MLOps Project Report

Yunhang Chi, Wenxi Huang, Shiyu Mao, Zhongke Sun

April 10, 2025

1 Introduction

This report presents an MLOps pipeline built around the classic iris classification problem. The goal of this project was to demonstrate a practical approach to model training, evaluation, deployment, and basic versioning within a simplified Python ecosystem. Although the iris dataset is relatively small and straightforward, it offers a convenient framework for exploring key concepts like reproducible training, containerization, and command-line automation.

2 Workflow Design

A typical workflow begins by calling the training step. The user can install dependencies from requirements.txt and then invoke python model_training.py to produce a trained random forest model. If they wish to evaluate performance again or modify certain parameters, they can adjust the training script and re-run the process. For inference, python model_prediction.py loads the saved model and displays predictions for a sample input. Once satisfied with local results, the user can containerize the project. The project includes a Dockerfile that sets up a minimal Python environment, installs the required libraries, and defaults to running the model code upon container startup. With the Docker image built, CLI.py offers straightforward commands like build, push, pull, and run_model for sharing or launching the container.

3 Project Structure

The project follows a modular design to keep training, evaluation, inference, and data preprocessing tasks both transparent and extensible. Each stage is defined in a separate Python script:

preprocessing.py Loads and prepares the iris dataset for training and testing. It also includes an optional exploratory analysis function that generates basic visualizations such as scatterplots and correlation matrices.

model_training.py Implements a train_model() function that trains a random forest classifier, prints training duration, and saves the final model to a local file (model.pkl). Once the model is trained, the script triggers an evaluation routine to measure classification performance.

model_evaluation.py Defines evaluate_model(...), which computes metrics like accuracy, precision, recall, and F1 score. It also plots and saves a confusion matrix. The script includes per-class accuracy reports, serving as a basic illustration of how fairness or bias checks might be performed in a more complex scenario.

model_prediction.py Loads the persisted model file and demonstrates a simple inference process. In this example, it predicts the species of a single test instance, though the logic can be adapted to support batch or interactive prediction.

model_registry.py Illustrate a simple versioning workflow. Each time the register_model() function is called, it increments the version by counting existing "Version" lines in the registry log, logs metadata (path and timestamp) to model_registry.txt, and appends a "Version X" header and a separator line for clarity.

CLI.py Handles both container operations and local model tasks. Users can build, run, push, or pull a Docker image, and can also train or predict locally without invoking Docker.

get_path.py Defines relevant functions to dynamically generate and manage project directories, such as saving models, logs, and plots outputs.

A Appendix: Screenshots

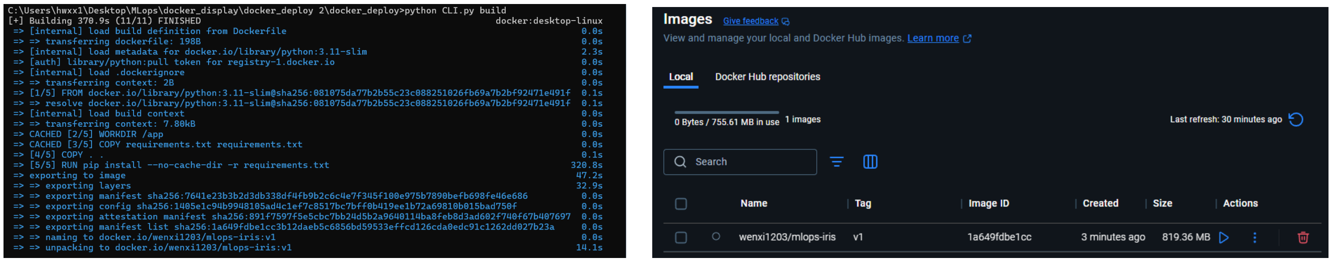

Figure 1: Build

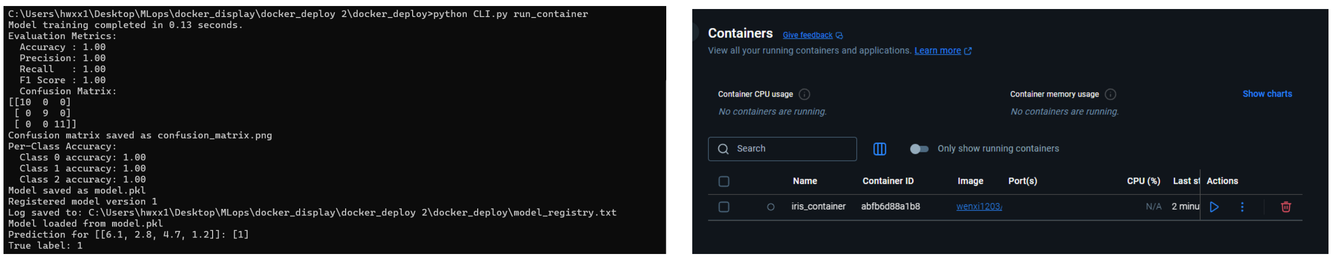

Figure 2: Run Container

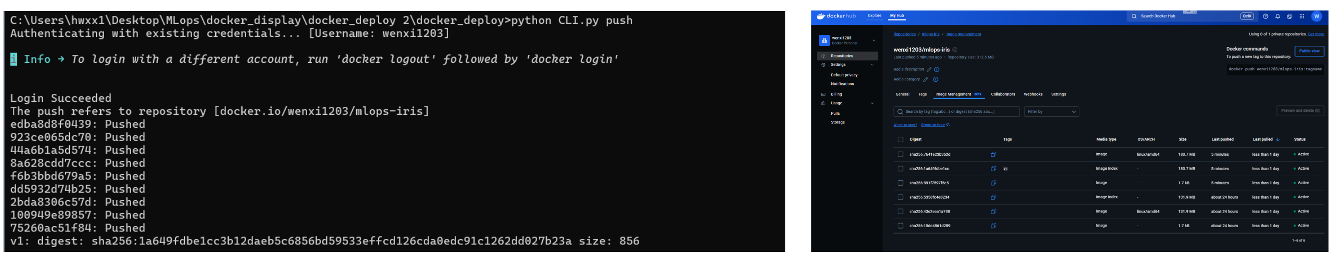

Figure 3: Push Docker Hub

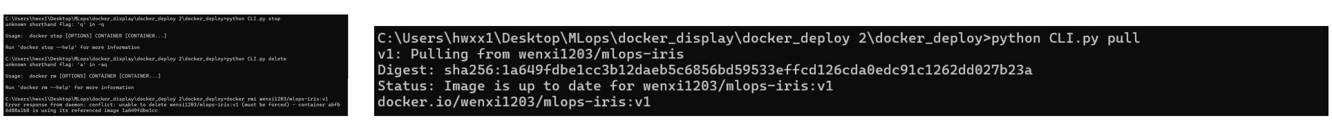

Figure 4: Delete Figure and Pull

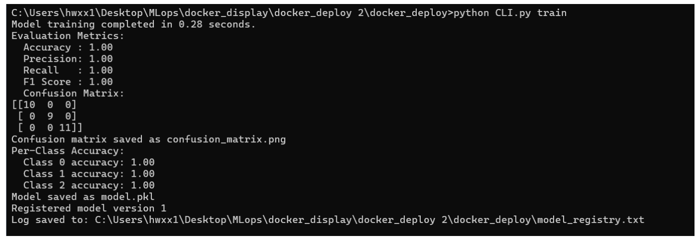

Figure 5: Run and Train the Model

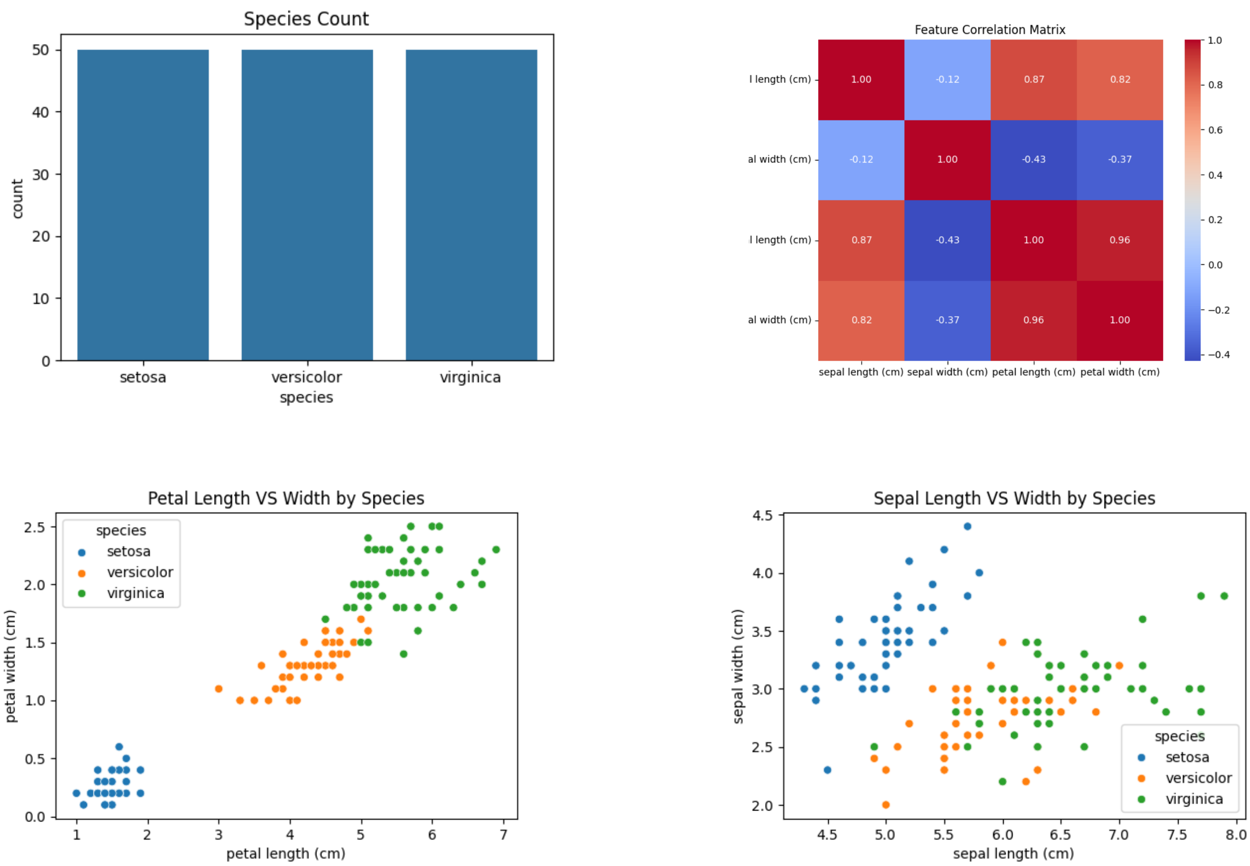

Figure 6: Visualization