PRACTICAL 1: Implement dataset versioning using DVC

GOAL: Version a dataset file using DVC and a local remote. **PRE-REQUISITE:** Git must be installed.

1. Setup Project Folders

```
mkdir dvc-practical
mkdir dvc-remote
cd dvc-practical
```

2. Create and Activate Virtual Environment

```
python -m venv venv
.\venv\Scripts\activate
```

3. Install Libraries

```
pip install dvc
```

4. Initialize Git and DVC

```
git init
dvc init
```

5. Configure a LOCAL remote

```
# Using '..' to go up one level from 'dvc-practical'
dvc remote add -d local_remote ..\dvc-remote
```

6. Create V1 of the dataset

```
echo "feature1,feature2,target" > data.csv
echo "1,2,0" >> data.csv
echo "3,4,1" >> data.csv
```

7. Track V1

```
dvc add data.csv
git add data.csv.dvc .gitignore
git commit -m "Add V1 of data.csv"
```

8. Push V1

```
dvc push -r local_remote
```

9. Create V2 of the dataset

```
echo "5,6,0" >> data.csv
echo "7,8,1" >> data.csv
```

10. Track V2

```
dvc add data.csv
git add data.csv.dvc
git commit -m "Add V2 of data.csv"
```

11. Push V2

```
dvc push -r local_remote
```

12. Simulate restoring data

```
del data.csv
dvc pull -r local_remote
type data.csv
# (This will show V2, the latest version)
```

13. Switch back to V1

```
git checkout HEAD~1 data.csv.dvc
dvc checkout
type data.csv
# (This will show V1)
```

14. Switch back to V2 (latest)

```
git checkout main data.csv.dvc
dvc checkout
type data.csv
# (This will show V2)
```

� PRACTICAL 2: Track experiments using MLflow

GOAL: Train a model and log its parameters and metrics to the MLflow UI.

1. Setup Project Folder

```
mkdir mlflow-practical
cd mlflow-practical
```

2. Create and Activate Virtual Environment

```
python -m venv venv
.\venv\Scripts\activate
```

3. Install Libraries

```
pip install mlflow scikit-learn pandas
```

4. Create Python file: `create_data.py`

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=100, n_features=10, n_informative=5, n_redundant=0, random_state=42)
df = pd.DataFrame(X, columns=[f'feature_{i}' for i in range(10)])
df['target'] = y

df.to_csv('dummy_data.csv', index=False)
print("Data created.")
```

5. Create Python file: `train.py`

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score
import mlflow
import mlflow.sklearn
import sys

# Set tracking URI to a local directory named 'mlruns'
mlflow.set_tracking_uri("file:./mlruns")

# Get C parameter from command line, default to 1.0
C_param = float(sys.argv[1]) if len(sys.argv) > 1 else 1.0

# Load data
df = pd.read_csv('dummy_data.csv')
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Start an MLflow run
with mlflow.start_run():
    print("Starting MLflow run...")
    # Train model
    model = LogisticRegression(C=C_param, max_iter=200, random_state=42)
    model.fit(X_train, y_train)

    # Make predictions
    preds = model.predict(X_test)

    # Calculate metrics
    accuracy = accuracy_score(y_test, preds)
    precision = precision_score(y_test, preds)

    # Log parameters
    mlflow.log_param("C", C_param)
    mlflow.log_param("model_type", "LogisticRegression")

    # Log metrics
    mlflow.log_metric("accuracy", accuracy)
    mlflow.log_metric("precision", precision)

    # Log the model
    mlflow.sklearn.log_model(model, "model")

    print(f"Run complete. Accuracy: {accuracy}")
```

6. Run setup and training

```
python create_data.py
python train.py 0.5
python train.py 1.0
python train.py 2.0
```

7. Launch MLflow UI

```
mlflow ui
```

> Open your browser to `http://127.0.0.1:5000`

PRACTICAL 3: Track experiments using Weights & Biases

GOAL: Train a model and log its parameters and metrics to the W&B dashboard. **PRE-REQUISITE:** You need a free W&B account (https://wandb.ai/site).

1. Setup Project Folder

```
mkdir wandb-practical
cd wandb-practical
```

2. Create and Activate Virtual Environment

```
python -m venv venv
.\venv\Scripts\activate
```

3. Install Libraries

```
pip install wandb scikit-learn pandas
```

4. Login to W&B

```
wandb login
```

> This will ask you to paste an API key from your W&B profile settings.

5. Create Python file: `create_data.py`

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=100, n_features=10, n_informative=5, n_redundant=0, random_state=42)
df = pd.DataFrame(X, columns=[f'feature_{i}' for i in range(10)])
df['target'] = y

df.to_csv('dummy_data.csv', index=False)
print("Data created.")
```

6. Create Python file: `train_wandb.py`

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import wandb

# Load data
df = pd.read_csv('dummy_data.csv')
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Configuration for the run
config = {
    "C": 1.0,
    "model_type": "LogisticRegression",
    "solver": "liblinear"
}

# 1. Initialize W&B run
wandb.init(project="practical-exam", config=config)

# 2. Train model
model = LogisticRegression(
    C=wandb.config.C,
    solver=wandb.config.solver,
    random_state=42
)
model.fit(X_train, y_train)

# Make predictions
preds = model.predict(X_test)
accuracy = accuracy_score(y_test, preds)

# 3. Log metrics
wandb.log({"accuracy": accuracy})

print(f"Run complete. Accuracy: {accuracy}")
wandb.finish()
```

7. Run setup and training

```
python create_data.py
python train_wandb.py
```

> Check the terminal output for the W&B link to your project dashboard.

� PRACTICAL 4: Register models using MLflow Model Registry

GOAL: Log a model to the MLflow UI and register it by name. **NOTE:** This builds directly on Practical #2.

1. Setup Project Folder (Same as Practical #2)

```
mkdir mlflow-practical
cd mlflow-practical
```

2. Activate Environment (If not already active)

```
.\venv\Scripts\activate
```

3. Ensure `create_data.py` exists (from Practical #2)

4. Create Python file: `register_model.py`

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import mlflow
import mlflow.sklearn

# --- This is the key model name for the registry ---
REGISTERED_MODEL_NAME = "PracticalExamModel"

mlflow.set_tracking_uri("file:./mlruns")

# Load data
df = pd.read_csv('dummy_data.csv')
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Start an MLflow run
with mlflow.start_run() as run:
    print("Starting MLflow run...")
    C_param = 0.75 # Using a different C value
    model = LogisticRegression(C=C_param, max_iter=200, random_state=42)
    model.fit(X_train, y_train)
    accuracy = accuracy_score(y_test, model.predict(X_test))

    # Log param and metric
    mlflow.log_param("C", C_param)
    mlflow.log_metric("accuracy", accuracy)

    # --- THIS IS THE KEY STEP ---
    # Log the model and register it under the specified name
    mlflow.sklearn.log_model(
        model,
        "model",
        registered_model_name=REGISTERED_MODEL_NAME
    )
    # --- END KEY STEP ---

    print(f"Run complete. Model registered as '{REGISTERED_MODEL_NAME}'")
```

5. Launch MLflow UI (in a separate terminal)

```
cd mlflow-practical
.\venv\Scripts\activate
mlflow ui
```

6. Run the registration script (in the first terminal)

```
python register_model.py
```

7. Check the MLflow UI

> Go to `http://127.0.0.1:5000`. Click the **"Models"** tab. You will see "PracticalExamModel". Click it to see **Version 1** and manage its stage (e.g., move to "Staging" or "Production").

⚙ PRACTICAL 5: Automate using Prefect

GOAL: Create a simple ETL and training pipeline using Prefect. **NOTE:** Prefect is simpler for local setup than Airflow.

1. Setup Project Folder

```
mkdir prefect-practical
cd prefect-practical
```

2. Create and Activate Virtual Environment

```
python -m venv venv
.\venv\Scripts\activate
```

3. Install Libraries

```
pip install prefect scikit-learn pandas
```

4. Create Python file: `pipeline.py`

```
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from prefect import task, flow

@task
def extract_data():
    """Generates dummy data and saves it."""
    X, y = make_classification(n_samples=100, n_features=10, random_state=42)
    df = pd.DataFrame(X, columns=[f'feature_{i}' for i in range(10)])
    df['target'] = y
    df.to_csv('dummy_data.csv', index=False)
    print("Task: Data extracted and saved.")
    return 'dummy_data.csv'

@task
def transform_data(data_path: str):
    """Loads and splits the data."""
    df = pd.read_csv(data_path)
    X = df.drop('target', axis=1)
    y = df['target']
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    print("Task: Data transformed.")
    return X_train, X_test, y_train, y_test

@task
def load_model(X_train, y_train, X_test, y_test):
    """Trains a model and prints accuracy."""
    model = LogisticRegression()
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    acc = accuracy_score(y_test, preds)
    print(f"Task: Model trained. Accuracy: {acc}")
    return acc

@flow(name="Practical Exam Flow")
def run_pipeline():
    """Main flow to run the ETL and training pipeline."""
    data_path = extract_data()
    X_train, X_test, y_train, y_test = transform_data(data_path)
    accuracy = load_model(X_train, y_train, X_test, y_test)
    print(f"Flow complete. Final accuracy: {accuracy}")

if __name__ == "__main__":
    run_pipeline()
```

5. Run the pipeline

```
python pipeline.py
```

6. (Optional) Start Prefect UI

```
# In a new terminal
prefect server start
```

> Go to `http://127.0.0.1:4200` to see the dashboard. Run `python pipeline.py` again, and your flow run will appear in the UI.

🐳 PRACTICAL 6: Build REST API (FastAPI) & Containerize (Docker)

GOAL: Train a model, wrap it in a FastAPI, and run it as a Docker container. **PRE-REQUISITE:** Docker Desktop must be installed AND RUNNING.

1. Setup Project Folder

```
mkdir fastapi-docker-practical
cd fastapi-docker-practical
```

2. Create and Activate Virtual Environment

```
python -m venv venv
.\venv\Scripts\activate
```

3. Install Libraries

```
pip install fastapi "uvicorn[standard]" scikit-learn pandas joblib
```

4. Create Python file: `create_model.py`

```
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
import joblib

X, y = make_classification(n_samples=100, n_features=4, n_informative=2, n_redundant=0, random_state=42)
model = LogisticRegression()
model.fit(X, y)

joblib.dump(model, 'model.joblib')
print("Model saved to model.joblib")
```

5. Create Python file: `main.py`

```python
from fastapi import FastAPI
from pydantic import BaseModel
import joblib
import numpy as np

app = FastAPI()

# Load the trained model
model = joblib.load('model.joblib')

# Define the input data model
class ModelInput(BaseModel):
    feature1: float
    feature2: float
    feature3: float
    feature4: float

@app.get("/")
def read_root():
    return {"message": "Model API is running."}

@app.post("/predict")
def predict(data: ModelInput):
    # Convert input data to numpy array
    input_data = np.array([[data.feature1, data.feature2, data.feature3, data.feature4]])

    # Get prediction and probability
    prediction = model.predict(input_data)[0]
    probability = model.predict_proba(input_data).max()

    return {
        "prediction": int(prediction),
        "probability": float(probability)
    }
```

6. Create file: `requirements.txt`

```
fastapi
uvicorn
scikit-learn
pandas
joblib
numpy
```

7. Create file: `Dockerfile`

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

# Copy the requirements file into the container
COPY requirements.txt .

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code
COPY . .

# Expose the port the app runs on
EXPOSE 8000

# Define the command to run the application
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

8. Run the model training script

```
python create_model.py
# (This creates model.joblib)
```

9. Test locally (Optional)

```
uvicorn main:app --reload
```

> Open `http://127.0.0.1:8000/docs` to test. Press `Ctrl+C` to stop.

10. Build the Docker image

```
docker build -t practical-api .
```

11. Run the Docker container

```
docker run -d -p 8000:8000 --name my-api practical-api
```

> Test the container at `http://127.0.0.1:8000/docs`

12. Stop and remove the container

```
docker stop my-api
docker rm my-api
```

□ PRACTICAL 7: Deploy with CI/CD (GitHub Actions)

GOAL: Automatically build the Docker image from Practical #6 when pushing to GitHub. **PRE-REQUISITES:**

- Project from Practical #6.
- A new, empty GitHub repository.
- Git installed.

1. Initialize Git in your project folder (from Practical #6)

```
cd fastapi-docker-practical
git init
git add .
git commit -m "Initial commit of FastAPI project"
```

2. Link to GitHub repository

```
# Get this command from your new GitHub repo page
git remote add origin https://github.com/YOUR_USERNAME/YOUR_REPO_NAME.git
git branch -M main
git push -u origin main
```

3. Create GitHub Actions workflow folder

```
mkdir .github
cd .github
mkdir workflows
cd workflows
```

4. Create file: `main.yml` (inside `.github/workflows`)

```
name: CI-CD Pipeline

on:
  push:
    branches: [ "main" ] # Trigger on push to main branch

jobs:
  build:
    runs-on: ubuntu-latest # Use a Linux runner

    steps:
    - name: Check out the repo
      uses: actions/checkout@v3

    - name: Set up QEMU
      uses: docker/setup-qemu-action@v2

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2

    - name: Log in to GitHub Container Registry
      uses: docker/login-action@v2
      with:
        registry: ghcr.io
        username: ${{ github.repository_owner }}
        password: ${{ secrets.GITHUB_TOKEN }} # This is automatically provided

    - name: Build and push Docker image
      uses: docker/build-push-action@v4
      with:
        context: . # Use the root of our repo
        push: true
        tags: ghcr.io/${{ github.repository_owner }}/practical-api:latest # Tag image
```

5. Commit and push the workflow file

```
cd ../..  # Go back to the project root (fastapi-docker-practical)
git add .github/workflows/main.yml
git commit -m "Add GitHub Actions CI/CD workflow"
git push origin main
```

6. Check the "Actions" tab

> Go to your GitHub repository in the browser. Click the **"Actions"** tab. You will see your workflow running. When it's done, check the **"Packages"** tab on your repo's main page to see your published Docker image.

PRACTICAL 8: Monitor model with Grafana (and Prometheus)

GOAL: Expose metrics from the FastAPI (Practical #6) and view them in Grafana. **PRE-REQUISITES:** Docker Desktop (with Docker Compose) must be installed AND RUNNING.

1. Setup Project Folder (Use Practical #6)

```
cd fastapi-docker-practical
.\venv\Scripts\activate
```

2. Install Prometheus instrumentator

```
pip install prometheus-fastapi-instrumentator
pip freeze > requirements.txt
```

3. Modify `main.py` to add the metrics endpoint

```
from fastapi import FastAPI
from pydantic import BaseModel
import joblib
import numpy as np
from prometheus_fastapi_instrumentator import Instrumentator # Import

app = FastAPI()

# --- ADD THIS BLOCK ---
@app.on_event("startup")
async def startup():
    """Instrument the app on startup."""
    Instrumentator().instrument(app).expose(app)
# --- END BLOCK ---

# Load the trained model
model = joblib.load('model.joblib')

# Define the input data model
class ModelInput(BaseModel):
    feature1: float
    feature2: float
    feature3: float
    feature4: float

@app.get("/")
def read_root():
    return {"message": "Model API is running."}

@app.post("/predict")
def predict(data: ModelInput):
    input_data = np.array([[data.feature1, data.feature2, data.feature3, data.feature4]])
    prediction = model.predict(input_data)[0]
    probability = model.predict_proba(input_data).max()
    return {
        "prediction": int(prediction),
        "probability": float(probability)
    }
```

4. Modify `Dockerfile` (No change needed if `requirements.txt` was updated)

> Just rebuild the image to include the new library.

```
docker build -t practical-api .
```

5. Create file: `prometheus.yml` (for Prometheus config)

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'fastapi-app'
    static_configs:
      # Use 'host.docker.internal' to allow Docker container
      # to access the API running on the host machine.
      - targets: ['host.docker.internal:8000']
```

6. Create file: `docker-compose.yml` (to run Grafana/Prometheus)

```
version: '3.8'

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'

  grafana:
    image: grafana/grafana-oss:latest
    container_name: grafana
    ports:
      - "3000:3000"
    depends_on:
      - prometheus
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin
```

7. Run everything

```
# Terminal 1: Run Docker Compose (Prometheus + Grafana)
docker-compose up

# Terminal 2: Run the FastAPI app (on the host)
.\venv\Scripts\activate
uvicorn main:app --host 0.0.0.0 --port 8000
```

8. Configure Grafana

- Open Grafana: `http://127.0.0.1:3000` (user: `admin`, pass: `admin`)
- **Add Data Source:**
  - Connections > Add new connection > Prometheus
  - URL: `http://prometheus:9090`
  - Click **"Save & Test"**.
- **Create Dashboard:**
  - Dashboards > New > Add visualization
  - Select "Prometheus" as data source.
  - In the "Metric" browser, try: `http_requests_total`
  - Click **"Run queries"**. You will see graphs.
- **Simulate Drift (Optional):** Send traffic to your API (e.g., using `curl` or Postman) to see the graphs change.

9. Stop everything

```
# In Terminal 1
docker-compose down
```

� PRACTICAL 9: Apply explainability using SHAP

GOAL: Train a model and use SHAP to explain its predictions.

1. Setup Project Folder

```
mkdir shap-practical
cd shap-practical
```

2. Create and Activate Virtual Environment

```
python -m venv venv
.\venv\Scripts\activate
```

3. Install Libraries

```
pip install shap scikit-learn pandas matplotlib
```

4. Create Python file: `create_data.py`

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=100, n_features=10, n_informative=5, n_redundant=0, random_state=42)
df = pd.DataFrame(X, columns=[f'feature_{i}' for i in range(10)])
df['target'] = y

df.to_csv('dummy_data.csv', index=False)
print("Data created.")
```

5. Create Python file: `explain.py`

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import shap
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv('dummy_data.csv')
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a model
model = RandomForestClassifier(random_state=42)
model.fit(X_train, y_train)
print(f"Model trained. Accuracy: {model.score(X_test, y_test)}")

# Initialize SHAP Explainer
# For tree models, use shap.TreeExplainer for speed
explainer = shap.TreeExplainer(model)

# Calculate SHAP values for the test set
shap_values = explainer.shap_values(X_test)

# Generate summary plot (beeswarm)
# shap_values[1] is for the "positive" class (class 1)
shap.summary_plot(shap_values[1], X_test, show=False)

# Save the plot to a file
plt.savefig('shap_summary.png', bbox_inches='tight')
print("SHAP summary plot saved to shap_summary.png")

# Generate plot for a single prediction
plt.clf() # Clear the previous plot
shap.force_plot(explainer.expected_value[1], shap_values[1][0,:], X_test.iloc[0,:], matplotlib=True, show=False)
plt.savefig('shap_force_plot.png', bbox_inches='tight')
print("SHAP force plot saved to shap_force_plot.png")
```

6. Run setup and explanation

```
python create_data.py
python explain.py
```

> Check the folder for `shap_summary.png` and `shap_force_plot.png`