One prediction per request

You can find all of the following code in the `no-batch/` directory. In your terminal `cd` to this directory to continue with the lab. If you are currently within the root of the repo you can use the command `cd course4/week2-ungraded-labs/C4_W2_Lab_1_FastAPI_Docker/no-batch/`.

Notice that the server's code must be in the file `main.py` within a directory called `app`, following FastAPI's guidelines.

Coding the server

Begin by importing the necessary dependencies. You will be using `pickle` for loading the pre-trained model saved in the `app/wine.pkl` file, `numpy` for tensor manipulation, and the rest for developing the web server with `FastAPI`.

Also, create an instance of the `FastAPI` class. This instance will handle all of the functionalities for the server:

```
import pickle
import numpy as np
from fastapi import FastAPI
from pydantic import BaseModel


app = FastAPI(title="Predicting Wine Class")
```

Now you need a way to represent a data point. You can do this by creating a class the subclasses from pydantic's `BaseModel` and listing each attribute along with its corresponding type.

In this case a data point represents a wine so this class is called `Wine` and all of the features of the model are of type `float`:

```
# Represents a particular wine (or datapoint)
class Wine(BaseModel):
    alcohol: float
    malic_acid: float
    ash: float
```

```
    alcalinity_of_ash: float
    magnesium: float
    total_phenols: float
    flavanoids: float
    nonflavanoid_phenols: float
    proanthocyanins: float
    color_intensity: float
    hue: float
    od280_od315_of_diluted_wines: float
    proline: float
```

Now it is time to load the classifier into memory so it can be used for prediction. This can be done in the global scope of the script but here it is done inside a function to show you a cool feature of FastAPI.

If you decorate a function with the `@app.on_event("startup")` decorator you ensure that the function is run at the startup of the server. This gives you some flexibility if you need some custom logic to be triggered right when the server starts.

The classifier is opened using a context manager and assigned to the `clf` variable, which you still need to make global so other functions can access it:

```
@app.on_event("startup")
def load_clf():
    # Load classifier from pickle file
    with open("/app/wine.pkl", "rb") as file:
        global clf
        clf = pickle.load(file)
```

Finally you need to create the function that will handle the prediction. This function will be run when you visit the `/predict` endpoint of the server and it expects a `Wine` data point.

This function is actually very straightforward, first you will convert the information within the `Wine` object into a numpy array of shape `(1, 13)` and then use the `predict` method of the classifier to make a prediction for the data point. Notice that the prediction must be casted into a list using the `tolist` method.

Finally return a dictionary (which FastAPI will convert into `JSON`) containing the prediction.

```
@app.post("/predict")
def predict(wine: Wine):
    data_point = np.array(
        [
            [
                wine.alcohol,
                wine.malic_acid,
                wine.ash,
                wine.alcalinity_of_ash,
                wine.magnesium,
                wine.total_phenols,
                wine.flavanoids,
                wine.nonflavanoid_phenols,
                wine.proanthocyanins,
                wine.color_intensity,
                wine.hue,
                wine.od280_od315_of_diluted_wines,
                wine.proline,
            ]
        ]
    )

    pred = clf.predict(data_point).tolist()
```

```
    pred = pred[0]
    print(pred)
    return {"Prediction": pred}
```

Now the server's code is ready for inference, although you still need to spin it up. If you want to try it locally (given that you have the required dependencies installed) you can do so by using the command `uvicorn main:app --reload` while on the same directory as the `main.py` file. **However this is not required as you will be dockerizing this server next**.

Dockerizing the server

Going forward all commands are run assuming you are currently within the `no-batch/` directory.

Also you should create a directory called `app` and place `main.py` (the server) and its dependencies (`wine.pkl`) there as explained on the official FastAPI docs on how to deploy with Docker. This should result in a directory structure that looks like this:

```
..
└── no-batch
    ├── app/
    │   ├── main.py (server code)
    │   └── wine.pkl (serialized classifier)
    ├── requirements.txt (Python dependencies)
    ├── wine-examples/ (wine examples to test the server)
    ├── README.md (this file)
    └── Dockerfile
```

Create the Dockerfile

The `Dockerfile` is made up of all the instructions required to build your image. If this is the first time you see this kind of file it might look intimidating but you will see it is actually easier than it looks. First take a look at the whole file:

Base Image

```
FROM frolvlad/alpine-miniconda3:python3.7
```

The `FROM` instruction allows you to select a pre-existing image as the base for your new image. **This means that all of the software available in the base image will also be available on your own.** This is one of Docker's nicest features since it allows for reusing images when needed.

In this case your base image is `frolvlad/alpine-miniconda3:python3.7`, let's break it down:

- `frolvlad` is the username of the author of the image.
- `alpine-miniconda3` is its name.
- `python3.7` is the image's tag.

This image contains an alpine version of Linux, which is a distribution created to be very small in size. It also includes miniconda with Python 3. Notice that the tag let's you know that the specific version of Python being used is 3.7. Tagging is great as it allows you to create different versions of similar images. In this case you could have this same image with a different version of Python such as 3.5.

You could use many different base images such as the official `python:3.7` image. However if you compared the size you will encounter it is a lot heavier. In this case you will be using the one mentioned above as it is a great minimal image for the task at hand.

Installing dependencies

Now that you have an environment with Python installed it is time to install all of the Python packages that your server will depend on. First you need to copy your local `requirements.txt` file into the image so it can be accessed by other processes, this can be done via the `COPY` instruction:

```
COPY requirements.txt .
```

Now you can use `pip` to install these Python libraries. To run any command as you would on `bash`, use the `RUN` instruction:

```
RUN pip install -r requirements.txt && \
        rm requirements.txt
```

Notice that two commands were chained together using the `&&` operator. After you installed the libraries specified within `requirements.txt` you don't have more use for that file so it is a good idea to delete it so the image includes only the necessary files for your server to run.

This can be done using two `RUN` instructions, however, it is a good practice to chain together commands in this manner since Docker creates a new layer every time it encounters a `RUN`, `COPY` or `ADD` instruction. This will result in a bigger image size. If you are interest in best practices for writing Dockerfiles be sure to check out this resource.

Exposing the port

Since you are coding a web server it is a good idea to leave some documentation about the port that the server is going to listen on. You can do this with the `EXPOSE` instruction. In this case the server will listen to requests on port 80:

```
EXPOSE 80
```

Copying your server into the image

Now you should put your code within the image. To do this you can simply use the `COPY` instruction to copy the `app` directory within the root of the container:

```
COPY ./app /app
```

Spinning up the server

Containers are usually meant to start and carry out a single task. This is why the `CMD` instruction was created. This is the command that will be run once a container that uses this image is started. In this case it is the command that will spin up the server by specifying the host and port. Notice that the command is written in a `JSON` like format having each part of the command as a string within a list:

```
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

What is meant by `JSON` like format is that Docker uses `JSON` for its configurations and the `CMD` instruction expects the commands as a list that follows `JSON` conventions.

Putting it all together

The resulting `Dockerfile` will look like this:

```
FROM frolvlad/alpine-miniconda3:python3.7

COPY requirements.txt .

RUN pip install -r requirements.txt && \
        rm requirements.txt

EXPOSE 80

COPY ./app /app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

Remember you can also look at it within the `no-batch` directory.

Build the image

Now that the `Dockerfile` is ready and you understand its contents, it is time to build the image. To do so, double check that you are within the `no-batch` directory and use the `docker build` command.

```
docker build -t mlepc4w2-ugl:no-batch .
```

You can use the `-t` flag to specify the name of the image and its tag. As you saw earlier the tag comes after the colon so in this case the name is `mlepc4w2-ugl` and the tag is `no-batch`.

After a couple of minutes your image should be ready to be used! If you want to see it along with any other images that you have on your local machine use the `docker images` command. This will display all of you images alongside their names, tags and size.

Run the container

Now that the image has been successfully built it is time to run a container out of it. You can do so by using the following command:

```
docker run --rm -p 80:80 mlepc4w2-ugl:no-batch
```

You should recognize this command from a previous ungraded lab. Let's do a quick recap of the flags used:

- `--rm`: Delete this container after stopping running it. This is to avoid having to manually delete the container. Deleting unused containers helps your system to stay clean and tidy.
- `-p 80:80`: This flags performs an operation knows as port mapping. The container, as well as your local machine, has its own set of ports. So you are able to access the port 80 within the container, you need to map it to a port on your computer. In this case it is mapped to the port 80 in your machine.

At the end of the command is the name and tag of the image you want to run.

After some seconds the container will start and spin up the server within. You should be able to see FastAPI's logs being printed in the terminal.

Now head over to localhost:80 and you should see a message about the server spinning up correctly.

Nice work!

Make requests to the server

Now that the server is listening to requests on port 80, you can send `POST` requests to it for predicting classes of wine.

Every request should contain the data that represents a wine in `JSON` format like this:

```
{
  "alcohol":12.6,
  "malic_acid":1.34,
  "ash":1.9,
  "alcalinity_of_ash":18.5,
  "magnesium":88.0,
  "total_phenols":1.45,
  "flavanoids":1.36,
  "nonflavanoid_phenols":0.29,
  "proanthocyanins":1.35,
  "color_intensity":2.45,
  "hue":1.04,
  "od280_od315_of_diluted_wines":2.77,
  "proline":562.0
}
```

This example represents a class 1 wine.

Remember from Course 1 that FastAPI has a built-in client for you to interact with the server. You can use it by visiting localhost:80/docs

You can also use `curl` and send the data directly with the request like this:

```
curl -X 'POST' http://localhost/predict \
  -H 'Content-Type: application/json' \
  -d '{
  "alcohol":12.6,
  "malic_acid":1.34,
  "ash":1.9,
  "alcalinity_of_ash":18.5,
  "magnesium":88.0,
  "total_phenols":1.45,
  "flavanoids":1.36,
  "nonflavanoid_phenols":0.29,
  "proanthocyanins":1.35,
  "color_intensity":2.45,
  "hue":1.04,
  "od280_od315_of_diluted_wines":2.77,
  "proline":562.0
}'
```

Or you can use a `JSON` file to avoid typing a long command like this:

```
curl -X POST http://localhost:80/predict \
    -d @./wine-examples/1.json \
```

```
    -H "Content-Type: application/json"
```

Let's understand the flags used:

- `-X` : Allows you to specify the request type. In this case it is a `POST` request.
- `-d` : Stands for `data` and allows you to attach data to the request.
- `-H` : Stands for `Headers` and it allows you to pass additional information through the request. In this case it is used to the tell the server that the data is sent in a `JSON` format.

There is a directory called `wine-examples` that includes three files, one for each class of wine. Use those to try out the server and also pass in some random values to see what you get!

Congratulations on finishing part 1 of this ungraded lab!

During this lab you saw how to code a webserver using FastAPI and how to Dockerize it. You also learned how `Dockerfiles`, `images` and `containers` are related to each other and how to use `curl` to make `POST` requests to get predictions from servers.

In the next part you will modify the server to accept batches of data instead of a single data point per request.

Jump to Part 2 - Adding batching to the server