: `https://github.com/sqlmapproject/sqlmap`).

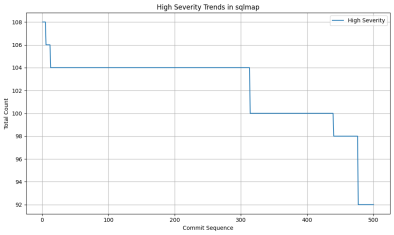


Figure 124: High Severity Trends

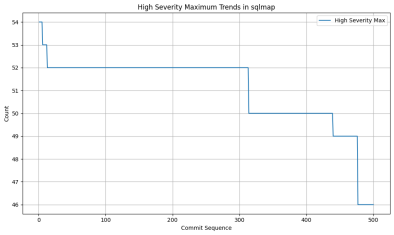


Figure 125: High Severity Max Trends

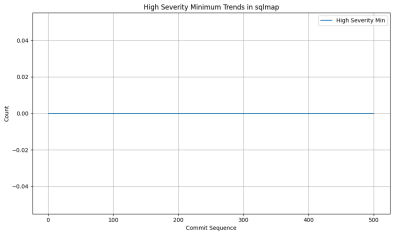


Figure 126: High Severity Min Trends

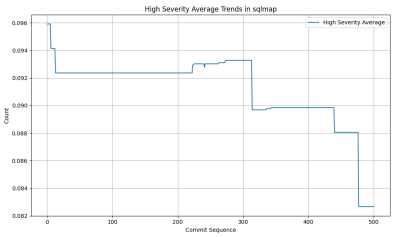


Figure 127: High Severity Avg Trends

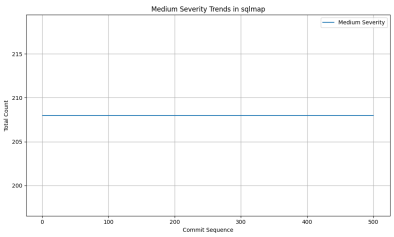


Figure 128: Medium Severity Trends

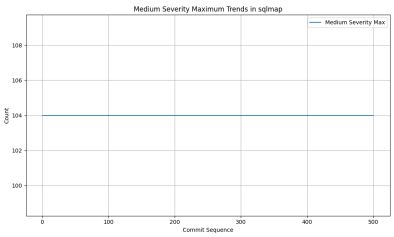


Figure 129: Medium Severity Max Trends

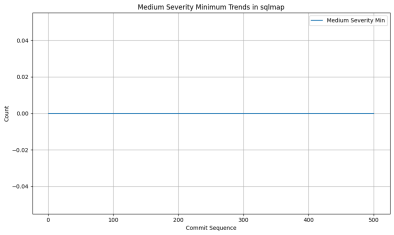


Figure 130: Medium Severity Min Trends

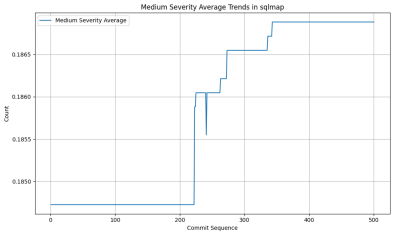


Figure 131: Medium Severity Avg Trends

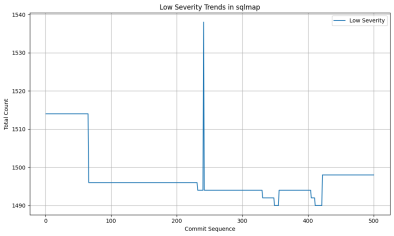


Figure 132: Low Severity Trends

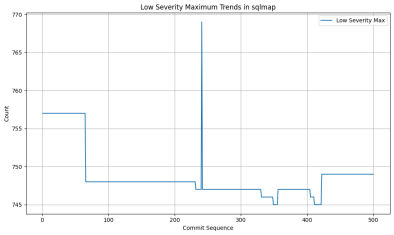


Figure 133: Low Severity Max Trends

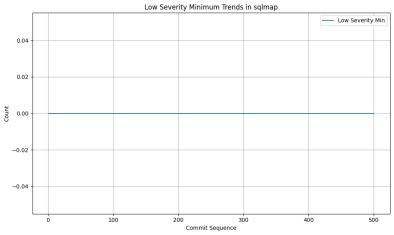


Figure 134: Low Severity Min Trends

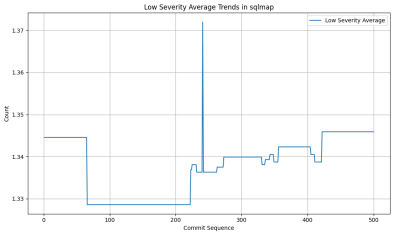


Figure 135: Low Severity Avg Trends

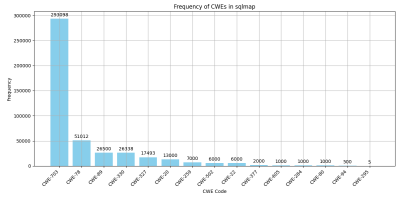


Figure 136: CWE Frequency

Figure 137: Sqlmap GitHub Repository Severity Analysis

10. **Tensorflow Models** (GitHub: `https://github.com/tensorflow/models`).

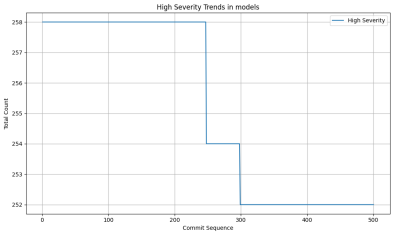


Figure 138: High Severity Trends

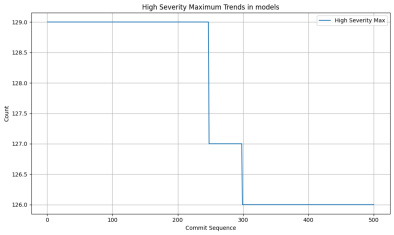


Figure 139: High Severity Max Trends

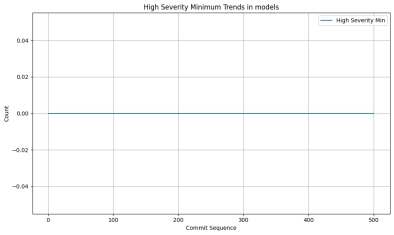


Figure 140: High Severity Min Trends

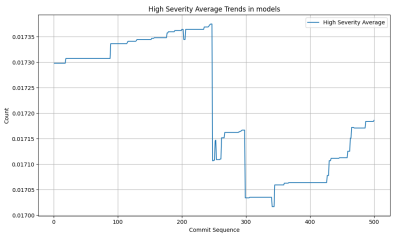


Figure 141: High Severity Avg Trends

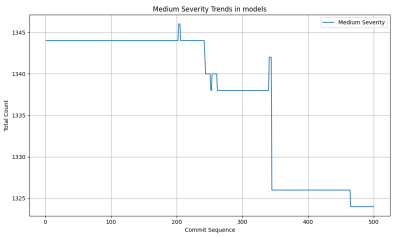


Figure 142: Medium Severity Trends

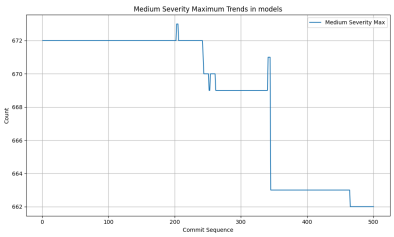


Figure 143: Medium Severity Max Trends

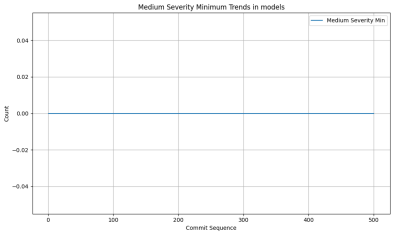


Figure 144: Medium Severity Min Trends

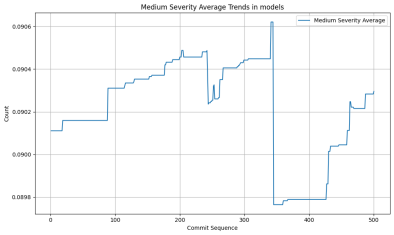


Figure 145: Medium Severity Avg Trends

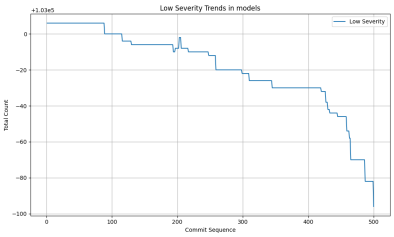


Figure 146: Low Severity Trends

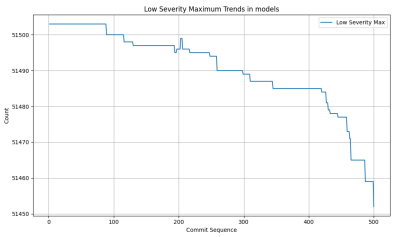


Figure 147: Low Severity Max Trends

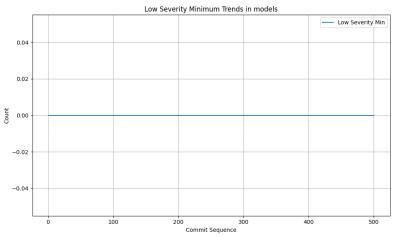


Figure 148: Low Severity Min Trends

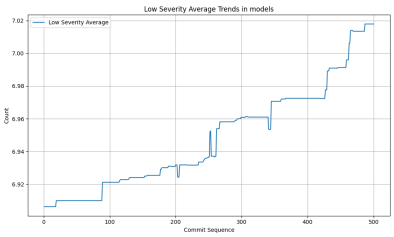


Figure 149: Low Severity Avg Trends

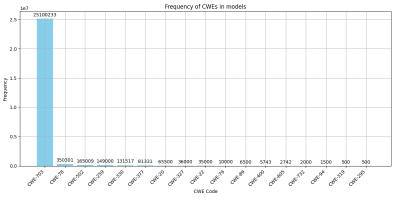


Figure 150: CWE Frequency

Figure 151: Tensorflow Models GitHub Repository Severity Analysis

3.3 Overall Dataset-Level Analyses (Research Questions)

All Bandit generated reports are available here: Bandit Reports for all Repositories
The dataset-level analysis addresses the following research questions:

RQ1: High Severity Vulnerabilities

Purpose: Determine when high severity vulnerabilities are introduced and subsequently fixed in the development timeline.
Approach: Analyze commit histories alongside `bandit` outputs to correlate vulnerability introduction and remediation.
Results: From the graphs of each repository, it can be observed that as the repository size increases, the number of high severity vulnerabilities increases, but at a slower rate. This may be because larger and more complex repositories make it more difficult to track every vulnerability. In contrast, in smaller repositories, the trend appears to be downward, likely due to the more manageable codebase, which allows for quicker identification and resolution of high severity vulnerabilities.
Takeaway: It is crucial to conduct timely vulnerability checks across the repository; otherwise, vulnerabilities can begin to increase at an exponential rate.

RQ2: Vulnerability Patterns Across Severities

Purpose: Investigate if vulnerabilities of different severity levels follow similar patterns in terms of introduction and elimination.
Approach: Compare trends in `bandit` output across HIGH, MEDIUM, and LOW severity issues.
Results: In most repositories, there is a significant gap between the number of low severity vulnerabilities and those of high and medium severity. However, it is also observed that low severity vulnerabilities decrease at a much faster rate compared to high and medium severity vulnerabilities. This may be due to the relatively lower complexity and ease of fixing low severity vulnerabilities, whereas addressing high and medium severity vulnerabilities requires considerably more effort and attention.
Takeaway: Although it is easier to fix low severity vulnerabilities but the high and medium vulnerabilities shouldn't be ignored as keeping them as low as possible to prevent any major damage to the code base. Regular fixing of these vulnerabilities is necessary so that later a low vulnerabilty doesn't become into a high vulnerabilty and becomes difficult to fix and manage.

RQ3: CWE Coverage

Purpose: Identify which CWEs are most frequently reported across the selected repositories.
Approach: Aggregate CWE data from all repositories and analyze their frequency distributions.
Results: The above CWE results from each repository indicate that CWE-703 (Improper Check or Handling of Exceptional Conditions) is the most prevalent vulnerability, with no other CWE ID coming close in frequency. This is primarily due to the common issue of inadequate use of try and except blocks, type errors, and the lack of comprehensive exception handling for all scenarios.
Takeaway: Proper handling of checks and exceptions is crucial, as incorrect inputs or outputs can cause the application to crash or behave unexpectedly.

4 Results and Analysis

4.1 Individual Repository Analysis

All Bandit generated reports are available here: Bandit Reports for all Repositories
As discussed in Section 3.2, all graphs and metrics are plotted.

1. **Spacy**
   - **High Severity Trends:** Moderate fluctuations with occasional spikes as shown in the "High Severity Max Trends" graph, indicating some commits introduce vulnerabilities more dramatically.
   - **Average & Minimum Trends:** The "High Severity Avg Trends" and "High Severity Min Trends" suggest that while the average is relatively stable, certain commits manage to keep vulnerabilities low, implying effective code reviews.
   - **Medium and Low Severity Trends:** Medium severity trends mirror the high severity pattern with less pronounced peaks, whereas low severity vulnerabilities—though more frequent—decrease rapidly, indicating faster remediation.
   - **CWE Frequency:** CWE-703 (Improper Check or Handling of Exceptional Conditions) is the most prevalent vulnerability.
   - **Overall Severity Analysis:** The severity analysis graph confirms a balanced trend where high severity issues, though less frequent, exhibit notable peaks compared to more steady medium and low severity issues.
2. **Transformers**
   - **High Severity Trends:** Displays sharper peaks than Spacy, suggesting that certain commits introduce significant security issues.
   - **Variation in Maximum and Minimum:** Greater volatility in the "High Severity Max" and "Min Trends" indicates a mix of controlled and problematic commits.
   - **Medium and Low Severity Patterns:** Medium severity issues show moderate variation while low severity vulnerabilities are reduced efficiently.
   - **CWE Frequency:** The frequency analysis again highlights CWE-703 as the dominant issue.
   - **Overall Insight:** The overall severity analysis shows that despite occasional high-risk commits, the trends tend to stabilize.
3. **Keras**
   - **High Severity Trends:** Periodic spikes are evident, with certain commits introducing significant vulnerabilities.
   - **Stability in Averages:** The "High Severity Avg Trends" indicate that most commits maintain a controlled level of vulnerabilities.
   - **Medium and Low Severity Observations:** Medium severity trends are less variable, and low severity issues decline steadily, suggesting effective fixes.
   - **CWE Frequency:** CWE-703 remains the most prevalent vulnerability.
   - **Severity Analysis Summary:** Overall, Keras demonstrates a controlled vulnerability management process despite occasional peaks.
4. **Matplotlib**

- **High Severity Trends:** Smoother and less erratic trends indicate consistent vulnerability management.
- **Consistency in Maximum and Minimum:** The "High Severity Max" and "Min Trends" show less dramatic changes, suggesting fewer risky commits.
- **Medium and Low Severity Trends:** Both trends are steady and exhibit low volatility, with low severity issues being effectively remediated.
- **CWE Frequency:** Consistent dominance of CWE-703 is observed.
- **Overall Analysis:** The severity analysis reflects controlled risk levels and effective vulnerability management.

5. **Numpy**
   - **High Severity Trends:** Noticeable spikes are present, likely correlating with significant code changes.
   - **Fluctuations in Average Trends:** The average trend shows that most commits are moderate in terms of vulnerability risk.
   - **Medium and Low Severity Observations:** Medium severity trends are moderate, while low severity vulnerabilities, although frequent, decrease rapidly.
   - **CWE Frequency:** CWE-703 is again the predominant issue.
   - **Severity Analysis Overview:** Overall, Numpy shows variability in high severity issues but maintains consistent remediation.
6. **Pandas**
   - **High Severity Trends:** Exhibits volatility with notable peaks indicating moments of increased risk.
   - **Max and Average Trends:** Occasional large spikes in the "High Severity Max Trends" are balanced by moderate average values.
   - **Medium and Low Severity Trends:** Medium severity trends are stable, and low severity issues are reduced steadily.
   - **CWE Frequency:** CWE-703 is the most common vulnerability.
   - **Overall Severity Analysis:** Demonstrates effective vulnerability management despite sporadic high-risk commits.
7. **Pillow**
   - **High Severity Trends:** Distinct spikes in certain commits highlight intermittent critical issues.
   - **Variation in Maximum and Minimum Trends:** The maximum values show significant increases, while the minimum values remain low.
   - **Medium and Low Severity Patterns:** Medium severity trends are moderately volatile and low severity vulnerabilities decline quickly.
   - **CWE Frequency:** CWE-703 dominates the frequency analysis.
   - **Severity Analysis Conclusion:** Intermittent high-risk commits are balanced by an overall controlled vulnerability profile.
8. **Scikit-learn**
   - **High Severity Trends:** Steady with occasional peaks, indicating most commits are well-managed.

- **Stability in Medium and Low Severity:** Minimal volatility in medium and low severity trends.
- **CWE Frequency:** Consistent with CWE-703 being the most frequently reported vulnerability.
- **Overall Severity Analysis:** Reflects consistent vulnerability management with only sporadic high severity outliers.

9. **Sqlmap**
   - **High Severity Trends:** Several spikes suggest that major changes or feature introductions have led to high vulnerability risks.
   - **High Variability in Maximum Trends:** Some commits exhibit significantly higher risk levels.
   - **Medium and Low Severity Patterns:** Medium severity trends show moderate variation, while low severity issues are reduced at a faster pace.
   - **CWE Frequency:** CWE-703 continues to be the dominant vulnerability.
   - **Overall Severity Analysis:** Despite bursts of high severity incidents, the repository shows systematic remediation.
10. **Tensorflow Models**
   - **High Severity Trends:** Significant peaks indicate vulnerable commits, likely due to complex integrations.
   - **Medium and Low Severity Observations:** Moderate variations in medium severity with a consistent decline in low severity issues.
   - **CWE Frequency:** CWE-703 remains dominant.
   - **Overall Severity Analysis:** While experiencing intermittent high-risk commits, the overall vulnerability management remains balanced.

5 Discussion and Conclusion

5.1 Challenges and Reflections

Managing multiple repositories and isolated virtual environments posed logistical challenges. Analyzing large volumes of commit data required effective aggregation and visualization techniques. Interpreting `bandit` outputs necessitated a careful review to correlate vulnerabilities with specific code changes.

5.2 Lessons Learned

Static analysis tools like `bandit` are critical for early vulnerability detection. A systematic repository selection process enhances the quality of security assessments. Clear documentation and visualization of results are essential for translating data into actionable insights.

5.3 Summary

Labs 07 and 08 have provided a comprehensive exploration into the security vulnerabilities of large-scale open-source Python projects. The use of `bandit` enabled detailed analysis at both the individual repository and dataset levels. The insights obtained from answering the research questions contribute to a better understanding of vulnerability patterns and common weaknesses in the open-source ecosystem.

6 Appendix

6.1 Tools and Resources

- **Bandit:** Bandit Documentation
- **GitHub Repository for Bandit:** Bandit on GitHub
- **SEART GitHub Search Engine:** SEART