Student Name: Alejandra Perea Rojas

# COMPSCI 91R
**Synthesizing a Testing Framework**

---

Users from all over the world use the PAWS SMART API, and the quality of data received is important. However, better checks and more extensive handling is needed so that the system can accommodate and meet the needs of these users. It's imperative that this is done in PAWS since most users might not know how to fix errors like an invalid polygon due to its geometry, whereas it's possible for a computer programmer to do so with Python modules. Therefore, from edge cases like these, it's important to start considering and documenting these edge cases, so that we can begin testing and implementing error handling. In this paper, there is an outline of the pipeline that could help address this objective of accommodating more users.

The pipeline involves Data Creation, a Framework for Data, Automated Validity Checks (like Shapefile Checking), and Integrated Testing for simplicity of handling testing files with one script. I also describe the simplest way to execute the testing program.

The data can be found mainly on GitHub, and a copy has been uploaded to Google Drive, while the test park data was manually added to Azure. The design pipeline has only been started, and it has brought up more extensive tasks such as creation of more raster files (both elevation and land cover), shape file checking for more geometric errors, and Coordinate Reference System projections (on the testing side and data creation side). Most of these can be created with Python and QGIS.

## Data Creation

The objective of this section is to develop a comprehensive suite of data set up for testing, ready for complete runs (JSON, shapefiles, CSVs, rasters).

To start the set-up of this data, we can start with edge cases, which includes:

- CSV errors: Patrol observations might crash PAWS with (for example):
  - No match in filters (defined in JSON scripts)
  - Invalid filter name
- Boundary shapefiles: The examples below and more are described more extensively in Shapefile Checking
  - Zip erros/ empty boundarys shoud not run
  - Multiple polygons should run
  - Empty boundaries should run- but what can it predict?
- Coordinate Reference System: A boundary in WGS84 or the correct UTM zone should run. However, wrong projections from WGS84 to a UTM zone should not run.
- Raster files:

    - Elevation raster
    - Land cover raster

Most of the data described has been created and synthesized into one sample input that has a series of JSON scripts and can be executed in a specific manner.

## Testing Data Framework

### File structure

- The link to a spreadsheet outlining every test-case, to which the JSON scripts pertain to, can be found here: **Test Data**.
- The link to a sample input directory in GitHub is **Sample Input**. This contains directories corresponding to each park, in which, there are sub-cases that are the test cases themselves. In addition, each test case will have a JSON script. The program described in the section Integrated Testing can also be found there.

### Validity Checks: Shapefile Checking Case

The function `shape_check` (found in Appendix) takes a set of shapefiles through a CSV data structure under the variables `(Park, Label, Filepath, Exp_Valid)`. This are processed by the function `shape_check`, and in the scope of this report, handles Python's `explain_validity` from the `shapely.validation` module. This can be used verify and explain validity of files through their geometry. Interestingly, the program will print or return the shapefiles that not match the expected outcome (to be valid or not). The test cases that were first found to have invalid shapefiles with self-intersections and specifically ring self-intersections (more detail in file outputs or running `shape_check`!).

Overall, a few characteristics that can make a shapefile invalid include:

- Missing or incomplete files: A shapefile must include all of the required files in order to be considered valid. If any of the required files are missing or incomplete, the shapefile will be invalid.
- Incorrect file extensions: Each file in a shapefile must have the correct file extension. If any of the files have the wrong file extension, the shapefile will be invalid.
- Inconsistent data: The data contained in the different files of a shapefile must be consistent. If the data is inconsistent, the shapefile will be invalid.
- Corrupted files: If any of the files in a shapefile are corrupted, the shapefile will be invalid. These are specific examples, but there should be more cases found. This in the PAWS case, would be focued on not only boundary shape files, but to roads, villages, and lakes as well (lines, points, polygons).
    - Ring self-intersection in `SMARTdata/matching.zip`
    - Self-intersection in `large_park/boundary.zip`

Overall, the PAWS system should probably not attempt to handle predictions on the first three characteristics listed, as these would throw an error immediately. However, the last characteristic of corrupted files is worth looking into since the large park boundary is invalid and has self-intersection, and it runs and predicts to completion.

### Integrated Testing

Testing the PAWS API interface should be done through POST requests and a JSON script that is then handled by `api.py`, `preprocess.py`, etc. To do this, it was established that an option would be to use the Postman interface to send POST requests to a defined address with a JSON script, the subscription key, the local/test key, and the local/test address. Both local and test key, address pairs are different since local deployment will work differently than running in the test server. One would follow this process, individually, for every request that pertained to a specific park data set to be tested.

Our objective here is to automate the process of executing POST requests with a synthesized set of JSON scripts and park data infrastructure, as described earlier. The current development for this is the script `test.py` attached to the end of this document.

Note that the test server requires special handling due to the more complicated access to the testing server. To run this to the local server, one must follow the specific steps:

- Set up Azure (link to Deployment Guide - check the setting up Azure section)
- Follow the login, docker build, docker run. To kill a process, docker ps, and kill ps id in a separate shell.
- With the local server running (through the docker image from docker run), 1) make sure that the JSON scripts and the CSV file determining, which parks to be run is accurate to the testing desired; 2)the test data is on Azure and follows the naming conventions (same as CSV definitions and JSON scripts); 3) execute the script.

Finally, beyond writing the script, a tiny amount of changes were made to `preprocess.py` and `api.py`. This is to display more information about sub-cases (e.g. large park + invalid boundary) and not just (e.g. large park or medium park).

## Execution

Finally, to outline an example to run a set of test cases, we make sure that

- Park data must be in the Azure container specified by the JSON scripts
- Naming convention in JSON scripts must resemble park data in Azure
- CSV file data must resemble the parks filepaths, etc.
- Docker must be set up and running
  - `az acr login -n pawsregistry`
  - `docker build . -t pawspredict`
  - `docker run -p 8020:8020 pawspredict`

  For more information on Docker and Azure, check the deployment guide here.

Additional dependencies for these modules (specifically the ones pertaining to POST request testing) are: `os.system`, `dotenv.dotenv_values`, `time.sleep`, `requests`, `json`, `csv`, `geopandas`, `matplotlib .pyplot`. So far, the testing modules include:

`shape_check()`, `get_test_cases()`, `send_post_request()`, and `setup_post_request()`

And more details can be found in the Appendix.

## Next steps

Continuation of data creation following the structure outlined (e.g. raster files and UTM projections especially). Modifications to the structure to better reflect purpose of testing while maintaining space complexity.

In addition, verifying more aspects of shapefiles might be fruitful in understanding the validity of certain test cases, and as a result, develop a more extensive implementation of error handling.

In detail, raster files are digital images that are made up of a grid of pixels, with each pixel representing a single color or shade. They are typically used for storing photographs or images that contain a lot of detail, such as scanned documents or digital photographs. Raster images are resolution dependent, which means that the quality of the image depends on the number of pixels per inch (PPI). The higher the PPI, the higher the quality of the image, but the larger the file size. Raster files can be saved in a variety of formats, including JPEG, GIF, and PNG. This is relevant to the project, as creation of distinct raster files can alter the predictions made.

Furthermore, UTM projections preserve the shapes of small areas, allow for the accurate measurement of distance and direction, and are simple to use and incorporate into computerized mapping systems. However, they are not suitable for global maps because they are not equal area projections and become increasingly distorted at high latitudes. This makes having a distinct set of UTM projections in shapefiles imperative to determine how the PAWS system should address such distortions.

Another step that can be taking is setting up the container name itself within the modules (as input) such that the JSON scripts will no longer need to be altered based on the location of the data set (group of test cases). For example, the scripts are under the aperearojascollegeharvardedu container, and were one to move all the data to a different one, the entire set of scripts must be altered to reflect such change. Therefore, this could be a useful refactor.

## Appendix

Testing modules include: `shape_check`, `get_test_cases`, `send_post_request`, and `setup_post_request`
.

```
from dotenv import dotenv_values
from time import sleep
from os import system
import requests, json, csv
import geopandas as gpd
import matplotlib.pyplot as plt
from shapely.validation import explain_validity

# Take CSV file with headers {Park,Label,Filepath,Exp_Valid}
# Returns validity of shapefiles and plots those that unequal expecation
def shape_check(filename):

    with open(filename, mode='r') as file:
        shapefiles = csv.DictReader(file)
        shapes = dict()

        for shapefile in shapefiles:
            boolean = True if shapefile['Exp_Valid'] == 'True' else False
            dataframe = gpd.read_file(shapefile['Filepath'])
            shapes[shapefile['Park']] = dict(Label=shapefile['Label'], df=dataframe,
                Exp_Valid=boolean)

    for shp in shapes:
        if (shapes[shp]['df'].is_valid[0] != shapes[shp]['Exp_Valid']):

            shapes[shp]['df']['validity'] = shapes[shp]['df'].apply(lambda row:
                explain_validity(row.geometry), axis=1)

            shapes[shp]['df'].plot(color='#98b8eb')
            plt.xlabel(str(shapes[shp]['Label']) + " should be " + str(shapes[shp]['
                Exp_Valid']))
            plt.savefig("figures/" + str(shapes[shp]['Label']) + ".png")

            print(shapes[shp]['Label'], "\n", shapes[shp]['df'], "\n", "Should be:",
                shapes[shp]['Exp_Valid'], "\n")


# Helper function for setup_post_request()
# input csv w/ headers = {test_case,json_script_filepath,should_run,run_now}
def get_test_cases(file_path) -> dict:

    tests = dict()

    with open(file_path, mode='r') as file:

        test_cases = csv.DictReader(file)

        for test in test_cases:

```

```python
            should_run = True if test['should_run'] == 'yes' else False
            run_now_decision = True if test['run_now_decision'] == 'yes' else False

            tests[test['test_case']] = dict(json=test['json_script_filepath'],should_run=
                should_run,run_now_decision=run_now_decision)

    return tests

# helper function for setup_post_request()
def send_post_request(addr, admin_key, key_value, json_filepath):

    with open(json_filepath) as json_object:
        json_script = json.load(json_object)

    return requests.post(addr, headers={ admin_key : key_value }, json = json_script)

# setup for individual post request
def setup_post_request(json_script, env, server='local_server'):

    config = dotenv_values(env)
    try:
        admin_key = config['Ocp-Apim']
    except:
        print("Wrong environment path or 'Ocp-Apim' is not part of it.")

    if server == 'local_server':
        addr = config['LOCAL_ADDR']
        key_value = config['LOCAL_KEY']

    elif server == 'testing_server':
        addr = config['TEST_ADDR']
        key_value = config['TEST_KEY']

    else:
        print("In the 3rd field, please indicate whether to deploy locally by specifying
            'local_server', or alternatively, the testing server denoted by '
            testing_server'")
        return ValueError

    try:
        response = send_post_request(addr, admin_key, key_value, json_script)
        response.raise_for_status()
    except requests.exceptions.HTTPError as e:
        print(e.response.text)

    return response

# Synthesize testing for all test cases in a CSV file
# Input: CSV file {test_case,json_script_filepath,should_run,run_now_decision}
def parks_testing_start(csv_filepath, env, server='local_server'):

    system('clear')
    test_cases = get_test_cases(csv_filepath)

    for case in test_cases:

        if test_cases[case]['run_now_decision'] is True:

            post_response = setup_post_request(test_cases[case]['json'], env, server)

            print(post_response.__dict__)
            print(case, 'POST request sent; execution & sleep for 20 seconds;')

            if test_cases[case]['should_run']:
                print("This test case should run succesfully.\n")
            else:
                print("This test case should fail and terminate.\n")

            sleep(20)

if __name__ == "__main__":

    parks_testing_start('tests.csv', '.env', 'local_server')
```

Currently, abstraction to container-name for the JSON script into the Python programs implemented would be helpful. For now, all the data is located in the current container name specified in the following JSON script. A transfer of data to another container would require the change of this to all of the JSON scripts. All scripts test park data has a resolution of 1000.

```
{
    "container_name": "aperearojascollegeharvardedu",
    "run_id": "medium_park",
    "test_case": "only_boundary",
    "spatial_resolution": 1000,
  "patrol_observations": {
      "start_date": "01/1/2015",
      "end_date": "12/31/2019",
      "file_name": "patrol_observations_basic.csv",
        "transport_type": [],
        "mandate_type": []
    },
    "raster_files": {
       "elevation_file_name": "",
       "elevation_layer_name": "",
       "landcover_file_name": "",
       "landcover_layer_name": "",
       "additional_raster_files": []
    },
  "geo_feature_shape_files": {
      "boundary_file_name": "boundary.zip",
      "boundary_layer_name": "CA",
      "additional_shape_files": []
    },
    "model_experimentation": {
        "train_start_year": 2019,
    "train_end_year": 2019,
    "temporal_training_resolution_month_count": 2
    },
    "model_forecasting": {
      "classifier_model": "decision_tree",
        "start_year": 2020,
        "end_year": 2020
    },
  "illegal_activity_class_mappings": []
}
```