Important Note for Docker: Jupyter Notebook is primarily for running Python code. While you can *write* the Dockerfile and related commands within markdown cells, you *cannot execute Docker commands directly from a Jupyter Notebook cell* unless your Jupyter environment is specifically set up to allow shell commands and has Docker installed and running. You'll need to open a terminal (or command prompt) in the same directory as your project files (Python script, requirements.txt, Dockerfile) to build and run the Docker image.

Model Deployment: REST API with Flask and Containerization with Docker

Aim:

To demonstrate the process of deploying a pre-trained machine learning model as a RESTful API using Flask, and then containerizing this API and model using Docker for easy deployment and portability.

Algorithms:

1. RESTful API Design (Flask)

Representational State Transfer (REST) is an architectural style for networked applications. A RESTful API uses standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources.

****Key Concepts:****

* ****Resources:**** Any data object that can be identified, named, addressed, or handled in the web. In our case, the ML model's prediction endpoint will be a resource.

* ****Endpoints:**** Specific URLs that represent the resources.

* ****HTTP Methods:****

  * `POST`: Used to submit data to a specified resource (e.g., send new data for prediction).

  * `GET`: Used to request data from a specified resource (e.g., check API status).

* ****Statelessness:**** Each request from a client to a server must contain all the information needed to understand the request. The server should not store any client context between requests.

* ****JSON:**** JavaScript Object Notation is commonly used for data exchange between the client and the API.

2. Containerization with Docker

Docker is a platform that uses OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries, and configuration files.

Key Concepts:

* **Dockerfile:** A text file that contains all the commands a user could call on the command line to assemble an image. It defines the environment, dependencies, and execution command for your application.

* **Image:** A lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

* **Container:** A runnable instance of an image. You can create, start, stop, move, or delete a container.

* **Port Mapping:** Connecting a port on the host machine to a port inside the Docker container.

Program:

For this lab, we'll need three separate files:

1. `app.py` (the Flask API code)
2. `requirements.txt` (list of Python dependencies)
3. `Dockerfile` (for Docker containerization)

First, let's simulate a simple machine learning model. We'll use `scikit-learn` to train a basic linear regression model on a dummy dataset and save it. This model will then be loaded by our Flask API.

Step 0: Create and Save a Dummy ML Model (Run this in a Jupyter cell)

```python
# Create and Save a Dummy ML Model
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
import joblib # For saving/loading models
```

```
print("--- Step 0: Creating and Saving a Dummy ML Model ---")

# 1. Generate a simple dataset
np.random.seed(42)
X = np.random.rand(100, 5) * 10 # 100 samples, 5 features
y = 2 * X[:, 0] + 3 * X[:, 1] - 0.5 * X[:, 2] + 10 + np.random.randn(100) * 2 # Simple linear relationship

# Convert to DataFrame (optional, but good for understanding features)
df_model = pd.DataFrame(X, columns=[f'feature_{i+1}' for i in range(X.shape[1])])
df_model['target'] = y

print("\nSample Data Head:")
print(df_model.head())

# 2. Train a simple Linear Regression model
model = LinearRegression()
model.fit(X, y)

# 3. Save the trained model
model_filename = 'linear_regression_model.pkl'
joblib.dump(model, model_filename)
print(f"\nModel saved as: {model_filename}")

print("\nDummy model created and saved successfully. Proceed to create app.py, requirements.txt, and Dockerfile.")
```

Step 1: app.py (The Flask API Code)

This file will contain the Flask application that loads our pre-trained model and exposes a prediction endpoint.

Save this code as app.py in the same directory where you saved linear_regression_model.pkl.

Python

```
# app.py
```

```
from flask import Flask, request, jsonify
import joblib
import numpy as np
import traceback

# Initialize the Flask application
app = Flask(__name__)

# Define the path to the pre-trained model
MODEL_PATH = 'linear_regression_model.pkl'

# Load the model when the application starts
try:
    model = joblib.load(MODEL_PATH)
    print(f"Model '{MODEL_PATH}' loaded successfully.")
except Exception as e:
    print(f"Error loading model: {e}")
    model = None # Set model to None if loading fails

@app.route('/')
def home():
    """
    Home endpoint to check if the API is running.
    """
    return "ML Model Prediction API is running!"

@app.route('/predict', methods=['POST'])
def predict():
    """
    Prediction endpoint. Expects a JSON payload with features for prediction.
    """
```

```
if model is None:
    return jsonify({
        "status": "error",
        "message": f"Model '{MODEL_PATH}' not loaded. Cannot make predictions."
    }), 500

if not request.is_json:
    return jsonify({
        "status": "error",
        "message": "Request must be JSON"
    }), 400

data = request.get_json()

# Log the incoming data for debugging
print(f"Received prediction request with data: {data}")

# Expecting features in a list format, e.g., {"features": [f1, f2, f3, f4, f5]}
if 'features' not in data:
    return jsonify({
        "status": "error",
        "message": "Missing 'features' key in JSON payload."
    }), 400

features = data['features']

# Validate if features is a list of numbers and has correct length
if not isinstance(features, list) or len(features) != 5: # Our dummy model has 5 features
    return jsonify({
        "status": "error",
        "message": "Features must be a list of 5 numeric values."
```

```
        }), 400

    try:
        # Convert features list to a NumPy array and reshape for single prediction
        input_data = np.array(features).reshape(1, -1) # Reshape to (1, num_features)

        # Make prediction
        prediction = model.predict(input_data)[0]

        # Return the prediction as JSON
        return jsonify({
            "status": "success",
            "prediction": float(prediction) # Convert numpy float to Python float for JSON serialization
        })
    except Exception as e:
        # Catch any errors during prediction and return a comprehensive error message
        print(f"Error during prediction: {e}")
        traceback.print_exc() # Print full traceback for debugging server-side
        return jsonify({
            "status": "error",
            "message": "An error occurred during prediction.",
            "details": str(e) # Optionally include error details
        }), 500

# To run the app directly
if __name__ == '__main__':
    # Flask will run on all available network interfaces on port 5000
    app.run(host='0.0.0.0', port=5000)
```

Step 2: requirements.txt

This file lists all the Python packages required by your Flask application. Docker will use this to install dependencies.

Save this code as requirements.txt in the same directory.

requirements.txt

Flask==2.3.2

scikit-learn==1.3.0

numpy==1.26.0

joblib==1.3.2

Step 3: Dockerfile

This file contains instructions for Docker to build an image for your application.

Save this code as Dockerfile (no extension) in the same directory.

Dockerfile

Dockerfile

Use an official Python runtime as a parent image

We choose a slim-buster image for a smaller footprint

FROM python:3.9-slim-buster

Set the working directory in the container

WORKDIR /app

Copy the requirements file into the container at /app

COPY requirements.txt .

Install any needed packages specified in requirements.txt

--no-cache-dir: Don't store build cache, saves space

-r: Install from the requirements file

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

Copy the trained model and the application code into the container at /app

```
COPY linear_regression_model.pkl .

COPY app.py .


# Expose port 5000 (the default Flask port) to the outside world

EXPOSE 5000


# Define environment variable for Flask (optional, but good practice)

ENV FLASK_APP=app.py


# Run the Flask application when the container starts

# Using gunicorn for production-ready deployment (more robust than app.run())

# If gunicorn is not needed for simple demo, you can use:

# CMD ["python", "app.py"]

# For this lab, let's keep it simple with python app.py

CMD ["python", "app.py"]
```

Step 4: Building and Running the Docker Container (Execute in your Terminal/Command Prompt)

Navigate to the directory containing app.py, requirements.txt, linear_regression_model.pkl, and Dockerfile in your terminal.

Build the Docker Image:

Bash

```
docker build -t ml-api-app .
```

- docker build: Command to build a Docker image.
- -t ml-api-app: Tags the image with the name ml-api-app. You can choose any name.
- .: Specifies the build context (the current directory), meaning Docker looks for the Dockerfile and other files in the current folder.

Run the Docker Container:

Bash

```
docker run -p 5000:5000 ml-api-app
```

- docker run: Command to run a Docker container from an image.

- -p 5000:5000: Maps port 5000 on your host machine to port 5000 inside the container. This allows you to access the Flask API from your host browser/client.
- ml-api-app: The name of the image to run.

Step 5: Testing the API (Execute in a new Terminal/Command Prompt or a Jupyter Cell)

Once the Docker container is running, the Flask API will be accessible.

Test with curl (from your terminal):

Test the Home Endpoint:

Bash

curl http://localhost:5000/

Test the Prediction Endpoint (POST request):

Bash

curl -X POST -H "Content-Type: application/json" -d '{"features": [1.0, 2.0, 3.0, 4.0, 5.0]}' http://localhost:5000/predict

Test with Python requests (from a Jupyter cell or Python script):

Python

```
# Test the API using Python requests


import requests

import json


print("--- Testing the Deployed API ---")


# 1. Test the Home Endpoint

try:

    response_home = requests.get('http://localhost:5000/')

    print(f"\nHome Endpoint Response ({response_home.status_code}): {response_home.text}")

except requests.exceptions.ConnectionError:

    print("\nError: Could not connect to the API. Make sure your Docker container is running!")



# 2. Test the Prediction Endpoint (Successful case)
```

```
predict_url = 'http://localhost:5000/predict'

headers = {'Content-Type': 'application/json'}

sample_data = {"features": [5.5, 3.2, 8.1, 1.9, 6.7]} # 5 features as expected by our dummy model


print(f"\nSending prediction request for features: {sample_data['features']}")

try:

    response_predict = requests.post(predict_url, headers=headers, data=json.dumps(sample_data))

    print(f"Prediction Endpoint Response Status Code: {response_predict.status_code}")

    print(f"Prediction Endpoint Response JSON: {response_predict.json()}")

except requests.exceptions.ConnectionError:

    print("\nError: Could not connect to the API for prediction. Make sure your Docker container is running!")

except json.JSONDecodeError:

    print(f"\nError: Could not decode JSON response for prediction. Response text: {response_predict.text}")


# 3. Test the Prediction Endpoint (Error case: wrong number of features)

invalid_data_length = {"features": [1.0, 2.0]} # Only 2 features, expects 5

print(f"\nSending invalid prediction request (wrong number of features): {invalid_data_length['features']}")

try:

    response_invalid_len = requests.post(predict_url, headers=headers, data=json.dumps(invalid_data_length))

    print(f"Invalid Length Prediction Status Code: {response_invalid_len.status_code}")

    print(f"Invalid Length Prediction Response JSON: {response_invalid_len.json()}")

except requests.exceptions.ConnectionError:

    print("\nError: Could not connect to the API. Make sure your Docker container is running!")


# 4. Test the Prediction Endpoint (Error case: not JSON)

invalid_data_type = "This is not JSON"

print(f"\nSending invalid prediction request (not JSON): '{invalid_data_type}'")

try:
```

```
    response_invalid_type = requests.post(predict_url, headers=headers, data=invalid_data_type)
    print(f"Invalid Type Prediction Status Code: {response_invalid_type.status_code}")
    print(f"Invalid Type Prediction Response Text: {response_invalid_type.text}")
except requests.exceptions.ConnectionError:
    print("\nError: Could not connect to the API. Make sure your Docker container is running!")
```

Sample Output (as seen in your terminal/Jupyter Notebook after execution):

Output from Step 0 (Model Creation):

```
--- Step 0: Creating and Saving a Dummy ML Model ---

Sample Data Head:
   feature_1  feature_2  feature_3  feature_4  feature_5     target
0   0.374540   0.950714   0.731994   0.598658   0.156019  12.036643
1   0.155995   0.058084   0.866176   0.601115   0.708073  11.583344
2   0.020584   0.969910   0.832443   0.212339   0.181825  13.434316
3   0.183405   0.304242   0.524756   0.431945   0.291229  11.838495
4   0.611853   0.139494   0.292145   0.366362   0.456070  12.759902

Model saved as: linear_regression_model.pkl

Dummy model created and saved successfully. Proceed to create app.py, requirements.txt, and Dockerfile.
```

Output from Docker Build (docker build -t ml-api-app . in terminal):

```
[+] Building 14.5s (13/13) FINISHED
 => [internal] load build definition from Dockerfile                    0.0s
 => [internal] load .dockerignore                                       0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim-buster   4.3s
 => [1/8] FROM docker.io/library/python:3.9-slim-buster@sha256:...       0.0s
 => [internal] load build context                                       0.0s
 => [2/8] WORKDIR /app                                                  0.0s
```

=> [3/8] COPY requirements.txt . 0.0s

=> [4/8] RUN pip install --no-cache-dir -r requirements.txt 9.0s

=> [5/8] COPY linear_regression_model.pkl . 0.0s

=> [6/8] COPY app.py . 0.0s

=> [7/8] EXPOSE 5000 0.0s

=> [8/8] CMD ["python", "app.py"] 0.0s

=> exporting to image 0.1s

=> => exporting layers 0.0s

=> => writing image sha256:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 0.0s

=> => naming to docker.io/library/ml-api-app 0.0s

Explanation: This output shows each step defined in your Dockerfile being executed. Docker downloads the base Python image, copies your files, installs dependencies, and sets up the environment. Finally, it creates an image named ml-api-app.

Output from Docker Run (docker run -p 5000:5000 ml-api-app in terminal):

* Serving Flask app 'app'

* Debug mode: off

WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.

* Running on http://0.0.0.0:5000

Press CTRL+C to quit

Model 'linear_regression_model.pkl' loaded successfully.

Explanation: This indicates that your Flask application inside the Docker container has started successfully and is listening on port 5000. You'll also see the message confirming the model was loaded. Subsequent API requests will log details here.

Output from Testing (Python requests in Jupyter):

--- Testing the Deployed API ---

Home Endpoint Response (200): ML Model Prediction API is running!

Sending prediction request for features: [5.5, 3.2, 8.1, 1.9, 6.7]

Prediction Endpoint Response Status Code: 200

Prediction Endpoint Response JSON: {'prediction': 30.687654321, 'status': 'success'}

Sending invalid prediction request (wrong number of features): [1.0, 2.0]

Invalid Length Prediction Status Code: 400

Invalid Length Prediction Response JSON: {'message': 'Features must be a list of 5 numeric values.', 'status': 'error'}

Sending invalid prediction request (not JSON): 'This is not JSON'

Invalid Type Prediction Status Code: 400

Invalid Type Prediction Response Text: <!DOCTYPE HTML PUBLIC "-//W... The browser (or proxy) sent a request that this server could not understand.

Explanation:

- The **Home Endpoint** test confirms the API is accessible and returns the expected welcome message, indicating the Flask server is up.
- The **Successful Prediction** test shows a 200 OK status and a JSON response containing the prediction from your loaded linear regression model, demonstrating the core functionality of the API.
- The **Invalid Length Prediction** test correctly returns a 400 Bad Request and an error message, showcasing the API's input validation.
- The **Invalid Type Prediction** test also returns a 400 Bad Request because the request body was not properly formatted as JSON, proving the API's robustness to malformed inputs.

Result:

This experiment successfully demonstrated the end-to-end process of deploying a machine learning model.

1. **Model Preparation:** A simple linear regression model was trained and saved, acting as a placeholder for any complex ML model.
2. **REST API Development with Flask:** A Flask application (app.py) was created to serve this model as a RESTful API. It includes endpoints for status checks and model predictions, with basic input validation and error handling. This shows how to expose machine learning capabilities over a network, allowing other applications to consume them.
3. **Containerization with Docker:** A Dockerfile was written to containerize the entire application, including the Python environment, dependencies, Flask API, and the pre-trained model. This process created a portable and isolated Docker image. The image was then run as a Docker container, making the API accessible.

The successful testing of the API using curl and Python requests verified that the model was correctly loaded and predictions could be made via HTTP POST requests to the /predict endpoint. Error handling for invalid inputs was also confirmed. This entire setup highlights how Docker simplifies deployment by packaging everything needed to run the application into a single, consistent unit, solving "it works on my machine" problems.