Automation of:

Data Acquisition and Transformation pipeline

Tech stack

- Python (IDE: VS Code Editor)
  - README.md
  - requirements.txt
  - setup.py
  - my_package/
    - __init__.py
    - main.py
  - MANIFEST.in
- Git (Repo: Gitlab)
- Jenkins (.jenkins file): CI/CD
- Steamlit (UI)

- HDFS (Hadoop distributed file storage system)
- Hive (data warehouse system)
- Spark/MapReduce
- CDSW (collaborative environment)
- NiFi (data movement, ingestion, ETL workflows)
- DAP
- Artifactory (packages, docker images)

ETL Pipeline

1) Python (data extraction and transformation) -> Git -> Jenkins
   - Data loaded using NiFi
2) HDFS -> Spark -> HDFS -> Hive -> SQL
   - HDFS: storage system to read and write data
   - Hive: structured data storage for managing schema and querying
3) NiFi -> HDFS -> Hive -> DAP -> CDSW -> PySpark -> Docker container -> production environment
   - Jenkins: automate deployment, triggered by code changes on Git
   - CDSW (jupyter notebook IDE): Further data Analysis, using Python or PySpark
   - Artifactory: containers docker images and python packages

Data Acquisition

Static web-scraping: Beautiful Soup (Parsing HTML/XML content of the web page)

API Integration: Access structured data directly via an API

Regex (‘re’ library in Python): Extract or search for specific patterns (dates, urls)

Requests (‘requests’ library in Python): for HTTP requests (Get)

Practices implemented

- Use of Proxies
- Error handling
- Unit tests (unittest, pytest, responses, mock, pytest-mock): mock status code (200), correct html parsing

Transformation

- Pandas, RegEx, OS
  - Web scraping
  - File renaming
  - File handling and transfers
  - Data cleaning and standardisation
  - Re-formatting and restructuring
- PySpark
  - complex transformations, aggregations, and filtering
  - processing large volumes of data in parallel
- Hive
  - structured SQL-based querying
  - data transformation on large datasets stored in HDFS

Unit tests (unittest: Python's built-in module)

Given originalfunction():

```
Class TestFunction(unittest.TestCase):
        def test_function_edge_case(self):
                result = original_function()
                self.assertEqual(result, expected_result)
```

Mocking:

@mock.patch decorator: replace request.get with a mock -> can be used as a context manager too

return_value: stimulate response with a status code

mock.MagicMock : supports magic methods

Unit tests (pytest: testing framework Python library)

- Write test functions starting with test_
- Supports fixtures, parametrised tests
- Standard Python assert statements
- Plugins for mocking (pytest-mock: wrapper around unittest.mock module)

@pytest.fixture decorator: set up and tear down resources before and after tests

@pytest.mark.parametrize decorator: run a test with different input values

- Can use pytest with unittest.mock - @mock.patch decorator or patch context manager (with patch as …)

Testing

- Ensure successful HTTP request (200 status code)
- Verify timeout and retry mechanisms
- Ensure the correct extraction of desired elements from HTML (e.g., titles, links, tables), and validate regex patterns
- Test error handling of 404/500 errors or invalid URLs: test for exceptions raised
- Test for correct logging messages for certain actions or failures
- Test if the correct exceptions are raised (eg. ValueError, TypeError, KeyError, FileNotFoundError, TimeoutError)
- Test for different edge cases (eg. Empty input, Boundary values, Negative numbers, Empty data structures, Null inputs)

Deployment

UI: Steamlit

1) Select: year, website, datasets
2) Answer user questions

Best practices: requirements.txt, docker, Jenkins CI/CD, modular programming and testing

Result:

- File downloads (Web scraping)
- Validation: data integrity checks
- Preprocessing: data transformations
- Export/File transfers
- Store in HDFS
- Create structured queries and tables in Hive
- Further analyses, using PySpark in CDSW