MLOps
Lab0

Fundamentals of Continuous Integration

Iker Jauregui

Contents

1 Introduction

In this report, we will see the logic behind the implemented unit and integration tests for the CI assignment (link to GitHub repository). These tests have two main purposes. On the one hand, they are meant to check if the implemented functionalities under *cli.py* and *preprocessing.py* modules (Figure 1) work correctly. On the other hand, they try to verify if the implementation of the code has taking into account rare or unexpected inputs or usages.

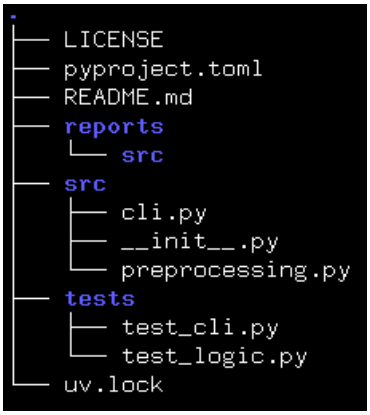

Figure 1: Project structure.

2 Tests

2.1 Unit Tests

This set of tests are included on the *test_logic.py* file and are related to the *preprocessing.py* module functionality. This module implements a set of preprocessing tools for different types of input values, and it is the core of the project. So, the implemented unit tests are really exhaustive and aim to check not only the common and expected performance, but also rare cases.

```python
# Tests for "normalize_min_max"
@pytest.mark.parametrize(
    "values,min_val,max_val,expected",
    [
        ([], 0.0, 1.0, []),  # Empty list
        ([1], 0.0, 1.0, [0.0]),  # One element
        ([0.0, 1.0], 0.0, 1.0, [0.0, 1.0]),  # Two elements
        ([1, 2, 3, 4, 5], 0.0, 1.0, [0.0, 0.25, 0.5, 0.75, 1.0]),  # Normal case
        ([5, 4, 3, 2, 1], 0.0, 1.0, [1.0, 0.75, 0.5, 0.25, 0.0]),  # Descending order
        ([1, 5, 4, 2, 3], 0.0, 1.0, [0.0, 1.0, 0.75, 0.25, 0.5]),  # Random order
        ([0.0, 1.0], -1.0, 1.0, [-1.0, 1.0]),  # Negative min + Positive max
        ([0.0, 1.0], -5.0, -1.0, [-5.0, -1.0]),  # Negative min and max
        ([-5.0, 0.0, 5.0], 0.0, 1.0, [0.0, 0.5, 1.0]),  # Negative and positive inputs
        ([-15.0, -10.0, -5.0], 0.0, 1.0, [0.0, 0.5, 1.0]),  # All negative inputs
    ],
)
def test_normalize_min_max_parametrized(values, min_val, max_val, expected):
    """Test normalize_min_max with various inputs."""
    result = preprocessing.normalize_min_max(values, min_val, max_val)
    assert result == pytest.approx(expected, rel=1e-5)
```

Figure 2: Implemented unit test for *normalize_min_max* function.

If we take one of the unit tests as the one developed for *normalize_min_max* function (Figure 2), we can notice the common testing template applied between all unit tests:

- **Edge cases**: Empty list, single element, two elements.
- **Order variations**: Ascending ($[1, 2, 3, 4, 5]$), descending ($[5, 4, 3, 2, 1]$), random order ($[1, 5, 4, 2, 3]$).
- **Range configurations**:
  - Standard minimum and maximum: $[0, 1]$
  - Negative minimum: $[-1, 1]$
  - Both negative: $[-5, -1]$
- **Input value diversity and data types**:
  - All positive (integer values): $[1, 2, 3, 4, 5]$
  - Mixed signs (float values): $[-5.0, 0.0, 5.0]$
  - All negative (float values): $[-15.0, -10.0, -5.0]$

2.2 Integration Tests

This set of tests are included on the *test_cli.py* file and are related to the *cli.py* module functionality. This module uses the *preprocessing.py* module and builds a command line wrapper for each of the imported functions. So, this time, even if common use cases were covered, the developed integration tests were designed to test the interaction between the end user and the CLI interface, putting more effort on verifying that provided commands worked correctly with all kind of input values.

```python
# Test for "normalize" with successful performance
@pytest.mark.parametrize(
    "values,new_min,new_max,expected",
    [
        ("[1, 2, 3, 4, 5]", "0", "1", "0.0"),
        ("[1, 2, 3, 4, 5]", "0", "10", "2.5"),
        ("[0, 10]", "-1", "1", "-1.0"),
        ("[]", "0", "1", "[]"),
        ("[5]", "0", "1", "["),
    ],
)
def test_normalize_exit_ok(runner, values, new_min, new_max, expected):
    """Test normalize command with valid inputs."""
    result = runner.invoke(
        cli,
        ["numeric", "normalize", values, "--new-min", new_min, "--new-max", new_max],
    )
    assert result.exit_code == 0
    assert expected in result.output


# Test for "normalize" with erratic performance
@pytest.mark.parametrize(
    "values,new_min,new_max",
    [
        ("not-json", "0", "1"),
        ("\\{1, 2\\}", "0", "1"),
        (1, "0", "1"),
        (None, "0", "1"),
        (1.0, "0", "1"),
    ],
)
def test_normalize_error(runner, values, new_min, new_max):
    """Test normalize command with invalid inputs."""
    result = runner.invoke(
        cli,
        ["numeric", "normalize", values, "--new-min", new_min, "--new-max", new_max],
    )
    assert ("Error" in result.output) or (result.exit_code != 0)
```

Figure 3: Implemented integration tests for *normalize* command.

If we look at the implementation of the integration tests of the *normalize* command (Figure 3), we can see the shared testing structure between these type of tests:

- **Successful executions**: The functions ended with *_exit_ok* at its name test the normal use cases of the command.
- **Erratic performance**: All erratic behavior is tested under the *_error* functions.

3 Linting and Formatting Processes

Several linting and formatting steps were performed during the development of both source code and tests. Here is the output of the Linux terminal when executing one of those steps:

Listing 1: Pylint analysis before formatting

```
(mlpops-lab0) alumno@eim-alu-83079:~/Desktop/datos/MLOps/Lab0/MLPOps_Lab0$ uv
    run python -m pylint src/*.py
************* Module src.cli
src/cli.py:31:4: W0107: Unnecessary pass statement (unnecessary-pass)
src/cli.py:41:4: W0107: Unnecessary pass statement (unnecessary-pass)
src/cli.py:59:11: W0718: Catching too general exception Exception (broad-
    exception-caught)
[... additional warnings ...]
src/cli.py:24:0: C0411: standard import "json" should be placed before third
    party import "click" (wrong-import-order)
************* Module src.__init__
src/__init__.py:7:0: C0304: Final newline missing (missing-final-newline)
************* Module src
src/__init__.py:1:0: C0114: Missing module docstring (missing-module-docstring)
************* Module src.preprocessing
src/preprocessing.py:27:0: C0301: Line too long (129/100) (line-too-long)
src/preprocessing.py:50:0: C0303: Trailing whitespace (trailing-whitespace)
src/preprocessing.py:75:0: C0303: Trailing whitespace (trailing-whitespace)
src/preprocessing.py:319:0: C0305: Trailing newlines (trailing-newlines)
src/preprocessing.py:1:0: C0114: Missing module docstring (missing-module-
    docstring)
src/preprocessing.py:5:0: E0401: Unable to import 'numpy' (import-error)

--------------------------------------------------------------------
Your code has been rated at 8.32/10 (previous run: 8.32/10, +0.00)
```

Listing 2: Black formatter execution

```
(mlpops-lab0) alumno@eim-alu-83079:~/Desktop/datos/MLOps/Lab0/MLPOps_Lab0$ uv
    run black src/*.py
reformatted src/__init__.py
reformatted src/preprocessing.py
reformatted src/cli.py

All done! @\textcolor{yellow}{\ding{72}}@ @\textcolor{yellow}{\ding{72}}@
3 files reformatted.
```

Listing 3: Pylint analysis after Black formatting

```
(mlpops-lab0) alumno@eim-alu-83079:~/Desktop/datos/MLOps/Lab0/MLPOps_Lab0$ uv
    run python -m pylint src/*.py
************* Module src.cli
src/cli.py:31:4: W0107: Unnecessary pass statement (unnecessary-pass)
[... warnings ...]

-------------------------------------------------------------------
Your code has been rated at 8.58/10 (previous run: 8.32/10, +0.26)
```

Listing 4: Adding numpy dependency

```
(mlpops-lab0) alumno@eim-alu-83079:~/Desktop/datos/MLOps/Lab0/MLPOps_Lab0$ uv
    add numpy
Resolved 22 packages in 305ms
Installing wheels...
warning: Failed to hardlink files; falling back to full copy.
Installed 1 package in 1.69s
 + numpy==2.3.4
```

Listing 5: Final pylint analysis with improved score

```
(mlpops-lab0) alumno@eim-alu-83079:~/Desktop/datos/MLOps/Lab0/MLPOps_Lab0$ uv
    run python -m pylint src/*.py
************* Module src.cli
src/cli.py:31:4: W0107: Unnecessary pass statement (unnecessary-pass)
[... remaining warnings ...]

-------------------------------------------------------------------
Your code has been rated at 9.00/10 (previous run: 8.84/10, +0.16)
```

4 Test Results and Coverage

Listing 6: Final result summary after running *uv run python -m pytest -v –cov=src*

```
================================ tests coverage
    ================================
_____________ coverage: platform linux, python 3.13.7-final-0
    ---------------

Name                 Stmts   Miss  Cover
----------------------------------------
src/__init__.py          3      0   100%
src/cli.py             177     30    83%
src/preprocessing.py    62      5    92%
----------------------------------------
TOTAL                  242     35    86%
======================== 236 passed, 10 warnings in 0.79s
    ========================
```

5 Conclusions

The testing phase yielded successful results with all 236 tests passing and achieving 86% overall code coverage. The *preprocessing.py* module demonstrated robust implementation with 92% coverage, while the *cli.py* interface reached 83% coverage through comprehensive integration tests.

Future improvements could focus on increasing CLI coverage by testing additional edge cases in command parsing and enhancing error handling for specific exception types beyond the current broad exception catching pattern. Nonetheless, the current test suite provides a solid foundation for confident deployment and future feature development.