University Master's Degree in
Machine Learning

Assignment: Continuous Integration using GitHub Actions

The main objective of this assignment is to develop a Continuous Integration (CI) pipeline using GitHub actions. Specifically, you will have to create a repository, named MLOps-Lab1, where you will develop a machine learning project through an incremental methodology. The project will end up (in Lab 3) being a tool to classify images using a deep learning method. In the first stage of the project, the one developed in this first lab, you must start building the initial logic of the process. That is, **you must construct a module where you**

- Define a method to predict the class of a given image. In this first lab, the class will be randomly chosen among a set of class names (of your choice).
- Define a method to resize an image to a certain size.
- You can define more methods to preprocess an image (of your choice).

Then, as in Lab0, **you must create a Command Line Interface (CLI)** to interact with these functionalities.

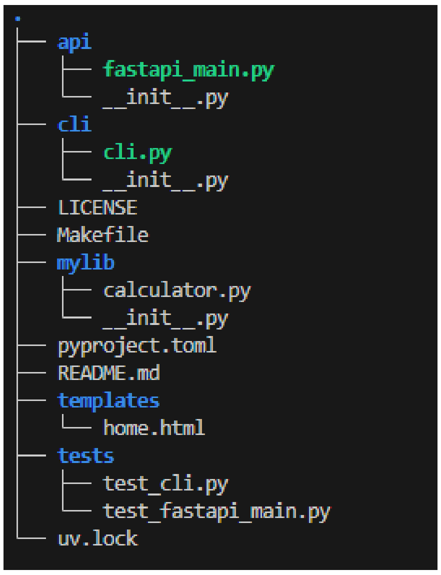
In this lab, you **also must create the API** using the FastAPI framework. It is suggested that you include these modules into three different folders (each one containing the __init__.py file to treat them as packages).

As in the previous lab, you also **must create the files devoted to testing the three components** (logic, CLI and API). You are suggested to include them in a different folder (named *tests*).

To develop the project, you must

- **Create a specific virtual environment** to include just the specific libraries you need.
- **Create a html file that will be used as the home of the constructed API** (included in a folder named, for instance, *templates*).
- To ease the CI process, you also need to **create a Makefile** where you define how to
  - Install the dependencies
  - Lint the code
  - Format the code
  - Test the code
  - You can add a refactor option (format plus lint) as well as an all option (all the previous options in the right order)
- In the **README.md file** you should provide the **information** about the project, and you also should include the **status badge of the CI pipeline** (how to obtain it is explained later in this document).

To **help in the development of this lab**, we created a repository to develop a calculator application devoted to performing some basic arithmetical operations. The public GitHub repository is https://github.com/JoseanSanz/MLOps-Lab1-demo. You can clone it and perform the modifications you want. The scaffold of this calculator application project is illustrated in the following figure.



```
.
├── api
│   ├── fastapi_main.py
│   └── __init__.py
├── cli
│   ├── cli.py
│   └── __init__.py
├── LICENSE
├── Makefile
├── mylib
│   ├── calculator.py
│   └── __init__.py
├── pyproject.toml
├── README.md
├── templates
│   └── home.html
├── tests
│   ├── test_cli.py
│   └── test_fastapi_main.py
└── uv.lock
```

Figure 1. Scaffold of the calculator application.

Now, the steps performed to produce all the code of this repository are explained.

In first place a public GitHub repository was created, adding a Readme file, selecting Python in the .gitignore option (it excludes versioning common Python build, cache, virtual environment and IDE artifacts) and adding the MIT license.

Then, the repository was added to the Visual Studio Code IDE. A new virtual environment was used for this project (according to the best practices in MLOps). Therefore, in the terminal:

- *deactivate* the *py313ml* environment
- create a new virtual environment is created: *uv init* and *uv sync*
- activate it: *source .venv/bin/activate*
- add the necessary packages in the project: *uv add package-name*
    - The packages you will need to solve this lab are: *pylint, pytest, pytest-cov, black, click, fastapi, uvicorn, jinja2* and *httpx*.

After that, all the files composing the project were programmed. The *Makefile* was also created so that one can easily use the installation, linting, formatting and testing options. In the Makefile you give name of every option and then you describe the command associated with each option.

option-name:

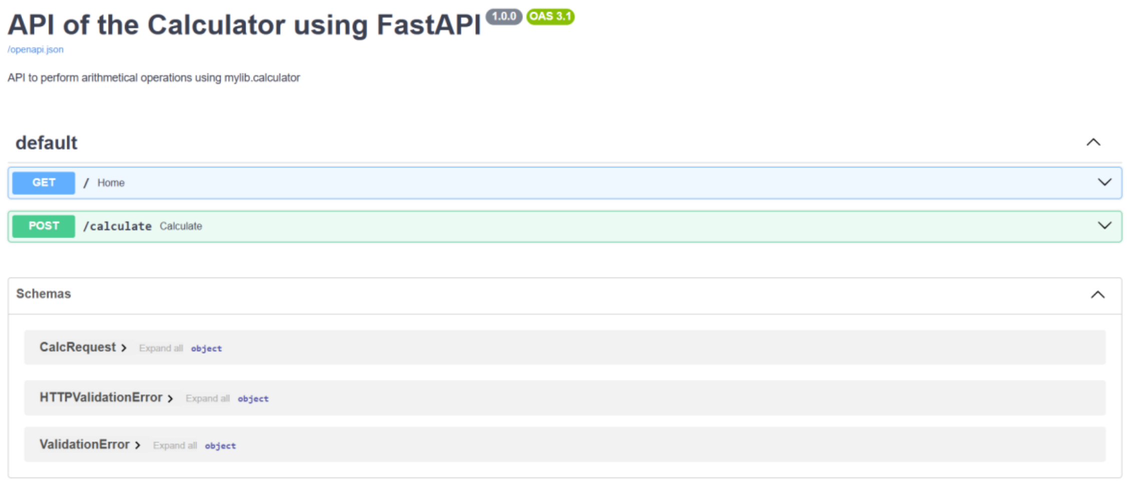
command(s) of its usage

NOTE: you can query the *Makefile* provided in the demo repository to see several examples of commands. To use each command, in the terminal, you have to run the command *make option-name*.

Once the functionalities (logic, CLI and API) are programed you can try them in your local environment. To execute them, you have to run the following command: *uv run python -m file-name.py*. If you obtain a permission denied error you have to give executions permission using the following command: *chmod +x file-name.py*.

- The command to execute the CLI (using, for instance, the add command) is
  uv run python -m cli.cli add 2 3
- The command to execute the API is
  uv run python -m api.api

When you execute the latter, you will see in the terminal several messages starting from INFO and a pop-up to open the API in the browser. You can open the browser when see the message "*INFO: Application startup complete.*". In the browser you will see the message determined in the main entry *endpoint (/)*, which loads the *home.html* file located in the *templates* folder. Then, to interact with the remainder *endpoints*, you have to add */docs* to the URL and you will see an interface like that of the following figure.



API of the Calculator using FastAPI 1.0.0 OAS 3.1

/openapi.json

API to perform arithmetical operations using mylib.calculator

default

GET / Home

POST /calculate Calculate

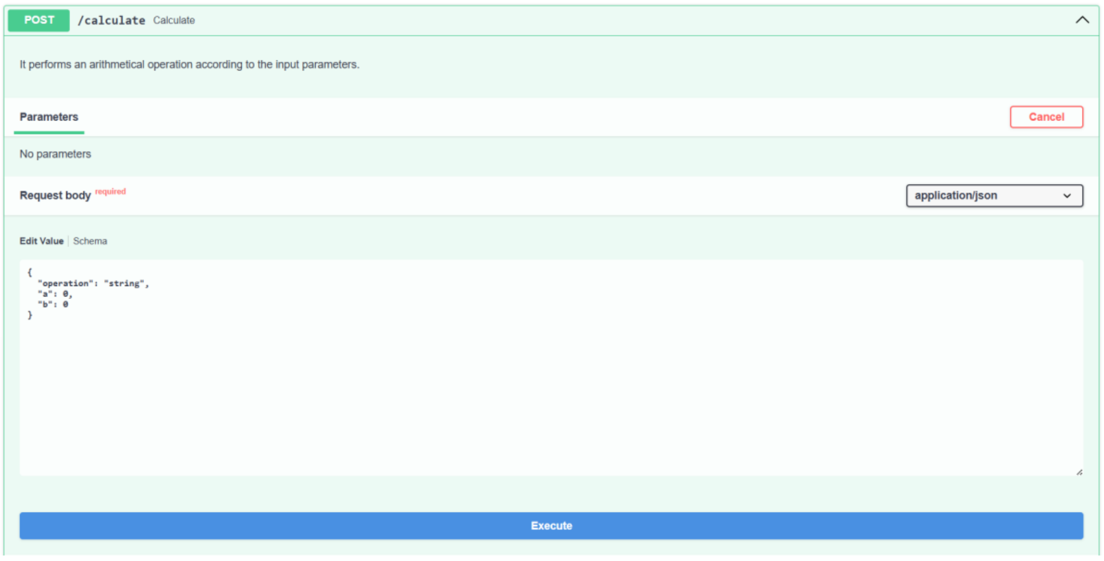
Schemas

CalcRequest > Expand all object

HTTPValidationError > Expand all object

ValidationError > Expand all object

If you press the POST option related to the */calculate endpoint*, you will see the interface associated with this *endpoint*. In this interface you have to press the *"Try it out"* button and you will see a JSON based-format text box (named *Edit Value* | *Schema*), like the one shown in the following image.


POST /calculate Calculate

It performs an arithmetical operation according to the input parameters.

No parameters

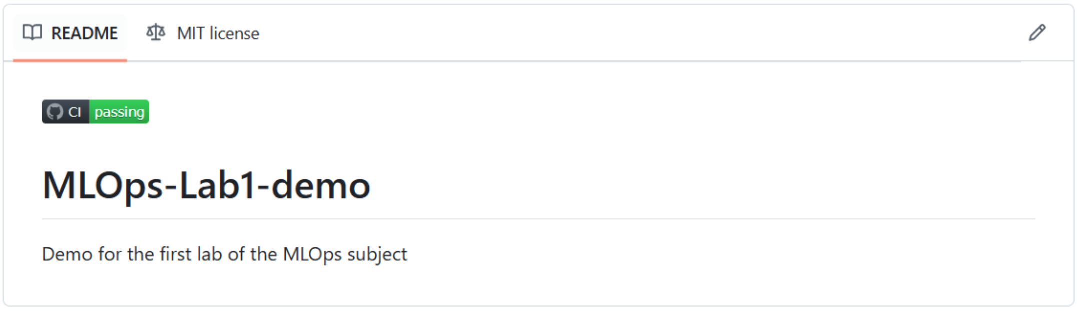
```
{
  "operation": "string",
  "a": 0,
  "b": 0
}
```

You have to change the information of this JSON based scheme so that you write the name of the arithmetical operation in the value of the *operation* key-value pair (add for instance), the values to be used by this operation in the *a* and *b* key-value pairs (2 and 3 for instance). Once you have determined these three values you have to press the *Execute* button. Once executed, ff you scroll down, you will find the *curl* command of this query and the result of the operation.

Finally, the CI pipeline was created. To do it, the *Actions* option of the GitHub repository was used. In this option, the *"set up a workload yourself"* option was used. When using this option, you use an editor to develop the CI pipeline, which is stored in a *.yml* file. You can see the CI pipeline created for this project in the *CI.yml* file stored in the GitHub Actions option of the repository. You can observe that you have to specify the name of the pipeline, when the CI pipeline is going to be triggered (*on*) and the jobs (*jobs*) to be made when the CI pipeline is triggered. In this example there is just one job named *build*, which runs on the *ubuntu-lastest* version and is composed of 5 steps: the first one is devoted to check the repository, and the remainder 4 ones are devoted to install, format, lint and test the code according to the commands established in the *Makefile*.

You can find information about the syntax of the CI pipelines developed with GitHub Actions at https://docs.github.com/es/actions/reference/workflows-and-actions/workflow-syntax. In this site you can see specific actions to be used to trigger the pipeline (on subsection), the job's definition (job subsection) and several other interesting options (env and permissions for instance). You can also see more options to be used to determine the jobs in https://docs.github.com/es/actions/how-tos/write-workflows/choose-what-workflows-do/use-jobs.

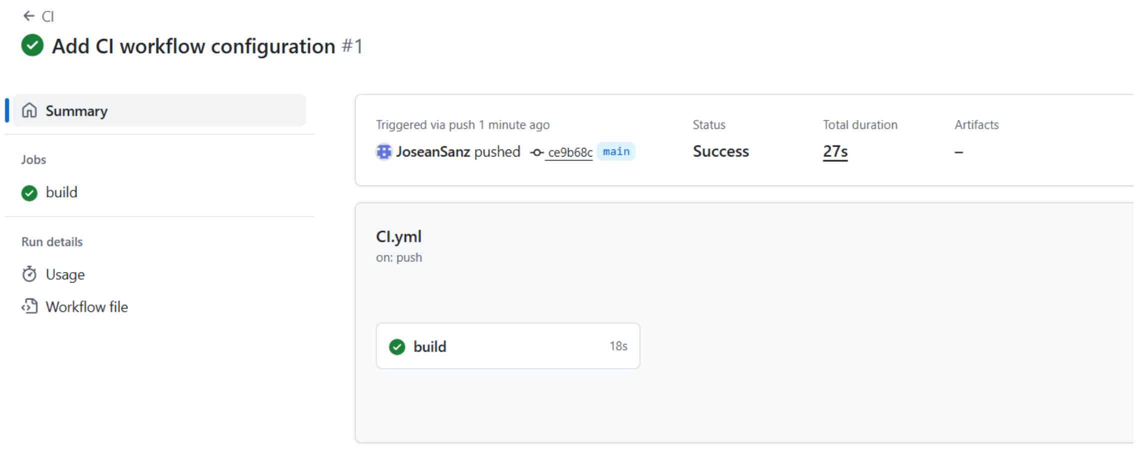
When you have the CI pipeline programed, if you click on the workflow name, you can press the three dots in the upper right corner, where you will see the *"Create status badge"* option. If you press it, you can copy its content and paste it in the README.md file so that everybody can see if your CI Pipeline is working fine (as shown in the next figure).



MLOps-Lab1-demo

Demo for the first lab of the MLOps subject

On the other hand, when the CI pipeline is ready, if you make some of the actions established to trigger the CI pipeline (*push* or *pull_requests* in this example), the CI pipeline will be triggered and the sequence of actions determined in the job will be automatically executed. If everything runs ok, you will obtain a green flag as in the following figure.



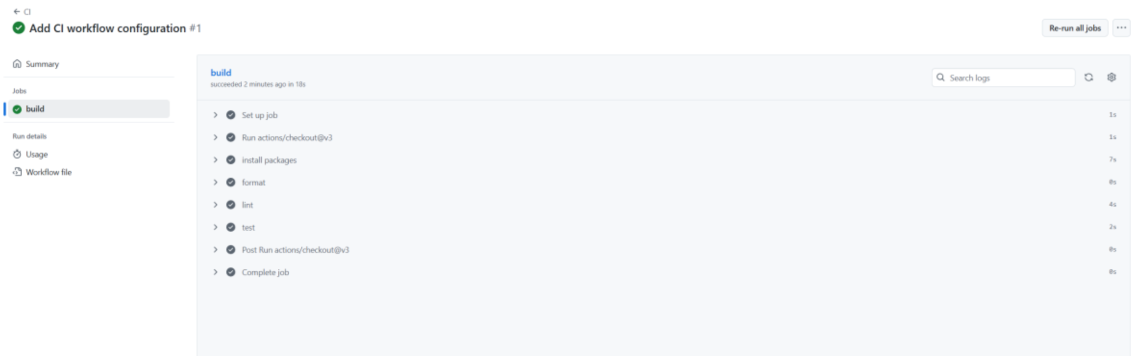
Triggered via push 1 minute ago
JoseanSanz pushed ce9b68c main

Status: Success
Total duration: 27s
Artifacts: –

CI.yml
on: push

build 18s

If you press the build button, you can find the flow of operations done to complete the CI workflow. You can click in each step and you will see the messages of each stage of the Pipeline.



build
succeeded 2 minutes ago in 18s

```
> Set up job                      1s
> Run actions/checkout@v3         1s
> install packages                7s
> format                          0s
> lint                            4s
> test                            2s
> Post Run actions/checkout@v3    0s
> Complete job                    0s
```

In case the CI pipeline does not run correctly, you will see a red flag (error). In this scenario, you can also click on the jobs name, as well as in the specific commands composing it, to see which one failed so that you can fix it.

In the API for this lab, you can avoid using a JSON-based text box to introduce the information as in the demo repository. Instead, you can create a form to specify the inputs to the system. To do it, you do not have to use a class inheriting from *BaseModel* as in the example. You can create directly the function associated with each endpoint and you can send as input arguments the following:

- For the image (or any file you want to upload to the system). If this input argument is named *file* you can define your function including
    - *file: UploadFile = File(...)*
        - A binary file (multipart/form-data) is received
- For the width and heigh you can use text boxes (using *Form(...)*). As these arguments are integers you can send them as (example for width):
    - *width: int = Form(...)*
- You have to import *UploadFile*, *File* and *Form* from *fastapi*

Obviously, in each function of the API you can check if the input arguments are valid (types) of if the file is loaded correctly, using try-except.

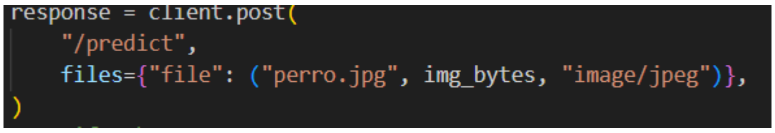
When FastAPI detects input parameters with *File()* or *Form(),* it expects the inputs is Content-Type (multipart/form-data). Then, the API client groups input of both types in two fields

- *files*: grouping all the inputs of type *File()*
    - For instance *files['file']* would have the image
- *data*: grouping all the inputs of type *Form()*
    - For instance *data['width']* would have the width to resize the image

Then, in the testing, you can test all the possible scenarios of wrong inputs parameters as well as the right ones. For the testing of the API, you have to use the *TestClient* imported from *fastapi.testclient* (you can use it as a fixture as made with the *CliRunner* for the testing of the *CLI*).

An important factor in performing the testing of the API is how to send the information to the endpoints because of the information expected for the FastAPI client

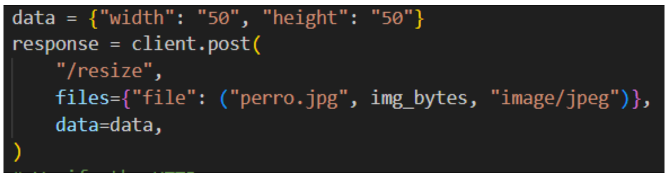
- Image: as mentioned before, *UploadFile* receives a binary file. Consequently, for the PIL library as example, you would have to
    - Read the image (*Image.open(path) as img*, path to an image you have in your repository using the *Path* function imported from *pathlib*
    - Convert it to RGB (*img.convert("RGB")*)
    - Create a buffer of bytes in memory: *img_bytes = io.BytesIO()*
    - Save this buffer as JPEG format: *img.save(img_bytes, format="JPEG")*
    - Return the pointer to the beginning of the buffer (otherwise an empty buffer is sent): *img_byter.seek(0)*
    - You will send the buffer in the value associated with the JSON entry for the file. For instance



```
response = client.post(
    "/predict",
    files={"file": ("perro.jpg", img_bytes, "image/jpeg")},
)
```

- Values of other parameters
    - You have to construct a dictionary composed of as many entries as inputs of *Form()* type

        - Key: name of the input argument of the endpoint
        - Value: string with the value of the input argument
    - You will send this dictionary to the endpoint. For instance



```python
data = {"width": "50", "height": "50"}
response = client.post(
    "/resize",
    files={"file": ("perro.jpg", img_bytes, "image/jpeg")},
    data=data,
)
```