Walkthrough: Preparing a model for deployment (Optional)

coursera.org/learn/foundations-of-ai-and-machine-learning/supplement/JfVUg/walkthrough-preparing-a-model-for-deployment-optional

Introduction

In this walkthrough, we will review the steps involved in the activity where you had to prepare an ML model for deployment. This guide will provide a detailed explanation of the correct approach to packaging, containerizing, and deploying the model, ensuring that it is ready for production. If you encountered any challenges during the lab, this walkthrough should help clarify the process and provide solutions.

By the end of this walkthrough, you will be able to:

- Package an ML model for deployment, including saving the model and defining dependencies.
- Create a RESTful API using Flask to serve the model.
- Containerize the model with Docker for consistent deployment.
- Test the model locally and understand the steps for cloud deployment.

1. Packaging the model

Step-by-step guide

Step 1: Save the trained model

To begin, you trained a simple logistic regression model on the Iris dataset and saved it using joblib. Here's the correct way to do this:

Train and save the model

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
import joblib

# Load the dataset
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)

# Train the model
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)

# Save the model to a file
```

```
joblib.dump(model, 'iris_model.pkl')
```

Explanation: This script trains a logistic regression model on the Iris dataset and saves the trained model to a file named iris_model.pkl using the joblib library. This file can then be loaded later to make predictions without retraining the model.

Step 2: Create the requirements.txt file

Next, you created a requirements.txt file listing all the dependencies your model needs. This ensures that the same environment can be recreated in production.

Example requirements.txt

```
numpy==1.21.2
scikit-learn==0.24.2
joblib==1.1.0
```

```
flask==2.0.2
```

Explanation: This file lists the specific versions of the libraries your model depends on, which ensures consistency between your development and production environments.

Step 3: Create a Python script for serving the model

You then created a serve_model.py script that uses Flask to serve your model as a RESTful API.

Correct script

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

app = Flask(__name__)

# Load the model
model = joblib.load('iris_model.pkl')

@app.route('/predict', methods=['POST'])
def predict():
    data = request.get_json(force=True)
    prediction = model.predict(np.array(data['input']).reshape(1, -1))
    return jsonify({'prediction': int(prediction[0])})
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=80)
```

Explanation: This script sets up a simple Flask web server that listens for POST requests on the /predictendpoint. It takes JSON input, uses the trained model to make a prediction, and returns the result as a JSON response.

2. Containerizing the model with Docker

Step-by-step guide:

Step 1: Write the Dockerfile

The Dockerfile is the script that defines how your Docker image will be built. Here's the correct Dockerfile you should have used:

Correct Dockerfile

```
# 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 80 available to the world outside this container

EXPOSE 80



# Run serve_model.py when the container launches

CMD [python, serve_model.py]
```

Explanation: This Dockerfile starts with a lightweight Python image (python:3.8-slim), sets up the working directory, installs the necessary dependencies from requirements.txt, exposes port 80, and finally runs the serve_model.py script to start the Flask server.

Step 2: Build the Docker image

You should have used the following command to build your Docker image:

Build command

```
docker build -t iris_model_image .
```

Explanation: This command builds the Docker image from the Dockerfile in the current directory (.) and tags it with the name iris_model_image.

Step 3: Run the Docker container

Once you built the image, you ran the Docker container using the following command:

Run command

```
docker run -d -p 80:80 iris_model_image
```

Explanation: This command runs the Docker container in detached mode (-d), mapping port 80 on your host to port 80 in the container. The container runs the Flask server, making your model available for predictions via the /predict endpoint.

3. Testing and deploying the model

Testing locally

After running the Docker container, you should have tested the model locally by sending a POST request to the /predict endpoint.

Testing command

```
curl -X POST -H Content-Type: application/json -d '{input:
[5.1, 3.5, 1.4, 0.2]}' http://localhost:80/predict
```

Expected response

```
{prediction: 0}
```

Explanation: This command sends a JSON payload containing the input features to the model’s prediction endpoint. The model returns a prediction based on the input, which should match the expected classification.

Optional: Deploying to a cloud environment

If you deployed your Docker container to a cloud environment such as Azure, you followed platform-specific instructions to push your Docker image to a container registry and deploy it to a container instance.

Conclusion

By following these steps, you have successfully prepared an ML model for deployment. This process involved packaging the model, containerizing it using Docker, and testing it in a local environment. Optionally, you could have deployed the model to a cloud environment for broader accessibility. Each step in this activity is crucial for ensuring that your model is robust, portable, and ready for production deployment.

If you encounter any issues, reviewing this walkthrough should help clarify the process and provide solutions to common challenges. Understanding these steps is essential for any data scientist or engineer looking to move their ML models from development to production.