Lab: Model testing
Professor: Fahd KALLOUBI
Year: 2023/2024
Major: QFM – Data engineering

In this Lab, we will test the performance of several deployments, namely:

- A Deployment using FastAPI
- A deployment using Docker to containerize an API
- A deployment using Tensorflow Extended (TFX) to serve a model

For more information, try consulting the following resources:

- Locust documentation: https://docs.locust.io
- TFX documentation: https://www.tensorflow.org/tfx/guide
- FastAPI documentation: https://fastapi.tiangolo.com
- Docker documentation: https://docs.docker.com
- HuggingFace TFX serving: https://huggingface.co/blog/tf-serving

The following packages must be installed:

- TensorFlow
- PyTorch
- transformers >=4.00
- fastAPI
- Docker
- Locust

1. Serve an NLP model (transformer) using FastAPI

We will use a QA (Question answering) model and serve it using FastAPI. To do this, we will use the "transformers" library to use language models based on a Transformer-Encoder architecture and more specifically we will use DistillBERT which is a lighter version of BERT "Bidirectional Encoder Representations from Transformers" which belongs to a family called "autoencoding language models".

In this part, we will use the DistillBERT model (a lightweight version of BERT) pre-trained on the SQUAD dataset "The Stanford Question Answering Dataset" (https://rajpurkar.github.io/SQuAD-explorer/).

In the Lab folder, open the "1. fastAPI_Transformer_model_serving" folder and then the "main.py" file:

1. We will start by creating a data model using Pydantic to model the inputs (Question-Answer).

```
from pydantic import BaseModel
class QADataModel(BaseModel):
    question: str
    context: str
```

2. After creating an instance of fastAPI, we will load the pre-trained model using the "pipeline" module of the "transformers" library

```
from transformers import pipeline
model_name = 'distilbert-base-cased-distilled-squad'
model = pipeline(model=model_name, tokenizer=model_name, task='question-answering')
```

3. Next, we will create an endpoint for our API, by creating a function in asynchronous mode

```
@app.post("/question_answering")
async def qa(input_data: QADataModel):
    result = model(question = input_data.question, context=input_data.context)
    return {"answer": result["answer"]}
```

4. Finally, and using uvicorn, we can start our API.

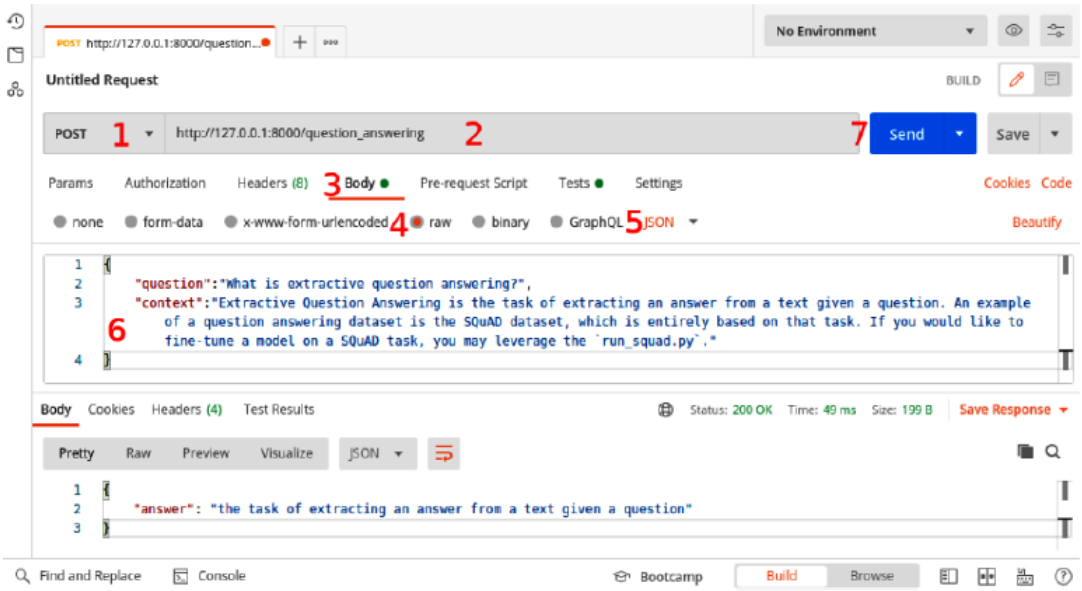
```
if __name__ == '__main__':
    uvicorn.run('main:app', workers=1)
```

Run the file by typing the command “python main.py”, and then navigate to the documentation http://127.0.0.1:8000/docs .

Experiment with this entry to validate that the API works.

```
{
    "question": "What is extractive question answering?",
    "context": "Extractive Question Answering is the task of extracting an answer from a text given a question. An example of a question answering dataset is the SQuAD dataset, which is entirely based on that task. If you would like to fine-tune a model on a SQuAD task, you may leverage the `run_squad.py`."
}
```

Now we'll try a tool to query our API using curl requests. Postman is an easy-to-use GUI (download link: https://www.postman.com/downloads/).



Each step of setting up the service on Postman is numbered in the figure:

1. Choose Post as method
2. Enter your full endpoint http://127.0.0.1:8000/question_answering
3. Select Body
4. Then Raw
5. Choose Json as data type
6. Data must be entered as Json type
7. Click Send

2. Containerizing our API using Docker

To save time in production and facilitate the deployment process, it is essential to use Docker. It is very important to isolate your service and your application. Also note that the same code can be executed anywhere, regardless of the operating system.

Open the "2. Dockerizing_API" folder, the steps for dockerizing our API can be summarized as follows:

1. Put the main.py file in the "app" folder.
2. Next, you need to eliminate the last part of the main.y file:

```
if __name__ == '__main__':
    uvicorn.run('main:app', workers=1)
```

3. Next, you need to create a Dockerfile for your fastAPI:

```
FROM python:3.9

RUN pip install torch

RUN pip install fastapi uvicorn transformers

EXPOSE 80
```

```
COPY ./app /app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8005"]
```

4. Finally, you can build your Docker container

```
docker build -t qaapi .
```

5. And launch it with the following command:

```
docker run -p 8000:8000 qaapi
```

As a result, you can now access your API using port 8000. However, you can still use Postman, as described in the previous section.

3. Serve a transformer type model using TFX

TFX provides a faster and more efficient way to serve deep learning models. But it has some important key points that you need to understand before using it. The model must be a registered model type from TensorFlow so that it can be used by TFX. For more information about TensorFlow saved models, you can read the official documentation: https://www.tensorflow.org/guide/saved_model

Open the "3. Faster_Transformer_model_serving_using_Tensorflow_Extended" folder and run the "saved_model.ipynb" notebook to produce the saved model.

Next, we will extract the Docker image for Tensorflow Extended (for more information: https://www.tensorflow.org/tfx/serving/docker):

```
docker pull tensorflow/serving
```

Now we will run the Docker container and copy the saved model into it.

```
docker run -d --name serving_base tensorflow/serving
```

```
docker cp tfx_model/saved_model serving_base:/models/bert
```

This will copy the saved model to the container. However, we must validate the change.

```
docker commit --change "ENV MODEL_NAME bert" serving_base
my_bert_model
```

Now that everything is working well, we can stop the container.

```
docker kill serving_base
```

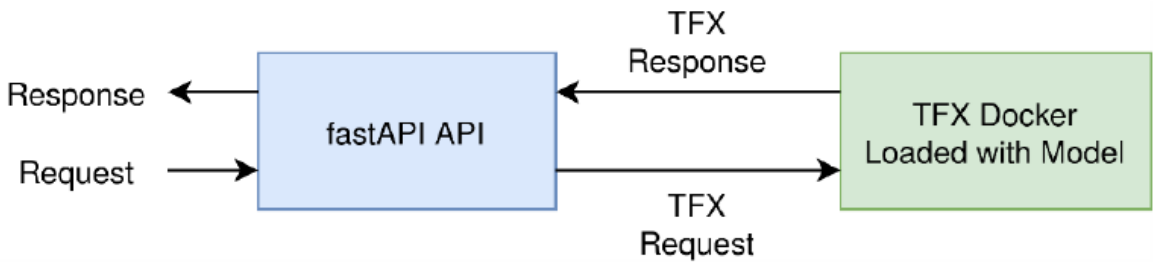
This will stop the container from running.
Now that the model is ready and can be served by TFX Docker, you can simply use it with another service. The reason we need another service to call TFX is that Transformer-based

models have a special input format provided by the tokenizers (this means the text must be processed before it is passed to the model).
To do this, you need to create a fastAPI service that will call the API that was served by the TensorFlow serving container. Before coding your service, you need to start the Docker container by giving it parameters to run the BERT-based sentiment analysis model.

```
docker run -p 8501:8501 -p 8500:8500 --name bert my_bert_model
```

- port 8500 is exposed for gRPC
- port 8501 is exposed for the REST API

Keep this window open to keep the service running because we are going to use it with Fast API. The overall architecture of the new API is as follows:

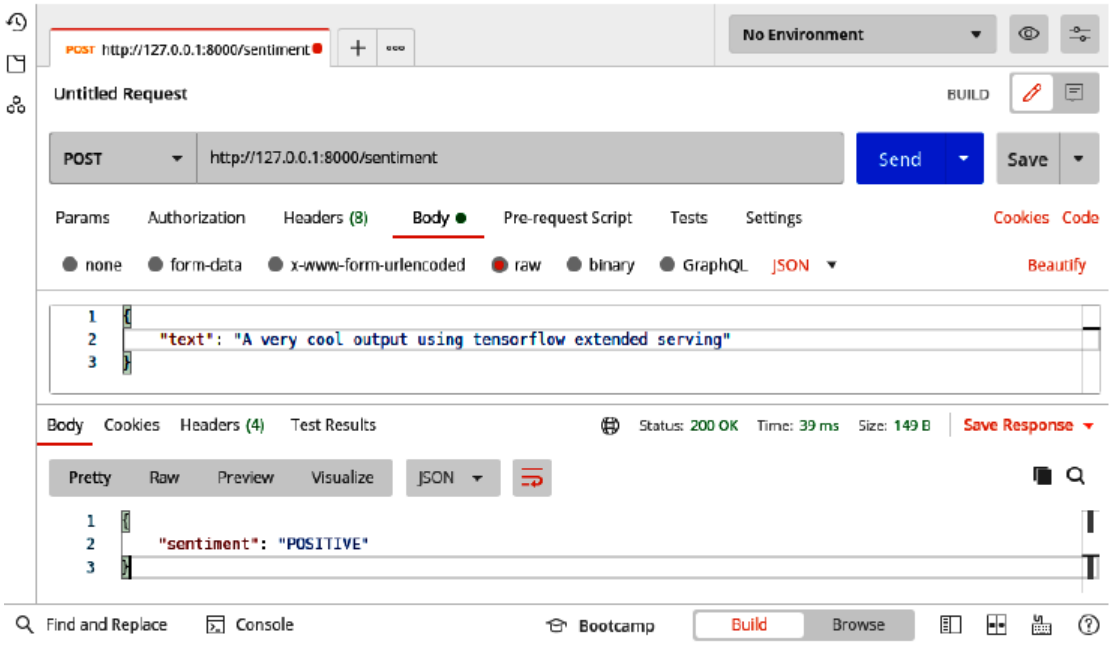


Now that we have our TFX service with Docker ready to consume using REST API (port 8501), we will consume it using FastAPI.

Now open the “main.py” file and look at its contents. What do you notice?

Run it using the “python main.py” command.

The service is now ready to use (127.0.0.1:8000/docs). However, you can use Postman to query it.



4. Loading Test using Locust

There are many applications that we can use to perform load testing. Most of these applications and libraries provide useful information about service response time and delay. They also provide information on the failure rate. Locust is one of the best tools for this purpose. We will use it to test the loading of the three methods seen previously:

- Using fastAPI only,
- Using Dockerized fastAPI and
- By serving the model with TFX using fastAPI.

First start with installing Locust.

```
pip install locust
```

You can now test your APIs (services), you must prepare a locust file in which you will define your user and his behavior. Below is an example of a "locust_file.py" file in which we will test the latest deployed model (i.e., TFX with FastAPI for sentiment analysis)

```
from locust import HttpUser, task
from random import choice
from string import ascii_uppercase
class User(HttpUser):
    @task
    def predict(self):
        payload = {"text": ".join(choice(ascii_ uppercase) for i in range(20))}
        self.client.post("/sentiment", json=payload)
```

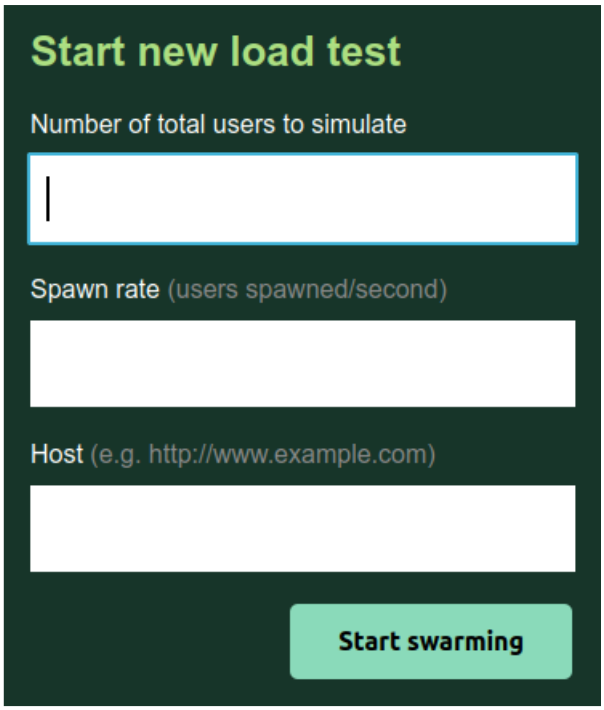
By using HttpUser and creating the User class that inherits from it, we can define an HttpUser class. The @task decorator is essential to define the task that the user should perform. The prediction function is the actual task that the user will perform repeatedly. It will generate a random string of length 20 and send it to your API.

To start the test, run the command "locust –f locust_file.py".

The service is available at http://localhost:8089/ . once on this link, you will find the interface below.

We will set the total number of users to simulate to 10, the spawn rate to 1, and the host to http://127.0.0.1:8000, this is where our service runs. After setting these parameters, click "Start swarming".

At this point the user interface will change and testing will begin. To stop the test at any time, click the Stop button.



You can click on “Charts” to see the results visualization.

Test all three versions and compare the results to see which one works best. Remember that services should be independently tested on the machine you want to serve them on. In other words, you need to run one service at a time and test it, close the service, run the other and test it, and so on.

Complete the table below with the approximate results you got.

What is the best deployment ?

	TFX-based FastAPI	FastAPI	Dockerized FastAPI

RPS			
Average RT(ms)			

In the previous table, requests per second (RPS) refers to the number of requests per second that the API responds to, while average response time (RT) refers to the milliseconds it takes for the service to respond to a given call.