Python FastAPI REST APIs - WOW - It's FAST!

> by Thom Ives

FastAPI works great for Python based REST APIs. It has and continues to help greatly in my work at my company. I hope it will help all of you just as much or more. I think that most of you can leverage from the simple API development that I show in here and take it much further. I wanted to share in detail how to release a basic model serving REST API using FastAPI. There are many good tutorials on how to get started with FastAPI, but I have not found one with the emphasis and approach that I give in here.

FastAPI has great tutorials. You can start at the main FastAPI Tiangolo pages, and, from those, you can go quite far. I hope that you will check them out. I went through the main one and then through all the tutorials. If you are coming from Flask to FastAPI like I did, it may take some time to become accustomed to FastAPI. However, once you are a few steps in, you'll greatly appreciate FastAPI's automatically created documentation pages for the APIs that you write. These documentation pages can also be used to test your APIs. That's right. You don't need to immediately write your own HTTP method call scripts. Nor do you need to use something like Postman. It's all built-in.

Finally, this tutorial only creates one post method. I have other examples of FastAPI APIs that I've made, and I will share those with you soon too. Those use multiple HTTP methods.

Prerequisites

I recommend that we first create, or activate, our preferred Python virtual environment. Once you are in there, you will want to do the following `pip` installations. The first one, I trust, is obvious. The second one is the package that will serve your API. Note that it's a different package than the one used by Flask. The third and last is simply the quickest way to obtain all the FastAPI goodies available.

In [ ]:
```
!pip install fastapi
```

In [ ]:
```
!pip install "uvicorn[standard]"
```

In [ ]:
```
!pip install "fastapi[all]"
```

The REST API Code

I'll cover and comment on the simple code base first. I can't say what's best, but I recommend going through the tutorials at FastAPI first at least a little bit. However, if a good portion of this makes sense to you, it would be OK to start with this code. Regardless of when you look at FastAPI's learning materials, **PLEASE** do look at them. I do not regard this overview of FastAPI as a tutorial worthy of use in isolation. PLEASE study other sources too. However, even IF I had the best FastAPI tutorial around, I'd still encourage you to look at other material to gain a greater understanding of it.

This tutorial is quite basic if you already know modeling. Even though this is a basic starting point, you can serve MANY basic models using this approach.

I do at least recommend that you become familiar with decorators through some web searching if they are foreign to you. An example of a decorator below is `@app.post("/run_model")`. I like to think of decorators as a Pythonic elegant way to have Python wrap the functions below the decorator in a standard well defined way. They will seem magic until you study how to create your own. To me, they are one of those elegant Pythonic things like context managers. I very much appreciate them.

Imports

We will need **pickle** to store our model after we train it. Of course we will want to import **FastAPI** from **fastapi**. The **BaseModel** from Pydantic is great. I won't say much about

Pydantic here, but PLEASE study it more. It's a major helpful set of data model classes that help FastAPI work like a charm. I've written APIs with and without Pydantic, and I **now very much prefer to use Pydantic**. I've also noticed that FastAPI practitioners love using the Pydantic data models. We will need a type of list in the BaseModel that we will create for the data that we pass in. This is why we've also imported **List** from the **typing** module. This will become more clear later in case you're feeling confused.

In [ ]:
```
import pickle

from fastapi import FastAPI
from pydantic import BaseModel
from typing import List
```

Necessary Declarations

Next, we declare a data class using Pydantic's BaseModel.

In [ ]:
```
class Features(BaseModel):
    note: str
    data: List[float]
```

Note that we will be loading a saved model file. The code to create the model and save it will be shown below soon.

IF you've done some Python based API work in the past, the `app = FastAPI()` statement should be expected. It is essential if this is completely new to you. We instantiate a class instance of FastAPI named app.

In [ ]:
```
model_file_name = "Linear_Regression_Model.pkl"
with open(model_file_name, 'rb') as f:
    mod_LR = pickle.load(f)

app = FastAPI()
```

When building APIs, we normally use the following specific HTTP methods:

- POST: to add complete new records to our data.
- GET: to read our existing data records.
- PUT: to update existing data.
- DELETE: to delete a specific record from our data.

In OpenAPI, each of the HTTP methods are also called an "operation". There are actually more methods than these 4, but these 4 are the most common.

Consider the next code block below. The first API endpoint, or method (like a method for a python class or like a function from a python module) is started with a decorator. That decorator is `@app.post("/run_model")`. For this basic model delivery API that takes inputs and provides predictions, we will only use this one method - a POST operation. I do have tutorials on how to use all of these basic operations. I will share that with you another time.

Consider the next line of code in the code block below. Our POST method, called `run_model` is expecting a features input of type Features defined previously (above) using Pydantic. We'll see shortly how to enter this and make calls for it from both the automatically generated documentation for this API and from python scripts.

For now, notice that:

1. we pass in a `note` that is a string that is only used for documentation about the prediction,
2. we pass in the list of three float values for the features and use them in the call to the prediction algorithm,
3. we load the prediction into the return data structure along with the note,

Note that for our case, `"/run_model"` part of the decorator is known as a route. We don't really need it here, but I am including it, because, later, we might add GET, PUT, and DELETE operations to this overall API too, and the `"/run_model"` route will help to distinguish this operation from the other future ones.

In [ ]:
```
@app.post("/run_model")
async def run_model(features: Features):
    print( {
            'note': features.note,
            'list': features.data
        } )

    Y_pred = mod_LR.predict([features.data])
    print(Y_pred)

    return  {
                "note": features.note,
                "value": Y_pred[0, 0]
            }
```

The entire API code is shown below. I've named this file Run_Model_API.py.

In [ ]:
```
# Run_Model_API.py
import pickle

from fastapi import FastAPI
from pydantic import BaseModel
from typing import List


class Features(BaseModel):
    note: str
    data: List[float]
```

```
model_file_name = "Linear_Regression_Model.pkl"
with open(model_file_name, 'rb') as f:
    mod_LR = pickle.load(f)

app = FastAPI()


@app.post("/run_model")
async def run_model(features: Features):
    print( {
            'note': features.note,
            'list': features.data
        } )

    Y_pred = mod_LR.predict([features.data])
    print(Y_pred)

    return  {
                "note": features.note,
                "value": Y_pred[0, 0]
            }
```

Training A Model With Fake Data And Saving It And The Test Data

Let's look at the code to create the fake data and train the model and save the trained model. Please use the comments in the code to help you understand it. Play with each line to understand this code if you need to.

In [ ]:
```
# Basic_Fake_Data_Model_Train_And_Store.py
import numpy as np
import pandas as pd
import sklearn.model_selection as sklms
import pickle

from sklearn.linear_model import LinearRegression

# Create the fake feature data
X1 = np.random.uniform(0, 1, 1000).reshape(-1, 1)
X2 = np.random.uniform(0, 1, 1000).reshape(-1, 1)
X3 = np.random.uniform(0, 1, 1000).reshape(-1, 1)

# Group the fake data features
X = np.hstack((X1, X2, X3))

# Use a known model to create the outputs and add noise
Y = 1.0 * X1 + 2.0 * X2 + 3.0 * X3
Y_peak_noise = np.max(Y) * 0.07
Y_noise = np.random.normal(0, Y_peak_noise, 1000).reshape(-1, 1)
Y += Y_noise
```

```
# Create train and test data
X_train, X_test, Y_train, Y_test = sklms.train_test_split(
    X, Y, test_size=0.2, random_state=42, shuffle=True)
print(f"X train shape is: {X_train.shape}")
print(f"X test shape is: {X_test.shape}")
print(f"Y train shape is: {Y_train.shape}")
print(f"Y test shape is: {Y_test.shape}")
print()


# Instantiate the model and train it
mod_LR = LinearRegression(fit_intercept=False, copy_X=True)
mod_LR.fit(X_train, Y_train)

# Check the scoring of the training
print(mod_LR.score(X_train, Y_train))
# Check that the coefficients values are close to the ones used above
print(mod_LR.coef_)

# Save the trained model to file
model_file_name = "Linear_Regression_Model.pkl"
with open(model_file_name, 'wb') as f:
    pickle.dump(mod_LR, f)

# Save the test data to file
data_file_name = "Test_Data.npz"
with open(data_file_name, 'wb') as f:
    np.save(f, np.hstack((X_test, Y_test)))

# Also save the X_test values to a csv file
X_test_df = pd.DataFrame(X_test, columns = ['X1','X2','X3'])
X_test_df.to_csv("X_Test_Data.csv")
```

Test The Loading Of The Saved Model And Data And Test The Model

The next code loads the saved model and the fake test data and measures the accuracy of the predictions. Again, please read the comments in the code to help you understand the code. Note that this code is testing in a Python script. We will do the same operations with our RestAPI in the Run_Model_API.py script once we have the server run it.

In [ ]:

```
# Basic_Fake_Data_Model_Test_And_Measure.py
import numpy as np
import sklearn.metrics as sklm
import math
import pickle


# A handy metric function I picked up from a course
def print_metrics(y_test, y_pred, n_params):
    ## First compute R^2 and the adjusted R^2
    ## Print the usual metrics and the R^2 values
    MSE = sklm.mean_squared_error(y_test, y_pred)
```

```
    RMSE = math.sqrt(sklm.mean_squared_error(y_test, y_pred))
    MAE = sklm.mean_absolute_error(y_test, y_pred)
    MedAE = sklm.median_absolute_error(y_test, y_pred)
    r2 = sklm.r2_score(y_test, y_pred)
    r2_adj = (r2 - (n_params - 1) /
        (y_test.shape[0] - n_params) * (1 - r2))

    print('Mean Square Error      = ' + str(MSE))
    print('Root Mean Square Error = ' + str(RMSE))
    print('Mean Absolute Error    = ' + str(MAE))
    print('Median Absolute Error  = ' + str(MedAE))
    print('R^2                    = ' + str(r2))
    print('Adjusted R^2           = ' + str(r2_adj))


# Load the model from the file
model_file_name = "Linear_Regression_Model.pkl"
with open(model_file_name, 'rb') as f:
    mod_LR = pickle.load(f)

# Load the test data
data_file_name = "Test_Data.npz"
with open(data_file_name, 'rb') as f:
    feature_label_data = np.load(f)

# Break the test data into features and labels (inputs and outputs)
X_test = feature_label_data[:, :-1]
Y_test = feature_label_data[:, -1]

# Perform predictions and measure the model's performance
Y_pred = mod_LR.predict(X_test)
print_metrics(Y_test, Y_pred, 4)
```

The Python Script That Will Call Our REST API

Now pretend we've launched our REST API using a server. We can now get values calling it from any routine that can reach that REST API's URL. We are running that API using the latest trained model as shown above. Now we will pretend to send new feature values (our X_test data) to our REST API through a Python Script. Basically, we pretend that this script is being used by one of our intended end users that we wrote this for. Again, please rely on the code comments to understand what is being done.

In [ ]:
```
# Call_Model_Run_API.py
import requests
import pandas as pd
import numpy as np
import json
import sklearn.metrics as sklm
import math


# The same print metrics function
```

```
def print_metrics(y_test, y_pred, n_params):
    ## First compute R^2 and the adjusted R^2
    ## Print the usual metrics and the R^2 values
    MSE = sklm.mean_squared_error(y_test, y_pred)
    RMSE = math.sqrt(sklm.mean_squared_error(y_test, y_pred))
    MAE = sklm.mean_absolute_error(y_test, y_pred)
    MedAE = sklm.median_absolute_error(y_test, y_pred)
    r2 = sklm.r2_score(y_test, y_pred)
    r2_adj = (r2 - (n_params - 1) /
        (y_test.shape[0] - n_params) * (1 - r2))

    print('Mean Square Error      = ' + str(MSE))
    print('Root Mean Square Error = ' + str(RMSE))
    print('Mean Absolute Error    = ' + str(MAE))
    print('Median Absolute Error  = ' + str(MedAE))
    print('R^2                    = ' + str(r2))
    print('Adjusted R^2           = ' + str(r2_adj))


# Loading X_test values from the CSV
# We pretend that these are new features
X_test = pd.read_csv("X_Test_Data.csv")
X_test = X_test.values

# Load full test data to get Y_test for metrics
data_file_name = "Test_Data.npz"
with open(data_file_name, 'rb') as f:
    feature_label_data = np.load(f)

Y_test = feature_label_data[:, -1]

# Create an empty array to hold predictions from REST API
Y_pred_from_api = []

# For each row of the X_test values
for curr_features in X_test:
    note = curr_features[0]
    data = curr_features[1:].tolist() # Pydantic NO LIKE numpy arrays

    # Form the correct data input structure
    features = {
        "note": str(note),
        "data": data
    }

    # Use requests.post with API_URL and features in json format
    # to do a post operation and have the REST API run a prediction
    API_URL = "http://127.0.0.1:8000/run_model"
    response = requests.post(API_URL, json=features)

    # Use json.loads to convert the json string to a dictionary
    output = json.loads(response.text)["value"]
    # Add the prediction to our Y_pred_from_api array
    Y_pred_from_api.append(output)

# Convert our Y_pred_from_api array to a numpy array
```

```
Y_pred = np.array(Y_pred_from_api)
# Check the metrics
print_metrics(Y_test, Y_pred, 4)
```

Implementation

Well, that looks all wonderful in theory, but does this work? First, we have to start this script and make sure it launches on our local server using uvicorn. Then, we need to test each method. Let's test each method two different ways. Using the automatically generated documentation for this API, and using requests from our fake user's Python script.

What does it look like when we start our API from the command line. AND how do we do that? Well, let me show you! We start our API from a command line terminal using the following line. **NOTE** that whatever the python script name of your API is, that's what you'd put in for Run_Model_API minus the .py part.

Launching The Uvicorn Server

I personally prefer to put the line below in a file named `run`, make that file executable, and then run that file from a terminal. Windows users may need to run a batch file or powershell script.

`uvicorn Run_Model_API:app --reload`

If successful, your uvicorn server launch will look like the following:

```
(py38std) thom@thom-
PT5610:~/DagsHub_Repos/API_Dev_Work/API_Dev_Work_Public/Basic_Model_Se
./run
INFO:     Will watch for changes in these directories:
['/home/thom/DagsHub_Repos/API_Dev_Work/API_Dev_Work_Public/Basic_Mode
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to
quit)
INFO:     Started reloader process [3238637] using watchgod
INFO:     Started server process [3238639]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

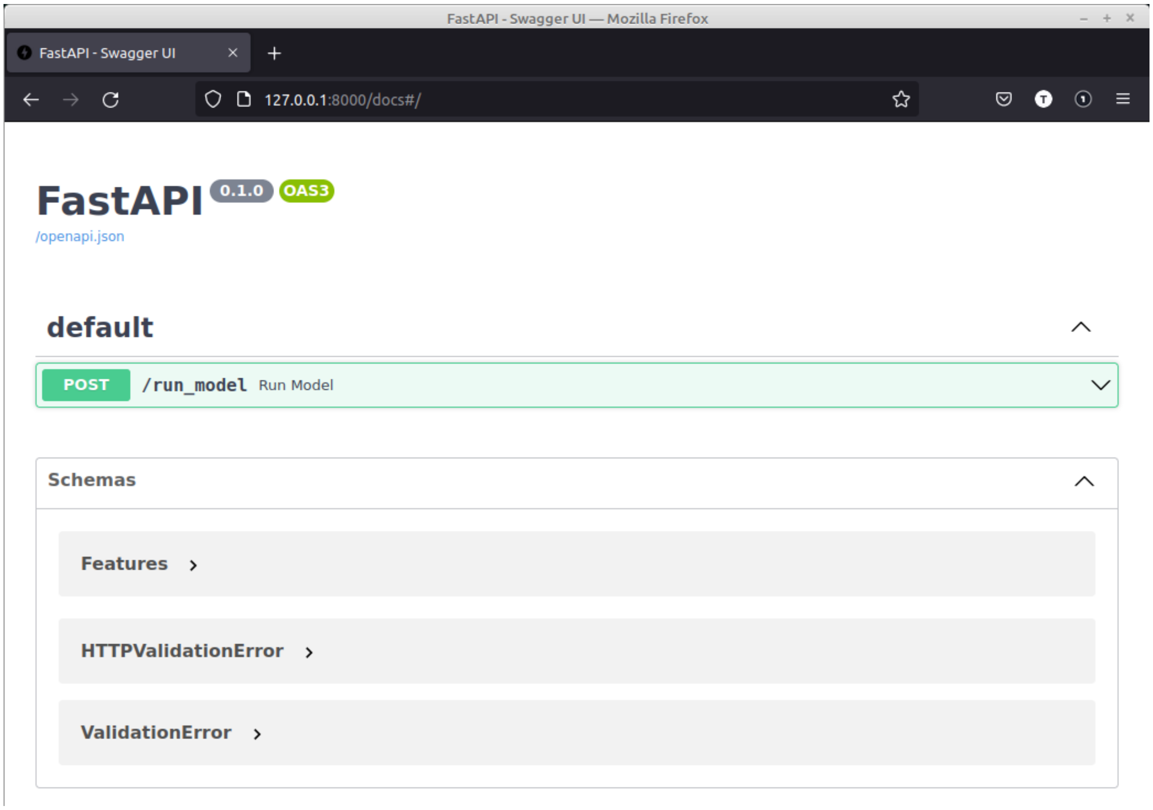
If you are running, great! If not, go back through this document carefully, OR step through the FastAPI tutorials until you can find your issue. Many people forget to make sure their terminal is looking at the same directory that their API script is in. You may also have a typo in your

`uvicorn <api-script-name-WITHOUT-.py>:app --reload`

I hope it runs for you, because once it does, the rest of this will likely go pretty smoothly for you.

Next, let's open the amazingly good automatically generated document page for our API using the below line. If you ever used PostMan for testing APIs, this is much the same, but specific to FastAPI and makes FastAPI easy to test and debug!

`http://127.0.0.1:8000/docs`



Expand the POST block and click on "Try it out" at the upper right of that expanded block. "Try it out" will switch to "Cancel" in case you decide you do NOT want to try it out right now. You will see a "Request body". Edit the dictionary in that "Request body" as shown in the next image.

Here's that `Request body` of data for your convenience.

```
{
    "note": "0",
    "data":
[0.6858635702788158,0.513191733213884,0.2007500892701115]
}
```

Now, we click on wide blue **Execute** button. We want to then check on two things. First, in this expanded POST block, scroll down a bit, and you will see the response that you formulated in your return statement if all went well. I have shown mine below.

Simulating Our Fake End User Using Our REST API

Well that's all great, but let's simulate a real life use case. Now we run the file above that we named Call_Model_Run_API.py. I will repeat this code below with the comments stripped. Again, the code below calls the REST API, that takes features inputs and provides prediction outputs, one record at a time.

In [ ]:

```
import requests
import pandas as pd
import numpy as np
import json
import sklearn.metrics as sklm
import math


def print_metrics(y_test, y_pred, n_params):
    pass # see full code for this function above.

X_test = pd.read_csv("X_Test_Data.csv")
X_test = X_test.values
```

```
data_file_name = "Test_Data.npz"
with open(data_file_name, 'rb') as f:
    feature_label_data = np.load(f)

Y_test = feature_label_data[:, -1]

Y_pred_from_api = []

for curr_features in X_test:
    note = curr_features[0]
    data = curr_features[1:].tolist()

    features = {
        "note": str(note),
        "data": data
    }

    API_URL = "http://127.0.0.1:8000/run_model"
    response = requests.post(API_URL, json=features)

    output = json.loads(response.text)["value"]
    Y_pred_from_api.append(output)

Y_pred = np.array(Y_pred_from_api)
print_metrics(Y_test, Y_pred, 4)
```

The results from the above code are ...

```
(py38std) thom@thom-PT5610:~/DagsHub_Repos/API_Dev_Work/API_Dev_Work_Public/Basic_Model_Se
/home/thom/.virtualenvs/py38std/bin/python /home/thom/DagsHub_Repos/API_Dev_Work/API_Dev_Work_Public/Basic_Model_S
Mean Square Error      = 0.0008951082926707904
Root Mean Square Error = 0.029918360460940877
Mean Absolute Error    = 0.023989555548717817
Median Absolute Error  = 0.020721802574338644
R^2                    = 0.9991017552979535
Adjusted R^2           = 0.9990880066545549
```

And these results are the same as the results we obtained when running Basic_Fake_Data_Model_Test_And_Measure.py.

Summary

We used Python FastAPI to create a REST API to provide model predictions using a trained model. Dang that was fun! We discovered some great new power and methods using FastAPI. I am eager for you to take your data science work to the next levels with FastAPI!

Until next time.

STRONGLY Suggested Near Future Work

What should you learn as you continue to develop FastAPI APIs for your data science work? Here are my suggestions with thoughts:

1. Docker - being able to run your APIs from a container makes them extra protected and long running. I am now successfully serving MANY APIs from production Linux servers in my company that are running from Docker containers. These containers automatically start backup if the server has to reboot. I can use these APIs in muliple applications, AND other data scientists are now using these APIs in their applications.
2. CI/CD - the Continuous Integration and Continuous Deployment system that your company uses will ... well, depend on your company. However, I feel that once you learn one system, you will find other CI/CD systems to be easier to learn.
3. Streamlit - I already know you LOVE Python like me. Why not code Dashboards and Simple Web UI applications for your data science work using Streamlit. It's a great system that uses pure Python to code up VERY nice interactive web pages to serve your models.
4. JavaScript - Now, when we say JavaScript, we obviously mean along with HTML and CSS. Why? We're not software developers. I'll come to answer this point at the end. Once you know JavaScript, which is a lot like Python, you can make any Web UI you need, AND you can call your APIs from that JavaScript to bring your API data into your HTML Web UI pages. JavaScript is great, because it allows you to dynamically change things in your HTML pages and make them interactive Web UIs.
5. Kubernetes - When your APIs and Web UIs that are using many APIs become popular start to be used by increasingly more people, it's time to add more servers to balance that load of more users. Kubernetes is the most current popular way to do this.

Thom! You're Crazy!

Yes, I know that. Oh! What type of crazy are you referring to? Oh ... You are thinking, "Thom, we are NOT software engineers and web developers and dev ops engineers or even ML ops engineers. We are data scientists." Yes. That is correct. However, in my experience, these other disciplines, even when crazy good at what they do, cannot always get to a point to support our projects, or at least not get to them quickly. What I've learned is that if I can build something and demonstrate it, then the organization will get behind it and support it. To get this support, I can't always garner software development time from software engineers and such to implement and deliver my data science ideas. So, what do I do? I develop a prototype and demonstrate it. Depending on the task, this at least gets things going and creates a vision of what can be. Also, if you learn to do this, you open up MANY more opportunities and potential for yourself. I hope this helps.