

# Docker Tutorial

Murat Aydin

Contact : <https://www.linkedin.com/in/murat-aydin-87a6a7136>

## 1 Introduction

Docker is an open-source software which enables OS-level virtualization or abstraction. What it is doing is that it helps developers in simplifying the application development. As it is said in the official website, docker handles the boring, complicated, tedious setup, configurations and dependency problems while the developers can focus on coding and send their project to their friends, teachers and colleagues without worrying about their system configurations to run the project.

It is actually relatively simple to learn it however some concepts like images, containers, volumes etc. might be complicated for beginners. The aim of this tutorial is get the reader familiarize with such concepts, see various functionalities of docker and learn how to dockerize a machine learning model.

## 2 Main Concepts

### 2.1 Docker Images and Containers

A docker image is the ideal heaven (assuming that there exist different kind of heavens!?) of a developer where we have preconfigured server environments, fully packaged applications which we can use for our own applications.

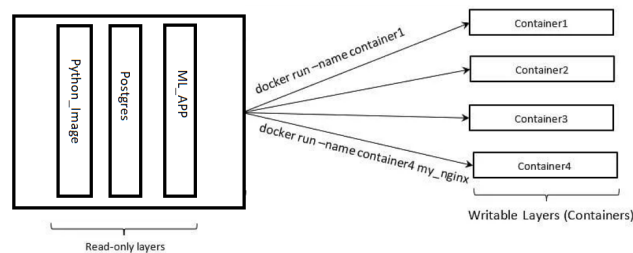


Figure 1: Docker Image Construction

Before getting into the details, let's get a step back and take a close look at the diagram in figure 1. On the left, we see a typical docker image with multiple read-only (immutable) layers.

And on the right, we see multiple runtime instances of this image. These instances are called docker containers. They are writable layers, mutable.

In this project, we will be doing something similar to the one in the diagram. Therefore, our final image consists of three different images. Python as a base image, postgres and a ML application that we programmed. So we pull the base image python from Dockerhub since we will be programming our ML model in python. Over it, we pull a postgres image which is another layer in our image and eventually we put our own application inside this image. Now the final image is a heaven for a machine learning application where data is acquired through postgres server. We do not have to deal with libraries, configurations, dependencies, operating systems or anything. Every single thing that our ML application will need is available inside this image. Our ML model will happily live inside it, train, test, do predictions. Thanks to our custom docker image, it will never ever need any external touch to find its way. That is what a docker image is and what it is doing.

Once we run this image(which we will see how to), we create a docker container. The container is a runtime instance of the docker image we created above. They are mutable. In the sense that, if there is something to change in the code, we can go inside them and modify it. But the image, once it is created, it cannot be modified.

## 2.2 Dockerfile

Dockerfile is simply a set of instructions written in a text file to build the docker image. These instructions are executed in the order in which they are written. These commands are used to assemble the final image. In other words, dockerfiles are used to create your own custom image using other images. Recall that in figure 1, we used 3 different images to create one single image which is our machine learning heaven where we only focus on code and do not have deal with anything else.

```
FROM jupyter/scipy-notebook

WORKDIR /project

COPY create_df.py ./create_df.py
COPY training_DT.py ./training_DT.py

RUN python3 create_df.py
```

Figure 2: Dockerfile example

We will go in details of dockerfile commands but just to get an idea, let's go through the dockerfile example in figure 2. Here with the FROM command, we choose a base image. In this case it is jupyter/scipy-notebook which includes python and some machine learning libraries(read online resources for more info). It then creates a working directory inside the image. Using COPY commands, we copy the py files in our physical machine to the docker image working directory.

We then using RUN command, we run an instance of python3 and as an argument we give createdf.py file to python.

Once this dockerfile is run, we will have an image where our happy py files live. They are happy because in this image they have every single thing they need in order to survive. Once we get the image, we can run a docker container with it and get the result of our program in the py files.

## 2.3 Basic Docker commands

We will go more in details and have a very detailed practical example but before going deeper, let's have a look at the basic commands.

- **docker build**: builds a custom image using a dockerfile.
- **Example** : `docker build -t my-project -f Dockerfile .`  
This command will create a docker image called my-project (-t flag is used to tag or name the image) using the Dockerfile given with the -f flag.
- **docker run** : runs an image and so it creates a container.
- **Example** : `docker run my-project`  
This command will create a docker container using the my-project image which is created in the previous step. Each container created will automatically get a name or you can give a custom name using `--name` command. It will also get an ID.
- **docker images** : lists the images you have already created. At this step, it will list only one image which is my-project.
- **docker ps** : lists the running containers.
- **docker ps -a** : lists the containers running or stopped.
- **docker stop [container-ID/Name]** : stops a running container.
- **docker rm [container-ID/Name]** : removes a stopped container. Note that running containers should be stopped before removed.
- **docker rmi [image-name]** : removes an image. Note that in order to remove an image, there should be no container running created through this image.
- **docker run -p [external-port-no:internal-port-no]** : When you run a container, the only way to access the process is from inside of it. To allow external connections to the container, you have to open (publish) specific ports.
- **Example** : `docker run -p 8080:5432 postgres`  
with this command, we run a postgres image. PostgreSQL's default port is 5432 and we map it to port 8080. However note that this command will result in an error saying that we did not specify postgres user and password information. I'm showing it here just to give as an example.  
What it means to map a port? Well, since we mapped the port from 5432 to 8080, using localhost and the port 8080, we can initiate a postgresql session in for example, say, jupyter notebook and use postgresql in python.

### 3 How to dockerize ML application

Before going into the details, I want to give you a brief overview of what we are doing in terms of machine learning. In our ML project, we will use famous Iris dataset and Adaboost algorithm to do multi-class classification. We have two py files used in the first part of the project. One is createdf.py and the other training-DT.py. Createdf.py loads the iris dataset and prepares it for machine learning model and training-DT.py is simply doing train and test splitting, model training and accuracy calculation. For the details and to see how it is done, please visit : [https://github.com/Murataydinunimi/Dockerize\\_Machine\\_learning](https://github.com/Murataydinunimi/Dockerize_Machine_learning)

#### 3.1 Part-1

Now that we know what is a docker image, container, dockerfile and also basic commands, we can see how to dockerize our ML application. Please open a terminal and execute the following commands:

```
C:\Users\Murat>cd Desktop
C:\Users\Murat\Desktop>mkdir docker-ml-project
C:\Users\Murat\Desktop>cd docker-ml-project
C:\Users\Murat\Desktop\docker-ml-project> mkdir part_1
C:\Users\Murat\Desktop\docker-ml-project>cd part_1
C:\Users\Murat\Desktop\docker-ml-project\part_1>
```

Figure 3: Commands to execute

So we first create a folder called docker-ml-project and then go in it and create another folder called part-1. Given the fact that we will slowly build up our application and since I want to show you each step in detail, we will need four parts. With the commands above, we will create the folder part-1 where we will put our dockerfile and py files to run our application.

Now go to my github account [https://github.com/Murataydinunimi/Dockerize\\_Machine\\_learning](https://github.com/Murataydinunimi/Dockerize_Machine_learning) and download createdf.py and training-DT.py from part-1 folder to your part-1 folder in your machine (do not download dockerfile because we will create it ourselves to get our hands dirty with docker). In part-1 folder and open a code editor(I use visual studio code and suggest to do, I will assume you are using it as well). Create a file and rename it as "Dockerfile" with capital D. You now should be able to see the blue sign of docker container. Great. We now need to give the instructions to build our image. We will give exactly the same instructions like in figure 2. We will get our base image from jupyter/scipy-notebook(Even if this is not efficient, we will see it later on) and will create a working directory inside the image called /project. Copy the python files inside our part-1 folder to the image working directory and Run createdf.py.

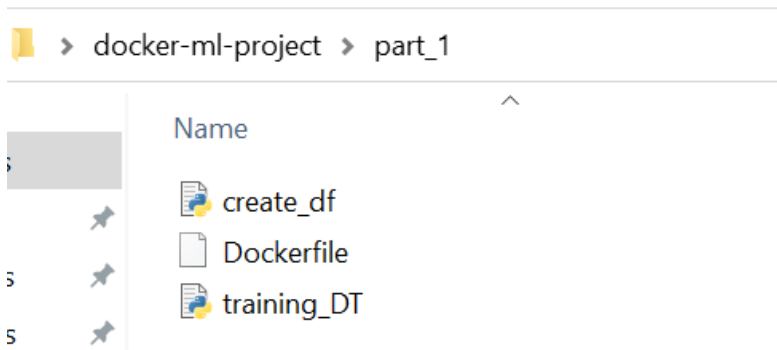


Figure 4: Files in part-1 folder

At this point, your part-1 folder inside docker-ml-project folder should look like this. Now open a terminal, go into the part-1 folder and execute the following command:

- `docker build -t my-project -f Dockerfile .`  
please mind the "." that comes after dockerfile.

After the process is completed, execute "docker images" command which lists the available images.

```
C:\Users\Murat\Desktop\docker-ml-project\part_1>docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
my-project latest 898f4ac3af49 15 minutes ago 3.03GB
```

Figure 5: docker images command output

If you had never created any docker image before, you now should have only one image available which what we called "my-project". It's name can be found under repository column. Tag is simply the version of our image. Given the fact that our image has only one version, the tag is latest. Image ID is the unique image ID assigned by docker. Size is around 3 GB which is big. Actually our ML application does not need such a big image. We will use a different base image to reduce the size because jupyter/scipy-notebook has many other tools that we do not use and are not necessary for this project.

Now, since our image is ready, we can eventually run a container. Recall that when our image is created, it automatically runs create-df.py file (see the last command in the dockerfile), so our iris dataframe is ready at the image directory /project, it is waiting to be used by the training-DT.py file for the adaboost algorithm.

Now go back to terminal and run :

- `docker run my-project python3 training-DT.py` ( you should use underscore when calling training-DT.py, latex requires mathematical notation for underscore so I do not use it)

```

C:\Users\Murat\Desktop\docker-ml-project\part_1>docker images
REPOSITORY      TAG         IMAGE ID      CREATED       SIZE
my-project       latest      898f4ac3af49  15 minutes ago  3.03GB

C:\Users\Murat\Desktop\docker-ml-project\part_1>docker run my-project python3 training_DT.py
Linux-5.10.16.3-microsoft-standard-WSL2-x86_64-with-glibc2.35
Python 3.10.6 | packaged by conda-forge | (main, Aug 22 2022, 20:36:39) [GCC 10.4.0]
NumPy 1.23.4
SciPy 1.9.3
Train data accuracy: 0.9375
Test data accuracy: 0.9736842105263158

C:\Users\Murat\Desktop\docker-ml-project\part_1>

```

Figure 6: docker run command output

As seen in figure 6, we get the train data and test data accuracy. We could successfully run our ML model using docker. To understand perfectly how the data is saved at docker directory and used back for the model, please the code in the github account.

Let's now list the containers we have:

- `docker ps`

This should return an empty list because we do not have any container running at the moment. The last container we created stopped working once it finished its job which was to execute training-DT.py.

- `docker ps -a`

This command should list one container with its automatically assigned name, its status etc.

```

C:\Users\Murat\Desktop\docker-ml-project\part_1>docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS              PORTS          NAMES
9117cc5ec3c6   my-project  "tini -g -- python3 ..."  10 minutes ago  Exited (0) 10 minutes ago          naughty_chatterjee

```

Figure 7: docker ps -a command

Let's now remove this container:

- `docker rm 9117`
- Providing first four digits in the container-ID is generally okay, otherwise you can add more digits or directly use the name. Note that container-ID might be different on your machine, you should use your own container ID or name.

Now let's recreate the container without executing any py file:

- `docker run my-project`

```

C:\Users\Murat\Desktop\docker-ml-project\part_1>docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
C:\Users\Murat\Desktop\docker-ml-project\part_1>docker run my-project
Entered start.sh with args: jupyter lab
Executing the command: jupyter lab
[I 2022-11-16 13:37:10.327 ServerApp] jupyterlab | extension was successfully linked.
[W 2022-11-16 13:37:10.339 NotebookApp] 'ip' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to
release.
[W 2022-11-16 13:37:10.340 NotebookApp] 'port' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to
release.
[W 2022-11-16 13:37:10.340 NotebookApp] 'port' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to
release.
[I 2022-11-16 13:37:10.351 ServerApp] nbclassic | extension was successfully linked.
[I 2022-11-16 13:37:10.352 ServerApp] Writing Jupyter server cookie secret to /home/jovyan/.local/share/jupyter/runtime/jupyter_cookiec
[I 2022-11-16 13:37:10.746 ServerApp] notebook_shim | extension was successfully linked.
[I 2022-11-16 13:37:10.786 ServerApp] notebook_shim | extension was successfully loaded.
[I 2022-11-16 13:37:10.788 LabApp] JupyterLab extension loaded from /opt/conda/lib/python3.10/site-packages/jupyterlab
[I 2022-11-16 13:37:10.788 LabApp] JupyterLab application directory is /opt/conda/share/jupyter/lab
[I 2022-11-16 13:37:10.800 ServerApp] jupyterlab | extension was successfully loaded.
[I 2022-11-16 13:37:10.807 ServerApp] nbclassic | extension was successfully loaded.
[I 2022-11-16 13:37:10.808 ServerApp] Serving notebooks from local directory: /project
[I 2022-11-16 13:37:10.809 ServerApp] Jupyter Server 1.23.2 is running at:
[I 2022-11-16 13:37:10.809 ServerApp] http://78b7b352de7e:8888/lab?token=1018a5b157aad443cd86e761c7a505c840ed50afb3373be0
[I 2022-11-16 13:37:10.810 ServerApp] or http://127.0.0.1:8888/lab?token=1018a5b157aad443cd86e761c7a505c840ed50afb3373be0
[I 2022-11-16 13:37:10.819 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 2022-11-16 13:37:10.816 ServerApp]

To access the server, open this file in a browser:
file:///home/jovyan/.local/share/jupyter/runtime/jpservice-7-open.html
Or copy and paste one of these URLs:
http://78b7b352de7e:8888/lab?token=1018a5b157aad443cd86e761c7a505c840ed50afb3373be0
or http://127.0.0.1:8888/lab?token=1018a5b157aad443cd86e761c7a505c840ed50afb3373be0

```

Figure 8: docker run my-project command

You should have an output like in figure 8 where the terminal is locked by the running container. Since we did not provide any files to execute like in the previous command, the container did not stop working and it is running. We can do ctrl+c to exit it which will cause the container to stop. Instead, let's open a new terminal and run:

- docker ps

```

C:\Users\Murat>docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
78b7b352de7e   my-project   "tiny -g -- start-no..."   2 minutes ago   Up 2 minutes (healthy)   8888/tcp   laughing_northcutt

```

Figure 9: docker ps command

See the Status column, it is up and running. Now let's stop it using the second terminal we opened and then remove it. Recall that if you want to remove a running container, you first need to stop it.

- docker stop 78b7
- docker rm 78b7
- docker ps – verify that no container is listed.

Let's now close the second terminal we opened, go back to the first one and run the container in the detached mode.

- docker run -d my-project
- docker ps — verify that the container is running but not occupying our terminal because it is running in the detached mode.

That is how you run a container in the detached mode. As I said before, images are immutable but containers are mutable. So imagine that we mistook in one of the lines of our training-DT.py file and we need to modify the code. We will now see how to modify a file inside a running container. Note that to do a modification inside a file of a container, you should make sure that the container is running.

- docker ps –note the container id

- `docker exec -u 0 -it container-id /bin/bash`

U flag is to get the root user privileges, it flag is for interactive mode. This will allow us to go into the container and have bash session there so that we can execute commands like `ls`, `mkdir` etc. Now you should have the following output:



```
C:\Users\Murat\Desktop\docker-ml-project\part_1>docker exec -u 0 -it 9f4a /bin/bash
(base) root@9f4a887fc2a8:/project#
```

Figure 10: docker exec command

We are now inside the container with the root user privileges.

- `ls`

Run the `ls` command to see the available files. You should have `create-df.py`, `Dockerfile`, `training-DT.py`. Let's go a step back in the folder path and see what we have.

- `cd ..`

You should now see many folders like `bin`, `boot`, `dev`, `home` etc... You should also see the project folder we created when setting up the image. Our `py` and `dockerfiles` are inside that folder.

- `cd project`
- `ls` – verify that files are there.
- `vi training-DT.py`

`Vi` command opens visual instrument editor. You can now modify the `training-DT.py` file using `Vi` editor and `save-exit`. Then you need to commit the changes.

- `docker commit [container-id] [new-image-name]`

With this modification we will be creating a new image, you can now run your modified container:

- `docker run new-image-name python3 training-DT.py`

We are done with the first part. Let's now remove all the images and the containers.

- `docker stop container-id`
- `docker rm container-id`
- `docker ps -a --verify`
- `docker rmi image-name`



## 3.2 Part-2

We are now in the second part where we will use a different image to run our model. Open a terminal and create a folder called part-2 inside docker-ml-project folder. Now go to my github account [https://github.com/Murataydinunimi/Dockerize\\_Machine\\_learning/tree/main/Part\\_2](https://github.com/Murataydinunimi/Dockerize_Machine_learning/tree/main/Part_2) and download everything to part-2 folder you just created. You should have create-df.py, training-Dt.py, requirements.txt, script.sh and a Dockerfile.

```
FROM python:3.8.5

WORKDIR /project

COPY requirements.txt ./requirements.txt
COPY create_df.py ./create_df.py
COPY training_DT.py ./training_DT.py
COPY script.sh ./script.sh
RUN pip install -r requirements.txt

CMD ["bash", "./script.sh"]
```

Figure 11: Dockerfile part-2

In this part of the project, we use a different base image which is python version 3.8.5 because jupyter/scipy-notebook was too big for us and we did not need many of tools existed there. The size of the image was around 3gb, while with this new custom image we will be around 1 gb. Since we are creating a custom image without anything but only with python, we need to download some libraries. All the libraries we need are listed in the requirements.txt file. Using RUN pip install -r requirements.txt we download all we need. When you have a long list of libraries to download, the best practice is to create a txt file for them.

We make another slight modification to show the functionality of CMD command. Open the script.sh file, inside we have two commands to run our py files. The CMD command specifies the instruction that is to be executed when a Docker container starts. So when the image is created, CMD function will start a bash shell as in the first parameter and then run the script.sh so that when we run the container, we will get the result of our model without actually running any command explicitly.

- docker build -t my-project -f Dockerfile .
- docker run my-project

You should first see that the image step by step is downloading everything we asked it to do and then when you run the container, script.sh file will be automatically executed and will print out the result. Let's now run:

- docker images

Notice that the image is around 1.35gb. We reduced the image size by more than half. Our process is now faster and the image is occupying less place in our machine.

### 3.3 Part-3

Let's now imagine that our iris dataframe lives in a postgresql server and we need to extract it from there and prepare for the model. So we need to use two different services: A database and our ML application because there is a dependency, a link among them. Our ML application is linked to postgresql server in that it needs the dataset that is living there. Obviously iris dataframe is just one table and does not need to be inside a relational structure but we can imagine ourselves working with a very complicated database where we need to extract data for our ML application.

Now open a terminal and create a folder called part-3 inside docker-ml-project folder. Go into this folder and download all the files from my github account [https://github.com/Murataydinunimi/Dockerize\\_Machine\\_learning/tree/main/Part\\_3](https://github.com/Murataydinunimi/Dockerize_Machine_learning/tree/main/Part_3). You should have two file here now, docker-compose.yml file and ml-codes. Inside ml-codes, you should have create-df.py, training-DT.py, post-py.py, requirements.txt, script.sh and command.txt. Please open post-py.py and see how the configurations between postgresql and python is done.

We will use docker-compose. Docker-compose is a tool that was developed to help define and share multi-container applications. With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

```
services:
  db:
    image: postgres:latest
    environment:
      - POSTGRES_USER=docker
      - POSTGRES_PASSWORD=my_password
    ports:
      - 5432:5432
  ml_app:
    build: ./ml_codes
    environment:
      - DB_USER=docker
      - DB_PASSWORD=my_password
      - DB_HOST=db
      - DB_PORT=5432
    ports:
      - 8080:8080
    links:
      - db
```

Figure 12: Docker-compose.yml file

In figure 12, you can see the internal structure of docker-compose.yml file. YAML is a human-readable data-serialization language. It is commonly used for configuration files and in applications where data is being stored or transmitted. Under the services key, we have two values db and ml-app. Db has three keys: image, environment and ports. Image is postgres since we want to use postgres as database. Environment is the environmental variables that

are required to configure the postgresql server. It has two values, user and password. Ports is mapped from 5432 to 5432 locally.

The other service we have is our ML application. Since this is not a ready image in Docker registry like postgresql, we need to build it ourselves through a dockerfile like we did so far. That's why here we have a build key with a value referring to the path of the dockerfile. It has an environment key as well like db service above because when configuring pythong to postgresql, we will need them. See the post-py.py file to understand better. The key thing here is the links key. Our ml-app service is linked to the db service which is a postgres image.

```
FROM python:3.8.5
WORKDIR /project

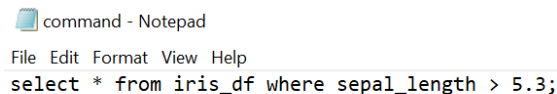
COPY requirements.txt ./requirements.txt
COPY create_df.py ./create_df.py
COPY training_DT.py ./training_DT.py
COPY post_py.py ./post_py.py
COPY command.txt ./command.txt
COPY script.sh ./script.sh

RUN pip install -r requirements.txt

CMD ["bash", "./script.sh"]
```

Figure 13: Dockerfile part3

As we can see from figure 13, in our dockerfile we have only one additional command with respect to the previous ones. We have copy command.txt command. Command.txt file contains the queries that is sent to postgresql server and it is in the following form:



```
File Edit Format View Help
select * from iris_df where sepal_length > 5.3;
```

Figure 14: Postgresql Query

In the post-py.py file, the database-manager class has the command attribute. This attribute reads this txt file and sends it to the postgresql server as a query which is processed and sent back to the training-DT.py file as an output for the ML model. For a better understanding of the internal structure, please see the post-py.py file.

Now, since we cleared up everything, open a terminal and cd into part-3 file where there is docker-compose.yml. Before doing anything remove any container and image just to have a clean memory. Execute the following command:

- docker-compose up

```

part_3-ml_app-1 | Linux-5.10.16.3-microsoft-standard-WSL2-x86_64-with-glibc2.2.5
part_3-ml_app-1 | Python 3.8.5 (default, Sep 10 2020, 16:47:10)
part_3-ml_app-1 | [GCC 8.3.0]
part_3-ml_app-1 | NumPy 1.23.4
part_3-ml_app-1 | SciPy 1.9.3
part_3-ml_app-1 | sklearn 1.1.3
part_3-ml_app-1 | data is saved at: /project
part_3-ml_app-1 | Linux-5.10.16.3-microsoft-standard-WSL2-x86_64-with-glibc2.2.5
part_3-ml_app-1 | Python 3.8.5 (default, Sep 10 2020, 16:47:10)
part_3-ml_app-1 | [GCC 8.3.0]
part_3-ml_app-1 | NumPy 1.23.4
part_3-ml_app-1 | SciPy 1.9.3
part_3-ml_app-1 | sklearn 1.1.3
part_3-ml_app-1 | Connection_String: postgresql://docker:my_password@db:5432/postgres
part_3-ml_app-1 | Connection to Postgre is successful
part_3-ml_app-1 | Data is added successfully
part_3-ml_app-1 | Changes are comitted successfully
part_3-ml_app-1 | ('sepal_length', 'double precision', 'YES')
part_3-ml_app-1 | ('sepal_width', 'double precision', 'YES')
part_3-ml_app-1 | ('petal_length', 'double precision', 'YES')
part_3-ml_app-1 | ('petal_width', 'double precision', 'YES')
part_3-ml_app-1 | ('target', 'bigint', 'YES')
part_3-ml_app-1 | The table is being printed
part_3-ml_app-1 | Command is executed successfully
part_3-ml_app-1 | The table is filtered with the following query: select * from iris_df where sepal_length > 5.3;
part_3-ml_app-1 | Connection is closed successfully
part_3-ml_app-1 | Postgre output is ready at: /project
part_3-ml_app-1 | The df has the following shape: (10, 5)
part_3-ml_app-1 | Train data accuracy: 0.987194871794872
part_3-ml_app-1 | Test data accuracy: 0.8846153846153846
part_3-ml_app-1 exited with code 0

```

Figure 15: Part-3 output

You should have the following output in your terminal. As you can see, test data accuracy is lower with respect to the previous ones because we filtered the data with a query where sepal-length is greater than 5.3 which disturbs the distribution of the observations. You can enter your own query in the command.txt file and see how the result changes.

As you can see, the terminal is occupied with the output. So you can run it in the detached mode as follows :

- `docker-compose up -d`

However this command will not provide you any logs. So right after the execution of this command is completed, run :

- `docker-compose logs -f [service-name]`

Where service-name is either db or ml-app, see the docker-compose.yml file.

### 3.4 Part-4

In part-4 which is the last one, we will see the concept of volumes. When a container is deleted, relaunching the image will start a fresh container without keeping the modifications made in the previously running container. The life cycle of the data in the container is ephemeral. When the container is dead, they are dead as well. We obviously do not like it.

In order to save(persist) the data and share between the containers, Docker comes up with the idea of volumes. In this project, we will see what is called bind mount.

Now create a folder called part-4 within docker-ml-project folder and download everything from part-4 folder in my github account [https://github.com/Murataydinunimi/Dockerize\\_Machine\\_learning/tree/main/Part\\_4](https://github.com/Murataydinunimi/Dockerize_Machine_learning/tree/main/Part_4) to your part-4 folder.

```

services:
  db:
    image: postgres:latest
    environment:
      - POSTGRES_USER=docker
      - POSTGRES_PASSWORD=my_password
    ports:
      - 5432:5432
  ml_app:
    build: ./ml_codes
    environment:
      - DB_USER=docker
      - DB_PASSWORD=my_password
      - DB_HOST=db
      - DB_PORT=5432
    volumes:
      - ./data_here:/project/data_here
    ports:
      - 8080:8080
    links:
      - db

```

Figure 16: Part-4 Docker-Compose Volumes

Notice the change in the volumes part under ml-app. This command will create a folder called "data-here" inside the part-4 folder and every data that is sent to the directory project/data-here in the docker image will be sent/mounted to our physical machine, in particular to the path docker-ml-project/part-4/data-here.

To make this possible, we need to do another modification in dockerfile and create-df.py

```
FROM python:3.8.5
WORKDIR /project
RUN mkdir /project/data_here

COPY requirements.txt ./requirements.txt
COPY create_df.py ./create_df.py
COPY training_DT.py ./training_DT.py
COPY post_py.py ./post_py.py
COPY command.txt ./command.txt
COPY script.sh ./script.sh

RUN pip install -r requirements.txt

CMD ["bash", "./script.sh"]
```

Figure 17: Part-4 Docker-Compose Volumes

Notice that we create another directory, in particular we create a folder within the /project path called data-here. And in the create-df.py, I simply added a line where I write the iris dataframe to /project/data-here path. If I did not use the volume, when the container removed, the data in this path would be gone but we are mounting it to data-here folder inside the physical machine so that once the process is completed, in our physical machine, independent of the container, we have the data available. Once you understand the modifications and why we needed them, run within the part-4 folder:

- docker-compose up

When the process is completed, inside the part-4 folder you should be able to see the data-here folder where there is the output.csv file which is the iris dataframe.