# Step 1: Create a Kubernetes cluster

First, we need to create a Kubernetes cluster with three nodes. If we have deleted the existing ones from previous assignments, we can start fresh. We will enable Google Kubernetes Engine (GKE) and create a Kubernetes cluster.

gcloud container clusters create kubia --num-nodes=1 --machine-type=e2-micro --region=us-west1

A group of lights on a black background

Description automatically generated

This step involves using Google Cloud's command-line tool to create a Kubernetes cluster named kubia with one node. The node uses the e2-micro machine type, and the cluster is set up in the us-west1 region. This command sets up the infrastructure needed to run containerized applications in a scalable way.

**Step 2: Verify the cluster and nodes**

Ensure that the Kubernetes cluster and nodes are properly created and operational.

gcloud container clusters list

A screen shot of a computer

Description automatically generated

Or, to run a single node using Minikube:

minikube start

A screenshot of a computer program

Description automatically generated

Here, we check the status of our GKE cluster by listing all clusters and verifying that they are running. Alternatively, we can use Minikube, a tool that allows you to run Kubernetes locally, to start a single-node cluster. This verification step is crucial to ensure that the environment is ready for deploying applications.

**Implementing Machine Learning Deployment Using Docker**

**Step 1: Train a Machine Learning model**

**A screenshot of a computer

Description automatically generated**

In this step, we use a dataset to train a machine learning model. The example shows loading the Iris dataset, splitting it into training and testing sets, training a logistic regression model, and then testing the model's accuracy. This trained model will later be saved and used for predictions.

**Step 2: Save and export the trained ML model**

Once the model is trained, save it using libraries like pickle or joblib for future use in predictions.

A screenshot of a computer

Description automatically generated Here, we save the trained machine learning model to a file using Python's pickle module. This allows us to persist the model to disk so that it can be reloaded and used later without having to retrain it. Saving the model is crucial for deploying it in a production environment.

**Step 3: Create a Flask app including the UI layer**

Build a Flask application with a user interface to deploy the machine learning model. This includes creating a Flask API that can serve predictions.

from flask import Flask, request

import numpy as np

import pickle

import pandas as pd

from flasgger import Swagger

app = Flask(\_\_name\_\_)

Swagger(app)

pickle\_in = open("logreg.pkl", "rb")

model = pickle.load(pickle\_in)

@app.route('/')

def home():

return "Welcome to the Flask API!"

@app.route('/predict', methods=["GET"])

def predict\_class():

"""Predict if Customer would buy the product or not.

---

parameters:

- name: age

in: query

type: number

required: true

- name: new\_user

in: query

type: number

required: true

- name: total\_pages\_visited

in: query

type: number

required: true

responses:

200:

description: Prediction

"""

age = int(request.args.get("age"))

new\_user = int(request.args.get("new\_user"))

total\_pages\_visited = int(request.args.get("total\_pages\_visited"))

prediction = model.predict([[age, new\_user, total\_pages\_visited]])

return "Model prediction is " + str(prediction)

@app.route('/predict\_file', methods=["POST"])

def prediction\_test\_file():

"""Prediction on multiple input test file.

---

parameters:

- name: file

in: formData

type: file

required: true

responses:

200:

description: Test file Prediction

"""

df\_test = pd.read\_csv(request.files.get("file"))

prediction = model.predict(df\_test)

return str(list(prediction))

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True, host='0.0.0.0', port=5000)

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

This code sets up a simple Flask web application that loads a pre-trained model and provides two endpoints: one for single predictions based on query parameters (/predict) and another for batch predictions using a file upload (/predict\_file). The use of Flasgger helps document the API and provides a UI to test it easily. Flask serves as the web framework, handling HTTP requests and returning responses with predictions.

**Step 4: Create a custom Dockerfile for the app**

Create a Dockerfile to build a Docker image for the Flask application, allowing it to run on any platform.

# Dockerfile for the Flask app

# Use an official Python runtime as a parent image

FROM python:3.8-slim

# Set the working directory in the container

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Install any needed packages specified in requirements.txt

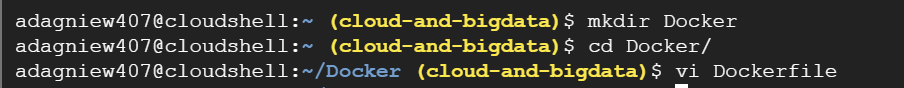
RUN pip install --no-cache-dir -r requirements.txt

# Make port 5000 available to the world outside this container

EXPOSE 5000

# Run app.py when the container launches

CMD ["python", "flask\_api.py"]



A screenshot of a computer program

Description automatically generated

This Dockerfile describes the steps needed to containerize the Flask application. It starts with a base image of Python 3.8, sets the working directory, copies the application code into the container, installs the required Python packages, exposes port 5000 for the Flask app, and specifies the command to run the app when the container starts. Docker allows the application to be packaged with all its dependencies, ensuring it runs consistently across different environments.

**Step 5: Create the requirements.txt file with the following contents. This file contains all**

the dependencies and libraries to be installed.



A screenshot of a computer program

Description automatically generated

sk==1.1.1

gunicorn==19.9.0

itsdangerous==1.1.0

Jinja2==2.10.1

MarkupSafe==1.1.1

Werkzeug==0.15.5

numpy>=1.9.2

scipy>=0.15.1

scikit-learn==0.22.1

matplotlib>=1.4.3

pandas>=0.19

flasgger==0.9.4

**Step 6: . Then we upload all the files we created before to the terminal which are logreg.pkl, ML.ipynb, and flask\_api.py files by clicking the three dots at the top-right of the Cloud Shell Terminal and then choose upload.**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

****

**Step 7: Run the app using a Docker container**

Build a Docker image from the Dockerfile and run the container, enabling the ML app to operate in a Dockerized environment.

# Build the Docker image

sudo docker build -t ml\_app\_docker .

A screen shot of a computer program

Description automatically generated

# Run the Docker container

sudo docker container run -p 5000:5000 ml\_app\_docker

A black screen with white text

Description automatically generated

We use Docker commands to build an image named ml\_app\_docker from the Dockerfile and then run a container from this image. The -p 5000:5000 flag maps port 5000 of the container to port 5000 of the host machine, making the Flask app accessible. Running the app in a Docker container ensures it is isolated and portable, making it easier to deploy in various environments.

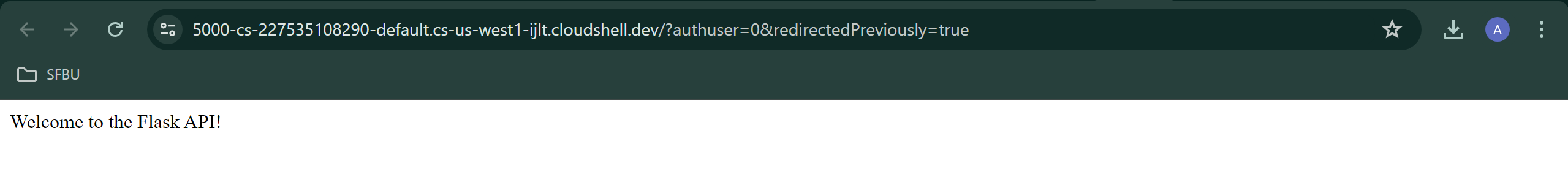
**Step 9:** In the upper right corner of the terminal, click the web preview option from the menu, and then select "Preview on port 5000." If the port is not set to 5000 by default, change it accordingly.

A screenshot of a search engine

Description automatically generated

A screenshot of a computer

Description automatically generated



After selecting "Change" and "Preview," you will see this message in the web preview.

**Step 10**: Add /apidocs/ at the end of the link to access to load the Swagger UI page.

A screenshot of a computer

Description automatically generated

There are two tabs: GET and POST. The GET tab is for single-customer predictions, while the POST tab is for predicting test data for customer groups

GET Tab: Allows for single-customer predictions. You can provide input parameters for the prediction.

POST Tab: Used for customer group predictions.

Try it out: Located in the top-right corner, it lets you fill in values for the input parameters.A screenshot of a computer

Description automatically generated

Fill values for the input parameters and then click Execute.

When you execute a call, the request is sent to the app, and the model generates predictions. The prediction result is then displayed in the "Prediction" section of the page.

A screenshot of a computer

Description automatically generated

Next, we will use a `test\_data.csv` file to make predictions for a group of customers through a POST request.

A screenshot of a computer

Description automatically generated

To upload the test data for predictions:

1. Click on the Post tab.

2. Click Try it Out.

3. Choose the `test\_data.csv` file.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A close-up of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

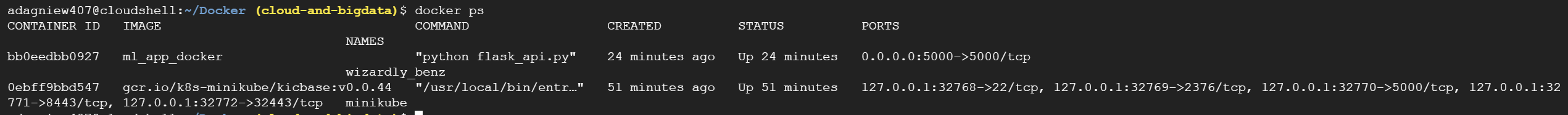
**Step 11: Stop the running container**

**Description:** Stop the Docker container running the application once it is no longer needed.

**Code:**

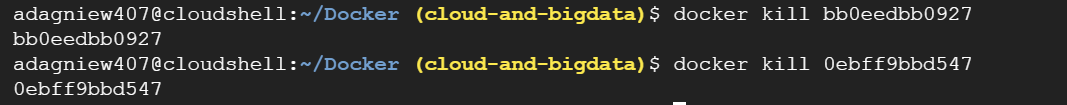
# List running containers

docker ps



# Stop the running container

docker kill bb0eedbb0927



To stop the running Docker container, we first list all active containers using docker ps to find the container ID. Then, we use the docker kill command with the container ID to stop the container. Stopping the container frees up system resources and ensures the application is no longer running when it's not needed.

These steps cover the entire process of setting up a Kubernetes cluster, training and saving a machine learning model, deploying it using a Flask app, Dockerizing the application, and managing the Docker container.