Step 5: Professional Deployment with Docker

Tutor: Haythem Ghazouani

Contents

1 Introduction

1.1 Why Docker?

In this module, you built a Python backend and a React frontend. To run this project on another machine (e.g., your friend's laptop or a cloud server), you would normally have to:

1. Install Python 3.9
2. Install Node.js v18
3. Install dependencies ('pip install', 'npm install')
4. Solve OS-specific errors (Windows vs Linux paths)

Docker solves this by packaging the application and its entire environment (OS libraries, dependencies) into a lightweight unit called a **container**.

Concept: Image vs. Container

- **Image:** The blueprint (read-only). Think of it as a Class in Python. It contains the code and libraries.
- **Container:** The running instance. Think of it as an Object. You can run multiple containers from one image.

2 Phase 1: Dockerizing the Backend

We create a file named `Dockerfile.backend` in the root directory.

```
# 1. Base Image: Start with a lightweight Linux + Python 3.9
FROM python:3.9-slim

# 2. Workdir: Create a folder inside the container
WORKDIR /app

# 3. Dependencies: Install gcc for some ML libraries
RUN apt-get update && apt-get install -y build-essential

# 4. Requirements: Copy only the file first (for caching)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 5. Code: Copy your Python scripts and Data
COPY code/ ./code/
COPY data/ ./data/

# 6. Command: What runs when the container starts?
EXPOSE 8000
CMD ["python", "code/app.py"]
```

Common Pitfall

Order matters! We copy ‘requirements.txt‘ **before** the code. This uses Docker’s "Layer Caching" mechanism. If you change your code but not your dependencies, Docker skips the slow ‘pip install‘ step during the next build.

3 Phase 2: Dockerizing the Frontend

For React, we use a **Multi-Stage Build**. We don’t want to ship the heavy Node.js environment to production; we only want the compiled static files (HTML/CSS/JS).

```
# Stage 1: The Build Environment
FROM node:18-alpine AS build
WORKDIR /app
COPY frontend/package*.json ./
RUN npm install
COPY frontend/ ./
RUN npm run build
# This creates a 'dist/' folder with optimized files

# Stage 2: The Production Environment (Nginx)
FROM nginx:stable-alpine
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

4 Phase 3: Orchestration with Docker Compose

Instead of running two manual ‘docker run‘ commands, we use **Docker Compose** to define the entire system.

File: `docker-compose.yml`

```
version: '3.8'

services:
  # The Python Backend Service
  backend:
    build:
      context: .
      dockerfile: Dockerfile.backend
    ports:
      - "8000:8000"  # Host Port : Container Port
    volumes:
      - ./data:/app/data  # Persist data if container dies

  # The React Frontend Service
  frontend:
    build:
      context: .
      dockerfile: Dockerfile.frontend
```

```
    ports:
      - "5173:80"    # Access dashboard at localhost:5173
    depends_on:
      - backend      # Make sure backend starts first
```

5 Launching the Stack

5.1 Method 1: Command Line Interface

To build and start all three services (Backend, Frontend, MLflow) in the background:

```
docker-compose up --build -d
```

5.2 Method 2: Docker Desktop (GUI)

For a visual overview of your project:

1. **Containers Tab:** Open Docker Desktop and click "Containers".
2. **Stack View:** Find the group `datasciencefopython2`.
3. **Monitoring:** You can see the status of each service, click the names to see logs, or use the "Open in Browser" shortcuts for the Frontend.

6 Advanced Concepts: Ensembling

We have implemented **Stacking**, where multiple base models (RF, XGBoost) contribute to a meta-model (Logistic Regression). This typically provides more stable and accurate results for complex datasets like Bank Churn.

7 Useful Commands

7.1 View Specific Logs

```
docker-compose logs -f backend
```

7.2 Stop & Cleanup

```
docker-compose down
```

8 Troubleshooting

Issue: "Connection Refused" between Frontend and Backend.
Fix: In Docker, ‘localhost‘ refers to the container itself. To talk to another service, use its service name.
Example: In React, request `http://backend:8000` instead of `localhost:8000` (Note: This requires specific Nginx proxy setup for browser access, so for local dev, we stick to localhost mapping).