Machine Learning for Business Applications
46-887

Carnegie Mellon University
Tepper School of Business

Project guide

This document is a high level how-to guide that your team may find helpful to build certain parts of your POC (Proof Of Concept) ML (Machine Learning) system. In particular, this guide may come in handy for your 3rd and 4th team project update.

Preliminaries

In this guide, it is assumed that we have already trained a ML model. In what follows we use the the simple linear regression model described in this blog post. Note that this is only done for simplicity, since this guide focuses on building an API, adding tests, and performing a load test. This guide does not focus on training a ML model using best practices. On the other hand, your team will have to train and evaluate a ML model following best practices as part of your project.

The model predicts the unit price of real estate properties in New Taipei City, Taiwan (measured in 10,000 New Taiwan Dollars / Ping) based on their distance from the closest Mass Rapid Transit station (measured in meters).

If you are trying to replicate the contents of this guide, you will need to install the following Python libraries:

- FastAPI
- HTTPX
- Locust
- joblib
- numpy
- pytest
- pytest-mock
- scikit-learn
- uvicorn

Building an API with FastAPI

First, we create a dedicated folder for our application. We name the folder `app`.

Next, we create a new file - `app.py` - in the `app` folder that we just created.

In the `app`, we also include the model artifact. In this case, this is a scikit-learn model serialized using `joblib`. The file name of the artifact in this example is `model.joblib`.

At this point, the `app` folder looks like this:

```
app
├── app.py
└── model.joblib
```

We can now use FastAPI to build an API for our POC ML system by adding the following contents to the `app.py` file that we just created:

```python
from fastapi import FastAPI
import joblib
import numpy as np
from pydantic import BaseModel, PositiveFloat

# See
# https://www.datacaptains.com/blog/building-and-load-testing-a-machine-learning-service
# for more info on this model.
ML_MODEL = joblib.load("./model.joblib")

# FastAPI.
api_title = "RealEstateUnitPriceApp"
api_description = """
RealEstateUnitPriceApp allows you to predict the unit price of real estate
in New Taipei City, Taiwan on the basis of the distance of the property from
the closest station of the Mass Rapid Transit (MRT).
"""
api = FastAPI(title=api_title, description=api_description)


class Distance(BaseModel):
    """
    Data model for distance.
    """

    Distance: PositiveFloat


class UnitPrice(BaseModel):
    """
    Data model for unit price.
    """

    # Our simple linear regression model does not make only positive
    # predictions. So, technically, we can only guarantee that we will return
    # a float. We can't guarantee it will be a PositiveFloat.
    UnitPrice: float


def predict(log_distance: float) -> float:
    """
```

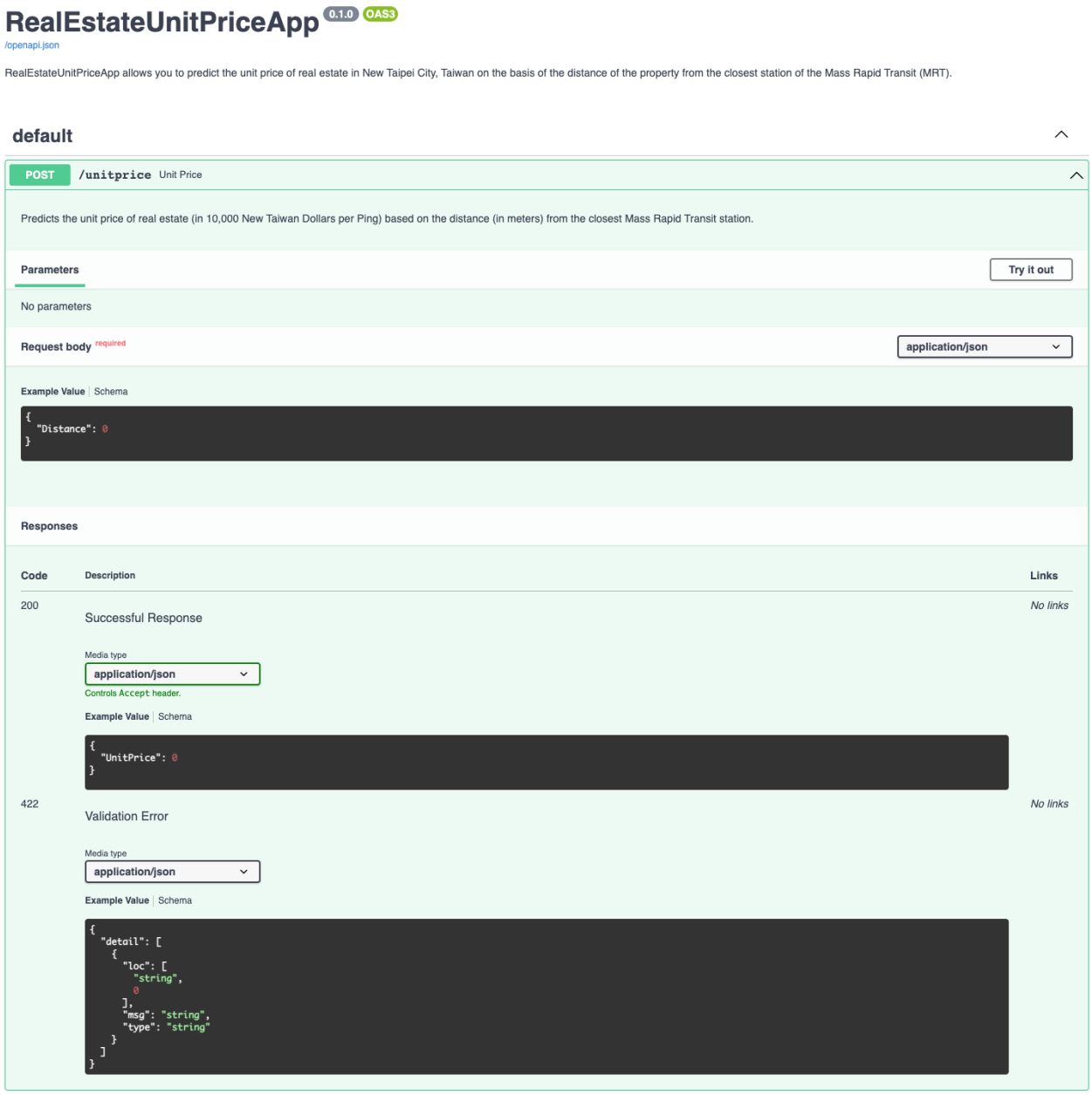
```
    Utility to make predictions from the ML model.
    """
    return ML_MODEL.predict(np.array([[log_distance]]))[0]


@api.post("/unitprice", response_model=UnitPrice)
def unit_price(distance: Distance) -> UnitPrice:
    """
    Predicts the unit price of real estate (in 10,000 New Taiwan Dollars
    per Ping) based on the distance (in meters) from the closest Mass
    Rapid Transit station.
    """
    log_distance = np.log(distance.Distance)
    unit_price = predict(log_distance)
    return UnitPrice(UnitPrice=unit_price)
```

We can then run our POC ML system locally using `uvicorn`:

```
cd app
uvicorn app:api
```

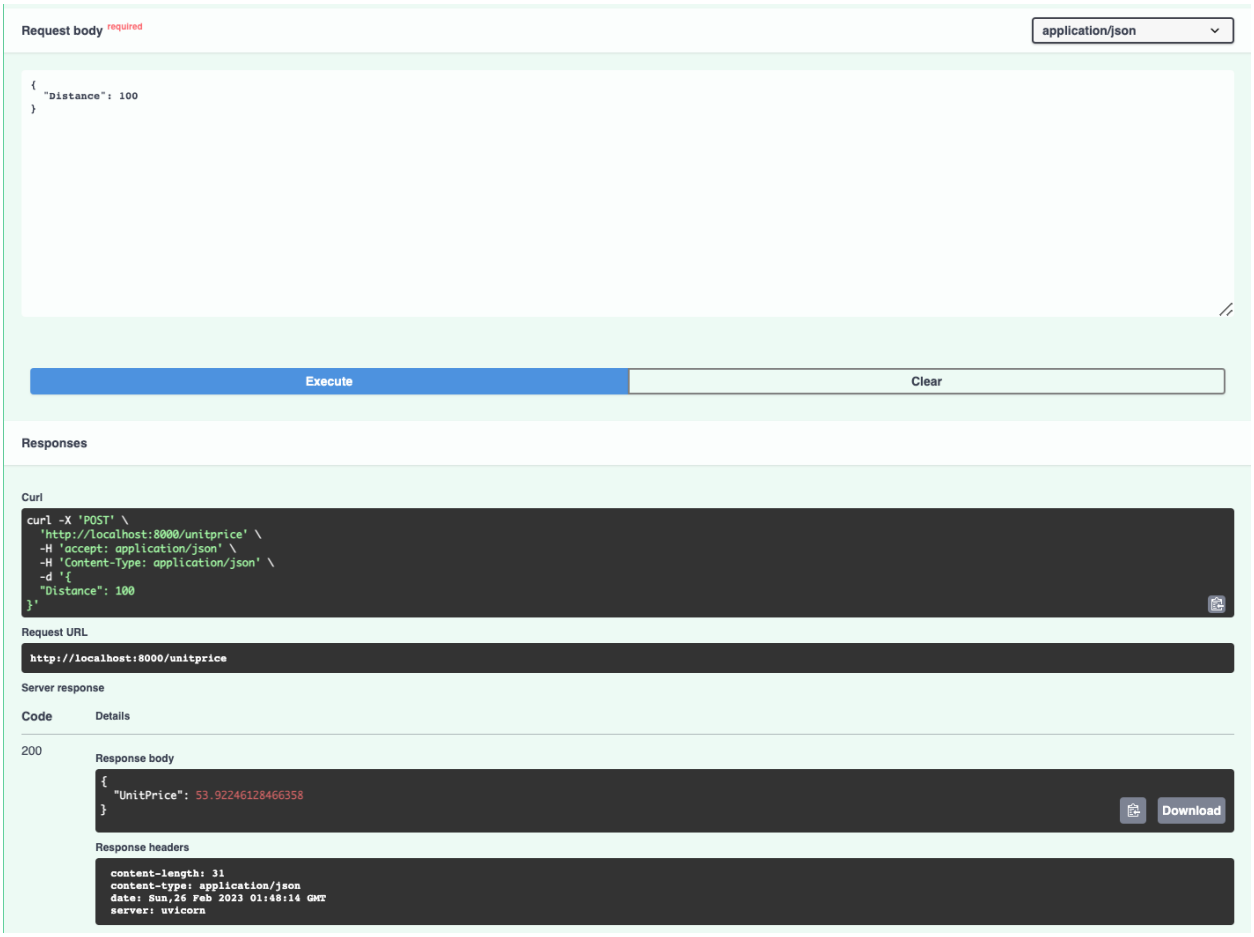
By default, our ML application is served at port 8000. We can then navigate with our browser (e.g., Chrome) to `http://localhost:8000/docs`. There, we find the Swagger UI of our API (autogenerated by FastAPI from on our code).



The Swagger UI allows us to interact with our POC ML system by means of its API:

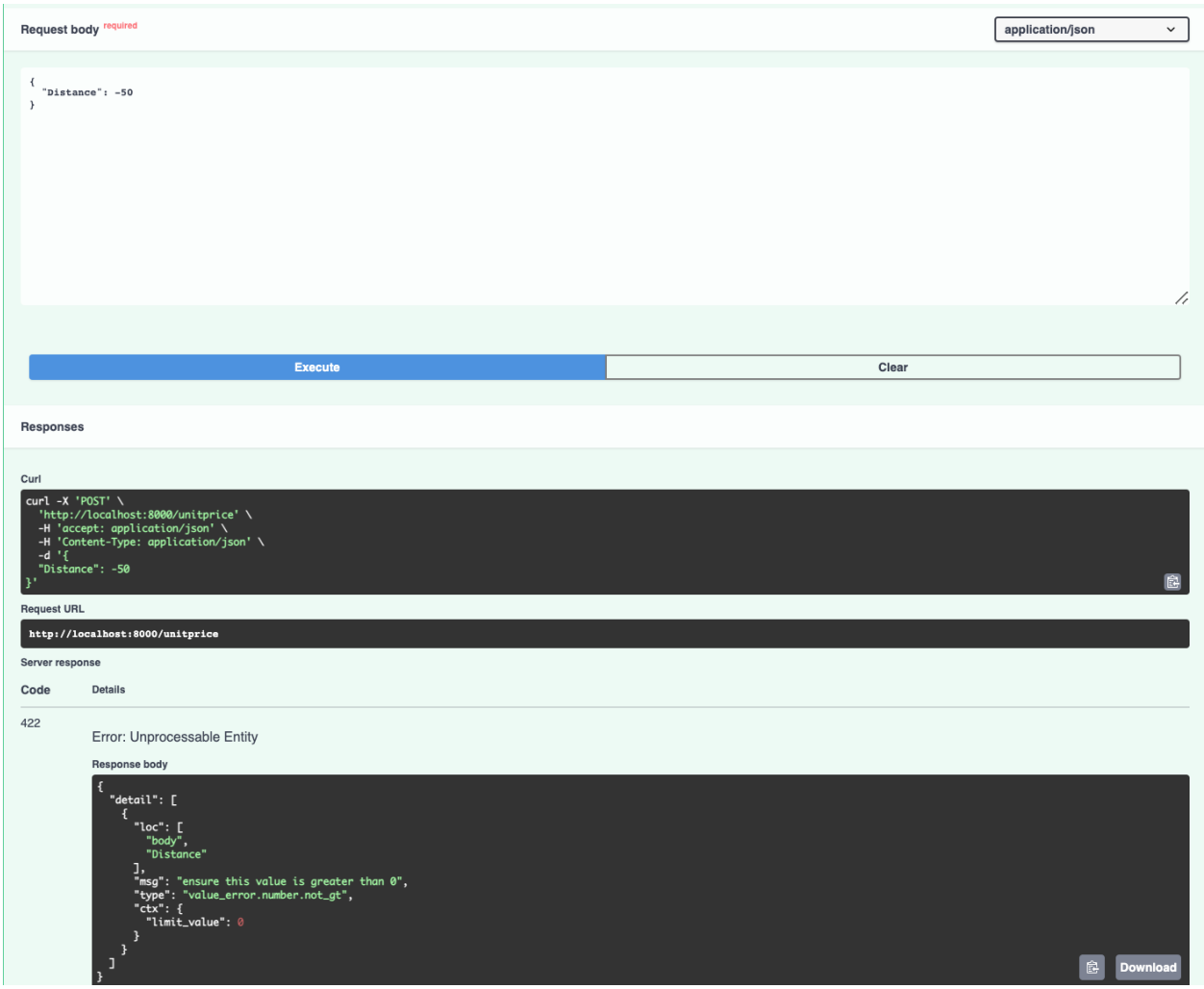
- Click the "Try it out" button.
- Enter a value for the distance (e.g., 100). This represents the distance in meters between a real estate property for which we want to predict the unit price and the closest Mass Rapid Transit station.
- Click on "Execute". This will make the API call.
- Look further down in the UI: you should see a section that reports the "Server response" together with the HTTP status code (200 indicates success), some response headers, and the response body with the predicted unit price in 10,000 New Taiwan Dollars per Ping. It will look something like this:

```
{
    "UnitPrice": 53.92246128466358
}
```



Note that if you were some other software component rather than a human, you would interact with this API programmatically e.g., by using the `curl` command that you see in the "Responses" section right under the "Execute" button in the Swagger UI.

Now, for fun, let's try to make this fail. What happens if you make another request and pass a negative value for the distance (e.g., -50)? Because we require distance to be a positive number (see line 26 in the code above), we receive a response with HTTP status code 422 which informs us that we our request contains an "unprocessable entity".



FastAPI gives us input validation for free by means of the Pydantic library.

Add tests

We can add a `test_app.py` file for our tests in the `app` folder that we previously created. Our `app` folder then looks like this:

```
app
├── app.py
├── model.joblib
└── test_app.py
```

We can follow the relevant section of the FastAPI documentation to add a test for our API. Here are the contents of our `test_app.py` file:

```
from fastapi.testclient import TestClient

from app import api, UnitPrice
```

```
4
5  client = TestClient(api)
6
7
8  def test_unit_price(mocker):
9      # Arrange.
10     request_body = {"Distance": 123.4}
11     mocker.patch("app.predict", return_value=56.7)  # this requires pytest-mock
12
13     # Act.
14     response = client.post("/unitprice", json=request_body)
15
16     # Assert.
17     assert response.status_code == 200
18     assert response.json() == UnitPrice(UnitPrice=56.7).model_dump()
```

In the test, we:

1. Set the request body (line 10).
2. "Mock" the utility function that is responsible to obtain predictions from the ML model (line 11). Basically, we override it - in the context of the test - with a version that always returns 56.7. Mocking allows us to isolate the behaviour of the object under test (the `unitprice` endpoint of the API) by simulating the behaviour of other objects (in this case the `predict` utility function) on which the object under test depends.
3. Make the request (line 14).
4. Verify that the response indicates success i.e., HTTP status code 200 (line 17) and that the response body is exactly what we expect (line 18).

The arrange-act-assert pattern is a very effective general pattern that can be used to write tests. If you are curious, you can read more about it in the pytest documentation for example.

To execute our tests (in this case only one in a single test file), we can run the `pytest` command from the `app` folder. The output is:

```
==================================== test session starts ====================================
platform darwin -- Python 3.11.1, pytest-7.2.1, pluggy-1.0.0
rootdir: /Users/mattiaciollaro/git_repositories/tepper-machine-learning-for-business-applications/project-guide/app
plugins: mock-3.10.0, anyio-3.6.2
collected 1 item

test_app.py .                                                                        [100%]

===================================== 1 passed in 2.42s =====================================
```

The test passes!

For fun, let's now make the test fail. For example, here is what happens if we change our `unitprice` API endpoint to this:

```
1  from fastapi import FastAPI, Response, status
2
3  # ... our previous code here ...
4
5  # Modified API endpoint.
6  @api.post("/unitprice", response_model=UnitPrice)
7  def unit_price(distance: Distance, response: Response) -> UnitPrice:
8      """
9      Predicts the unit price of real estate (in 10,000 New Taiwan Dollars
```

```
    per Ping) based on the distance (in meters) from the closest Mass
    Rapid Transit station.
    """
    log_distance = np.log(distance.Distance)
    unit_price = predict(log_distance)
    response.status_code = status.HTTP_400_BAD_REQUEST
    return UnitPrice(UnitPrice=unit_price)

```

Here, we are deliberately creating a bug in our implementation by forcing the HTTP status code of the response to always be 400 (line 10). However, as suggested by our test logic, on success we expect to receive the HTTP status code 200.

Let's rerun the test to see if it helps us catch that we have introduced a bug in our code. Here is what we get now:

```
============================================= test session starts ==============================================
platform darwin -- Python 3.11.1, pytest-7.2.1, pluggy-1.0.0
rootdir: /Users/mattiaciollaro/git_repositories/tepper-machine-learning-for-business-applications/project-guide/app
plugins: mock-3.10.0, anyio-3.6.2
collected 1 item

test_app.py F                                                                                          [100%]

=================================================== FAILURES ===================================================
_____________________________________________ test_unit_price _____________________________________________

mocker = <pytest_mock.plugin.MockerFixture object at 0x11cd46e10>

    def test_unit_price(mocker):
        # Arrange.
        request_body = {"Distance": 123.4}
        mocker.patch("app._predict", return_value=56.7) # this requires pytest-mock

        # Act.
        response = client.post("/unitprice", json=request_body)

        # Assert.
>       assert response.status_code == 200
E       assert 400 == 200
E        +  where 400 = <Response [400 Bad Request]>.status_code

test_app.py:16: AssertionError
=========================================== short test summary info ============================================
FAILED test_app.py::test_unit_price - assert 400 == 200
============================================== 1 failed in 2.76s ===============================================
```

As expected, the test now fails and helps us detect that we have introduced some problem in our code and that our API is no longer operating as we originally intended.

Performing a simple load test with Locust

In a real-world production scenario, our POC ML system would likely operate on multiple servers and not on our laptop. We might then decide to configure the pool of available servers to "scale out" dynamically based on the volume of requests that our system receives at any given time: additional servers capable of handling prediction requests would be added dynamically when the current pool of servers is no longer sufficient to satisfy the requests that it receives.

In a real-world production scenario, the purpose of a load test would then be to "generate pressure" on our system by simulating a large volume of requests. The goal then is to verify that our system is configured in such a way that all requests can be handled successfully and without delays even under heavy load.

Executing a load test on our laptop is not particularly realistic, but it is still a good learning opportunity. Besides the fact that in this situation we are using a single server (our laptop), nothing changes conceptually from how we would perform a load test in a real-world production scenario.

In the context of this guide, we will use the Locust library to define and execute a load test on our POC ML system. A useful starting point is the Locust quickstart guide.

First, we need to write a `locustfile.py` file that contains the logic of our load test. We add this file to the `app` folder that we previously created. Now, the `app` folder contains the following:

```
app
├── app.py
├── locustfile.py
├── model.joblib
└── test_app.py
```

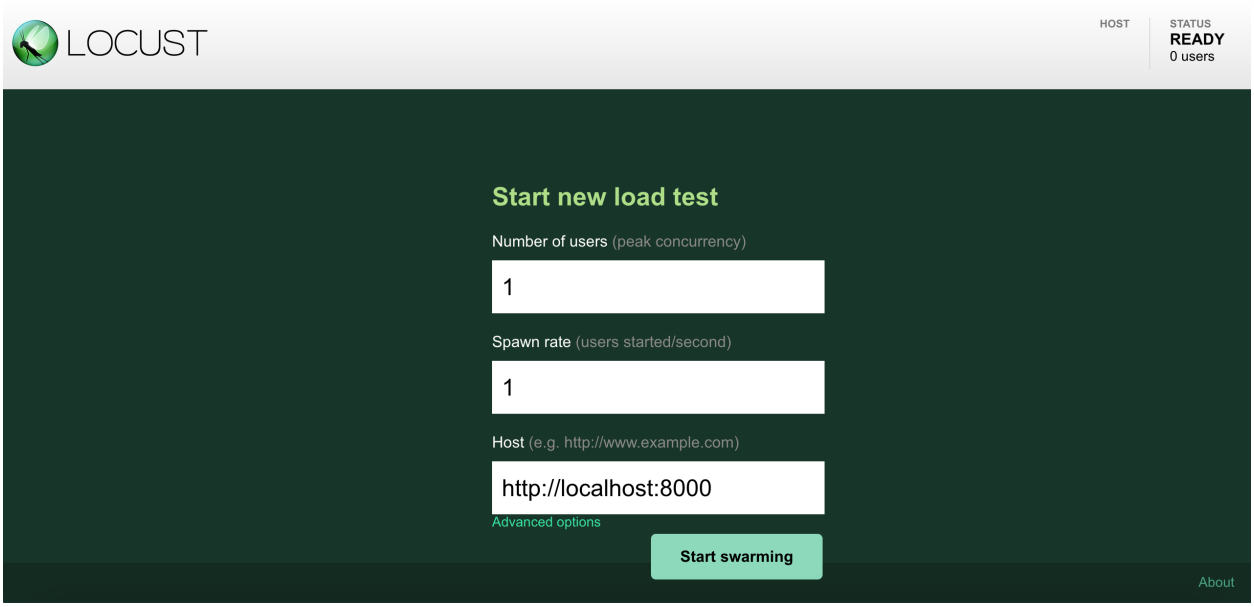
Here are the contents of `locustfile.py`:

```
1  import random
2
3  from locust import HttpUser, constant_pacing, task
4
5
6  class UnitPriceUser(HttpUser):
7      host = "http://localhost:8000"
8      wait_time = constant_pacing(1)
9
10     @task
11     def unit_price(self):
12         self.client.post(
13             "/unitprice",
14             json={"Distance": 1.0 + random.random() * 6499.0},
15         )
```

In this case, we simulate a user of our system: `UnitPriceUser`. For ML systems, this "user" likely represents some other software component that is trying to get predictions from it and not an actual person. For this user, we specify:

- The URL and the port at which our POC ML system is "listening" for incoming prediction requests via the API that we built using FastAPI (line 7).
- What the user does (lines 11-15): it makes requests against the `unitprice` endpoint of our API, passing a value for the distance that is a random number between 1 and 6500 meters.
- How often the user makes requests: in particular, this user makes approximately 1 request every second (line 8).

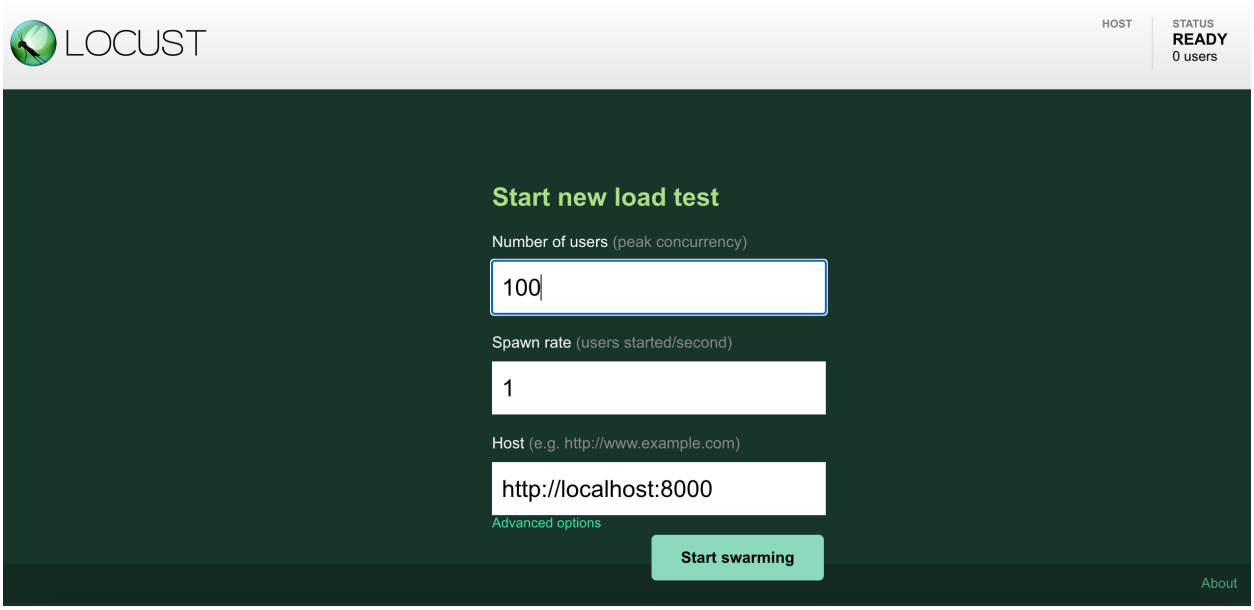
With this in place, we can now launch a load test where multiple clones of `UnitPriceUser` start making prediction requests to our POC ML system at approximately the same time. To do so, we can execute the `locust` command in a new terminal window. This will launch Locust and serve its UI at `http://localhost:8089` by default. We can visit this URL with our browser:



For the sake of the example, let's say that we want to test our POC ML system under the following conditions:

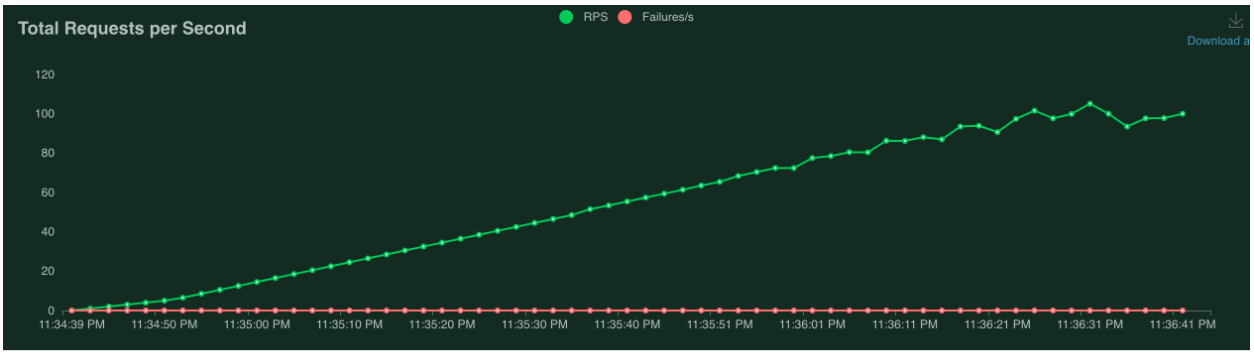
- The peak load is 100 `UnitPriceUser`s making requests at approximately the same time at a rate of 1 request per second.
- We start the test with 1 user and we add 1 user every second until we hit the target load of 100 users.

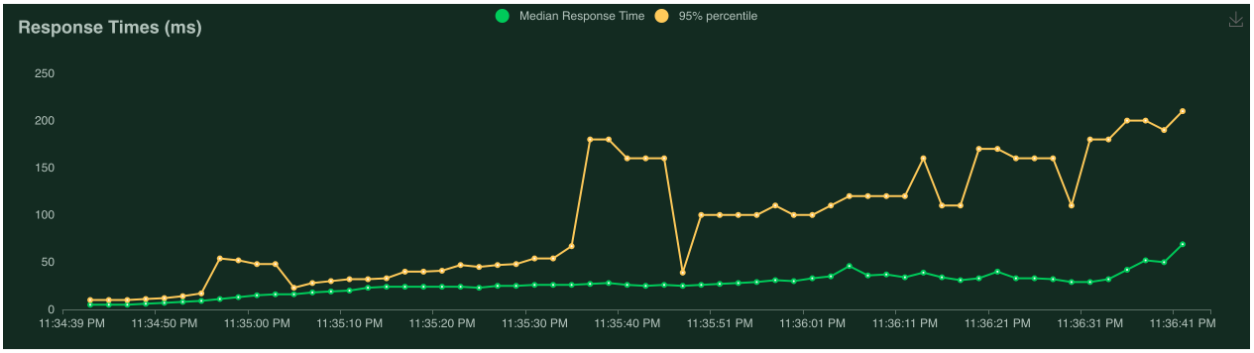
To accomplish this, it's enough to change the value of the "Number of users (peak concurrency)" parameter from 1 to 100.

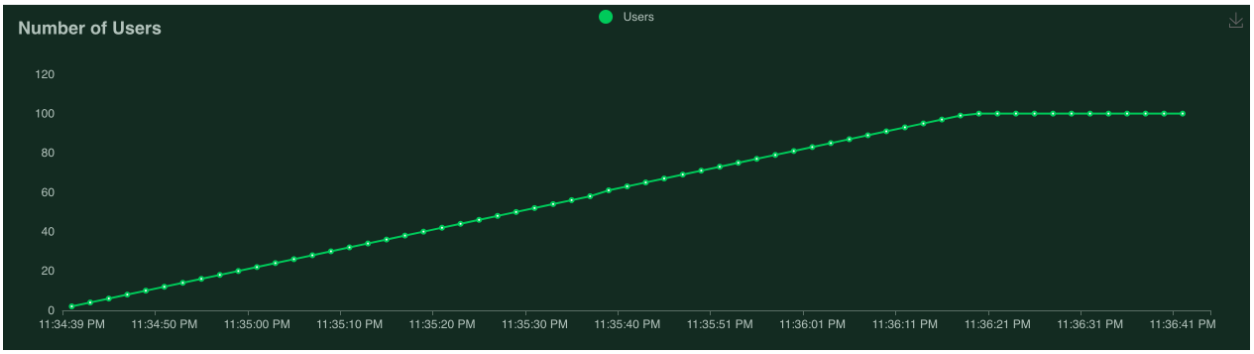


Next, it's time to launch the load test by clicking on the "Start swarming" button.

Here are the results:







The bottom graph shows how many users were making prediction requests to our POC ML system over time: we started with a single user and added 1 more user every second, until we reached 100 users (the test was then manually interrupted).

The top graph shows the number of requests per second (RPS) received by our POC ML system over time. Because every user was configured to make 1 RPS on average, the test started with approximately 1 RPS and ended with approximately 100 RPS.

The central graph is the most interesting one. It shows us how the median latency (green line) and the P95 latency (yellow line) increased over time as the pressure on our POC ML system increased. Beyond approximately 50-60 concurrent users making 1 RPS each, the system performance started deteriorating: the P95 latency raised from approximately 50ms to somewhere between 100ms and 200ms.

We can also get a report on the load test with some interesting statistics:

Statistics | Charts | Failures | Exceptions | Current ratio | Download Data

Type	Name	# Requests	# Fails	Median (ms)	90%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/unitprice	7550	0	30	100	220	46	3	303	32	94.3	0
	Aggregated	**7550**	**0**	**30**	**100**	**220**	**46**	**3**	**303**	**32**	**94.3**	**0**

About

From the report, we see that:

- The load test involved POST requests to the `unitprice` endpoint of our API (the only endpoint of our API in this case).
- A total of 7550 requests were made over the entire duration of the load test.
- None of the requests resulted in an error ("# Fails" is 0).
- Across these 7550 request, the median latency of our POC ML system was 30ms, the average latency was 46ms, the P90 latency was 100ms, and the P99 latency was 220ms. No requests were fulfilled in less than 3ms and our POC ML system took 303ms to serve at least 1 of these requests.

Exercises left to the reader

- Refactor the code in the `app.py` file to load the `ML_MODEL` using a lifespan function.
- Modify the code in the `app.py` file to leverage the ability of the `ML_MODEL.predict` method to perform vectorized calculations. How would you then implement another API operation such that the user can pass an array of distances (corresponding to multiple real estate properties) and get in return an array of unit price predictions?
- Extend the API tests to also cover other cases beyond the "happy path".