Ronald Pacheco

State Farm Model Deployment

In this project, we used Flask to create our API, and Docker to package it:

- Flask is a popular Python framework for making web APIs. This will be the core of our API — where we import our ML model and write a function to predict and return a JSON response.
- Docker is a great way to make the API easy to deploy on any server. It is easily customizable to run with any configuration. Moreover, you can combine multiple instances of the docker container when you want to scale up your API.

Project structure

```
root
    |- docker-compose.yml
    |- run_api.sh
    |- README.md
    |- .gitignore
    |- api
        |- __init__.py
        |- app.py
        |- data_prep.py
        |- requirements.txt
        |- Dockerfile
        |- unit_test.py
        |- endpoints
            |- __init__.py
            |- prediction.py
        |- models
            |- model.pkl
            |- imputer.pkl
            |- scaler.pkl
        |- test
            |- __init__.py
            |- test_prediction.py
```

root

Houses the following files:

- ‘docker-compose.yml’ is the compose file that documents and configures all of the application's service dependencies.
- ‘run_api.sh’ is the shell script that runs the ‘.yml’ file.
- ‘README.md’ is exactly what you are reading right now.
- ‘.gitignore’ is the .gitignore file for the GitHub repository.

api

- 'app.py' is the main script where all our blueprints are imported to that runs our app at port 1313.
- 'data_prep.py' is the script referenced by 'prediction.py' before running the data through the model. This file uses 'scaler.pkl' and 'imputer.pkl' to scale and impute the data, as well as create dummy variables and returns only those needed by the model.
- 'requirements.txt' is the file with all the packages needed to run the docker container.
- 'Dockerfile' is the dockerfile to create the docker image.
- 'unit_test.py' is the script that runs our tests.

Endpoints (blueprints)

Flask Blueprints are a good way to modularize an API for scalability, since stuffing it all into app.py is less than ideal.

Step 1: Creating endpoint

- We need to initialize our API as a blueprint instead of a Flask app. I chose to call it "prediction._api".

Step 2: Register the endpoint in app.py

- Now that we have the prediction.py file to deal with the endpoint code, we can simply import the blueprint and use it in our Flask app.

Models

Here we have the pickle files exported from the Jupyter Notebook, 'imputer.pkl' and 'scaler.pkl' were also exported to be used in 'data_prep.py'.

Test (unit testing)

Step 1: Writing tests

- We call register_blueprint() and pass in the prediction_api blueprint we created in prediciton.py
- Following this, we will store the app.test_client() in a local tester variable called "tester". This will give us access to the API, as if we are hitting it with actual traffic.
- Now, we write the test functions, which are test_predict_single, test_predict_triple, and test_predict_missing. Here, we use tester.post() to do a POST request on the '/prediction' endpoint. The data and content_type are determined by what kind of request your API can handle, in this case JSON. The returned JSON can be accessed using the response.get_data() method as text that I can then print, or write assert statements on.
    - NOTE: In these tests, we are only testing the API, not the model. The model has already been validated and it is assumed the predictions are accurate. Therefore, the assert statements are testing that the status code returned is a success (which should be 200) for this structure of input JSON, and that the data in the response is not None (i.e. null).

Step 2: Write a script to run all tests for all endpoints (unit_test.py)

Once all the testing scripts are ready, we wrote a script to run all the tests for all endpoints.

Docker repo: https://hub.docker.com/repository/docker/ronaldpacheco/mle_state_farm_app/general