Lab: Model packaging and serving (1/2)
Professor : Fahd KALLOUBI
Year : 2023/2024
Major : QFM

This Lab focuses on different technologies and tools, which will serve as a starting point in order to move from a research environment to a production environment. Furthermore, in the next Lab we will bring together these tools in order to produce a package of our ML model available on PyPi (i.e., Python Package Index). Additionally, and based on the built package, we will serve it using an API.

1. Introduction to Pytest

Pytest has become a de facto standard in the industry thanks to several strengths that this library offers and which were able to replace the native python unit testing package.

> Python unit testing : https://docs.python.org/3/library/unittest.html
> Official Pytest documentation : https://docs.pytest.org/en/latest/

Create a folder named "Pytest", in which you create a file named "my_module.py" and then copy the following code into it:

```python
def square(x):
    return x ** 2
```

This is the function we want to test. Next, create a file named "test_my_module.py" containing the following code:

```python
from my_module import square

def test_square_gives_correct_value():

    subject= square(2)

    assert subject == 4
```

While on the root (i.e., Pytest), type the command "pytest" (this assumes that you have already installed pytest with "pip install pytest").

You will notice that pytest was able to automatically detect the test file. This means that your test file must start with "test_". You can test this by creating another file without "test_" and running the test again.

Now change the value "4" to another value and run the test again (i.e., the "pytest" command).

- Fixtures

Fixtures are functions attached to tests that execute before the test function is executed. A fixture function always begins with the decorator "@pytest.fixture".

Replace the contents of the "test_my_module.py" file with the following code:

```python
from my_module import square

import pytest

@pytest.fixture
def input_value():
    return 4

def test_square_gives_correct_value(input_value):
    subject= square(input_value)
    assert subject == 16
```

While on the root (i.e., Pytest), retype the "pytest" command.

Now create a new file called "test_my_module_again.py" containing the following code:

```python
from my_module import square

import pytest

@pytest.fixture
def input_value():
    return 4

def test_square_return_int(input_value):

    subject = square(input_value)

    assert isinstance(subject,int)
```

By typing the "pytest" command, you will notice that pytest was able to detect 2 tests now.

The point of fixtures is to share the function with several test functions. To do this, and by convention, pytest allows you to group the fixtures in a file called "conftest.py" whose content for our case is:

```python
import pytest

@pytest.fixture
def input_value():
    return 4
```

From now on, our two files can be written by removing the code above.

```python
from my_module import square

def test_square_return_int(input_value):

    subject = square(input_value)

    assert isinstance(subject,int)
```

Make said changes and repeat your test.

- Parameterized tests

Another very important feature of pytest is the ability to write parameterized tests.

Let's apply a parameterized test in our "test_my_module_again.py" file.

```python
from my_module import square
import pytest

@pytest.mark.parametrize(
    'inputs', [
    2,3,4]
)

def test_square_return_int(inputs):

    subject = square(inputs)

    assert isinstance(subject,int)
```

Take the test again. We had 2 tests while now we have 4 tests, because in the "test_my_module_again.py" file, we launched 3 tests (each parameter will be tested).

2. Introduction to Pydantic

Pydantic is a library for data validation and parameter management using python-like annotations. It enforces type conversions at runtime and provides user-friendly errors when data is invalid.

➢ Official documentation: https://pydantic-docs.helpmanual.io/

The main way to define objects in pydantic is through models (A model is a class that inherits from BaseModel).

A model is a type containing multiple data types.

Let's start with this example:

```python
from datetime import datetime
from typing import List, Optional
from pydantic import BaseModel
class User(BaseModel):
    id: int
    name = 'John Doe'
    signup_ts: Optional[datetime] = None
    friends: List[int] = []

external_data = {
    'id': '123',
    'signup_ts': '2019-06-01 12:22',
    'friends': [1, 2, '5'],
}
user = User(**external_data)
print(user.id)
#> 123
```

```
print(repr(user.signup_ts))
#> datetime.datetime(2019, 6, 1, 12, 22)
print(user.friends)
#> [1, 2, 3]
print(user.dict())
```

➢ Id: of type int. is an annotation-declaration which means that this field is mandatory. Strings, bytes or floats will be converted to integers if possible; otherwise an exception will be thrown.
➢ The other 3 fields: these are fields that are not mandatory
➢ Friends is a list of integers. As for the id field, integer type objects will be converted to integers.

Run this code and look at the result. What do you notice?

Change the value of the id field (in external_data) to a string that cannot be converted to an integer and run the code again.

- Recursive models

A model is also a type. We can call a model within another model.

```
from typing import List
from pydantic import BaseModel


class Foo(BaseModel):
    count: int
    size: float = None

class Bar(BaseModel):
    apple = 'x'
    banana = 'y'

class Spam(BaseModel):
    foo: Foo
    bars: List[Bar]

m = Spam(foo={'count': 4}, bars=[{'apple': 'x1'},
{'apple': 'x2'}])
print(m)
print(m.dict())
```

Run this code to see the result.

- Validators

Custom validation and complex relationships between objects can be achieved using the validator decorator.

```
from pydantic import BaseModel, ValidationError, validator
class UserModel(BaseModel):
    name: str
    username: str
    password1: str
```

```python
        password2: str

    @validator('name')
    def name_must_contain_space(cls, v):
        if ' ' not in v:
            raise ValueError('must contain a space')
        return v.title()

    @validator('password2')
    def passwords_match(cls, v, values, **kwargs):
        if 'password1' in values and v != values['password1']:
            raise ValueError('passwords do not match')
        return v

    @validator('username')
    def username_alphanumeric(cls, v):
        assert v.isalnum(), 'must be alphanumeric'
        return v
user = UserModel(
    name='samuel colvin',
    username='scolvin',
    password1='zxcvbn',
    password2='zxcvbn',
)
print(user)
try:
    UserModel(
        name='samuel',
        username='scolvin',
        password1='zxcvbn',
        password2='zxcvbn2',
    )
except ValidationError as e:
    print(e)
```

3. Introduction à FastAPI

You will notice that FastAPI looks a lot like Flask. However, FastAPI is much more practical in the sense that it tries to solve most of Flask's drawbacks:
  ➢ Validation is supported thanks to Pydantic, imagine an ML model receiving erroneous data.
  ➢ FastAPI is designed for rapid production
  ➢ Documentation is automatically generated with the code
  ➢ FastAPI allows asynchronous processing
  ➢ FastAPI adopts ASGI server which is a successor of WSGI (adopted by Flask), ASGI adds asynchronization
  ➢ FastAPI does not need html, thanks to swagger UI you can directly visualize the result of your API on a browser

Let's start by an example:

  ➢ Install FastAPI: pip install fastapi
  ➢ Install uvicorn (ASGI server): pip install uvicorn

Create a file named "main.py" containing the code below:

```python
from typing import Optional

from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

Note that the routing principle is the same as Flask.

To run this file, type the following command:

uvicorn main:app --reload

- o main: is the main file in the python module
- o app: is the object created in main.py (i.e., the line app = FastAPI())
- o --reload: to restart the server after each code change (only in a development environment)

As you can notice, it is port 8000 which hosts our application.

Open this link then: http://127.0.0.1:8000/items/5?q=somequery and you will see the Json response.

Change the item id value to a character or string to get an overview of the effect of Pydantic on FastAPI (in another Framework, you must program this validation)

To view the interactive application documentation go to http://127.0.0.1:8000/docs. More than that, you can test with other values on the form provided thanks to Swagger UI.

Now we will serve an ML model using FastAPI.

In the Lab folder you will find a notebook called "iris.ipynb", run it to get the model.

Then create a file called "app.py" containing the following code:

```python
from fastapi import FastAPI
from pydantic import BaseModel
import uvicorn

app = FastAPI()

class iris(BaseModel):
    a:float
    b:float
    c:float
    d:float
```

```python
from sklearn.linear_model import LogisticRegression
import pandas as pd
import pickle
#we are loading the model using pickle
model = pickle.load(open('model_iris', 'rb'))
@app.get("/")
def home():
    return {'ML model for Iris prediction'}
@app.post('/make_predictions')
async def make_predictions(features: iris):

    return({"prediction":str(model.predict([[features.a,features.b,featu
res.c,features.d]])[0])})

if __name__ == "__main__":
    uvicorn.run("app:app", host="0.0.0.0", port=8080, reload=True)
```

On your command line, run this file (python app.py).

Open this link http://localhost:8080/docs to see the Get and Post request response.

Click on post query and try other correct or wrong values.

To Do :

1. In the lab folder, you will find a notebook called "Furniture prediction notebook.ipynb", run it to get the model.pkl
2. Produce an API of this model capable of making the prediction based on the provided features.
3. Like flask, FastAPI also has the same jinja2 template engine, present the result of this API using this template engine.
4. Do the same thing again based on the same dataset of your previous assignment