Task 4 – Model Research and Evaluation

Data Bytes – Project Echo - Engine Team

**1. Executive Summary**

* Project Echo’s Engine Team needs a centralised experiment-tracking and model-registry solution to replace its manual TensorFlow save/load workflow, automate version control, and improve CI/CD integration and team collaboration under on-prem security constraints.
* Eight tools (MLflow, DVC, Guild AI, Metaflow, W&B, Comet, Neptune.ai, Azure ML) were evaluated against feature set, usability, integration, performance, cost, community support and maintenance.
* MLflow and DVC scored highest: MLflow for its end-to-end tracking, native registry and UI; DVC for its Git-native data/pipeline versioning and lightweight CLI.  
  Recommend adopting MLflow as the primary platform, with DVC as a complementary Git-centric tool for dataset and pipeline management.

**2. Context & Requirements**

**2.1 Problem Space**

Echo Engine’s existing codebase relies extensively on **TensorFlow** (with limited **PyTorch** usage) and employs direct file-based saving/loading for models. It lacks any dedicated experiment tracking or registry tool, as indicated by the absence of references to MLflow, Weights & Biases, or similar frameworks. Consequently, **version control** for model iterations and **reproducibility** of experiments is largely manual and decentralised. This setup constrains collaboration, hinders easy rollback to previous model states, and complicates performance comparisons across different training runs.

**2.2 Echo Engine’s Operational Needs**

1. **Model Lifecycle Management**  
   The code demonstrates repeated model.save(...) and tf.keras.models.load\_model(...) calls with various naming conventions. A registry should automate this process, storing model artefact metadata (e.g., date, hyperparameters, performance metrics) for each version.
2. **CI/CD Integration**  
   Although Docker and docker-compose are prevalent, there is no mention of automated pipelines (Airflow, Prefect, etc.). Any chosen tool should easily plug into containerised workflows and support future expansions into CI/CD for model deployment and validation.
3. **Collaboration Features**  
   Multiple environment YAMLs hint at distributed teams or varied dev setups. A registry must enable shared experiment logs, ensuring consistent references to model versions without requiring manual naming or file handling.
4. **On-Prem vs Cloud**  
   Hardcoded credentials (mongodb://root:root\_password@...) and concerns about exposing data externally suggest an **on-prem or private cloud** preference for compliance and data governance.
5. **Security/Compliance**  
   JWT-based user auth indicates role-based access requirements for model data. The solution must respect user roles, handle sensitive data, and potentially integrate with existing authentication layers.

**2.3 Evaluation Criteria**

1. **Feature Set** – Must support experiment logs (hyperparams, metrics, artefacts), model registry, lineage tracking.
2. **Usability** – Clear UI/CLI for data scientists, devops, and researchers.
3. **Integration Capability** – Python SDK, Docker-based workflows, minimal friction with TF and Torch.
4. **Performance/Scalability** – Efficient handling of large model files and frequent experiment logs.
5. **Cost** – Transparent licensing (open-source vs enterprise).
6. **Open-Source/Community Support** – Active development, frequent releases, large user base.
7. **Maintenance/Updates** – Ease of upgrades, available support channels, straightforward installations and patches.

**3. Tool Shortlist**

Below are **eight** representative tools to consider for model registry and experiment tracking. They include both open-source and commercial options, providing a broad view of maturity and ecosystem fit:

1. **MLflow (Open-Source)**
   * **Purpose & Fit**: End-to-end platform for experiment tracking, model packaging, and deployment. Integrates seamlessly with Python ML libraries and supports both TensorFlow and PyTorch out of the box.
   * **Maturity**: Backed by Databricks with a large OSS community; frequent releases and strong plugin ecosystem.
2. **DVC (Open-Source)**
   * **Purpose & Fit**: Think “Git for data” with experiment versioning. Strong synergy with code version control (Git) and can handle large files.
   * **Maturity**: Actively developed; widely adopted for dataset and ML pipeline versioning. Less emphasis on UI-based experiment tracking but excellent CLI.
3. **Guild AI (Open-Source)**
   * **Purpose & Fit**: Simple CLI-based experiment tracker that automatically captures run metadata, hyperparameters, logs, and model output. Integrates with any Python ML code.
   * **Maturity**: Lightweight, community-driven; lacks some advanced registry features but excels at quick experiment capture.
4. **Metaflow (Open-Source)**
   * **Purpose & Fit**: Developed by Netflix, it focuses on pipeline orchestration with experiment tracking. Emphasis on reproducible data science at scale.
   * **Maturity**: Large developer community, stable enterprise features. More pipeline-oriented than pure model registry.
5. **Weights & Biases (Commercial)**
   * **Purpose & Fit**: Cloud-based SaaS for experiment tracking, hyperparam sweeps, data versioning, and deep analytics. Strong TensorFlow/PyTorch integration.
   * **Maturity**: Rapid growth, robust UI, real-time collaboration features. Free tier available for small teams, but enterprise features cost.
6. **Comet (Commercial)**
   * **Purpose & Fit**: Cloud-based experiment tracking and model registry with real-time metric visualisations, code diffs, and collaboration.
   * **Maturity**: Well-established user base, varied pricing tiers. Good developer experience with Python SDK.
7. **Neptune.ai (Commercial)**
   * **Purpose & Fit**: Comprehensive experiment management, model registry, and metadata store. Strong project organisation features for large teams.
   * **Maturity**: Competitive with W&B. Offers on-prem or cloud hosting for compliance.
8. **Azure ML (Commercial)**
   * **Purpose & Fit**: Full-service ML platform by Microsoft. Provides experiment tracking, model registry, and deployment tooling tightly integrated with Azure cloud.
   * **Maturity**: Enterprise-grade, but heavily tied to Azure ecosystems. Potential licensing/cost overhead if you’re not already in Azure stack.

These eight candidates collectively represent the primary categories (pure open-source, hybrid OSS-commercial, and fully managed enterprise) to compare and contrast in **feature sets, cost, integration, and maintainability**.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Tool** | **Core Tracking (Params/**  **Metrics/**  **Artifacts)** | **Model Registry Support** | **UI/UX Quality** | **API/SDK Flexibility** | **Team Collab**  **oration** | **CI/CD Integration** | **Deployment Tracking** | **Cost/Licensing** | **Customisability** |
| **MLflow** | Full suite with automatic logging | Native registry with versioning/staging | Functional UI | Python-centric | Basic | Git integration | MLflow Models deployment | Open-Source | Plugin ecosystem |
| **DVC** | Metrics/params via CLI | Studio registry with lifecycle management | CLI-first + Studio | CLI/Python | Git-based | GitHub Actions compatible | Pipeline-based | Open-Source | Extension system |
| **Guild AI** | Auto-captured metadata | No native registry | Basic dashboards | CLI/R API | Limited | Limited | None | Open-Source | Minimal |
| **Metaflow** | Artifact checkpointing | Pipeline-oriented tracking | AWS Console UI | Python SDK | Team namespaces | AWS Step Functions | Cloud deployment | Open-Source | Checkpoint plugins |
| **W&B** | Real-time metric streaming | Model versioning with stages | Premium dashboard | Python/JS | Advanced sharing | Custom webhooks | Cloud endpoints | Freemium SaaS | Custom reports |
| **Neptune** | Rich metadata tracking | Lightweight registry with tags | Organized UI | Python/API | Project sharing | API triggers | Deployment monitoring | Paid/On-prem | Flexible metadata |
| **Azure ML** | Native Azure integration | Enterprise registry service | Azure Portal UI | Python/.NET | AD integration | Azure DevOps | Azure deployment | Enterprise SaaS | Azure extensions |

**4. Feature Comparison Matrix**

**Key Insights by Tool**

1. **MLflow**   
   **Pros**: Full lifecycle management, strong OSS community, model staging  
   **Cons**: Requires self-hosting for team features
2. **DVC**   
   **Pros**: Git-native data versioning, pipeline management  
   **Cons**: No built-in experiment comparison UI
3. **Guild AI**   
   **Pros**: Zero-config experiment capture  
   **Cons**: No model registry, basic visualization
4. **Metaflow**   
   **Pros**: Netflix-proven at scale, resumeable flows  
   **Cons**: AWS-centric, limited registry
5. **Weights & Biases**   
   **Pros**: Real-time collab, model hyperlinking  
   **Cons**: Cost escalates with usage
6. **Neptune**   
   **Pros**: Metadata flexibility, compliance-ready  
   **Cons**: Registry as secondary feature
7. **Azure ML**   
   **Pros**: End-to-end Azure integration  
   **Cons**: Vendor lock-in, complex pricing

**Shortlist Selection**

MLflow

DVC

**4. MLflow Integration**

MLflow is an open-source platform for managing the end-to-end machine learning lifecycle. It includes components for tracking experiments, packaging code into reproducible runs, and sharing and deploying models.

**1. Setup and Configuration**

**Installation:** Install MLflow using pip. For a more comprehensive setup including scikit-learn support, you can install with extras.

pip install mlflow[extras]

# Or for a minimal installation

# pip install mlflow

**Configuration (Local Tracking):** By default, MLflow logs data to a local ./mlruns directory. For more persistent or collaborative setups, you'd configure a remote tracking server (e.g., HTTP, Databricks, or a database). For this example, we'll use the local default.

To view the MLflow UI, navigate to your project directory in the terminal and run:

mlflow ui

This will start a local server, typically at http://127.0.0.1:5000 or <http://localhost:8080>.

**2. Code Snippets for Basic Usage**

The following Python script demonstrates basic MLflow usage within a hypothetical model training script for Project Echo.

**echo\_train\_mlflow.py:**

import mlflow

import mlflow.sklearn # MLflow integration for scikit-learn

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

import pandas as pd

import numpy as np

# --- 1. Project Echo: Setup Experiment ---

# Set an experiment name. If it doesn't exist, MLflow creates it.

# All runs will be logged under this experiment.

try:

mlflow.set\_experiment("Echo\_Engine\_Experiments")

except mlflow.exceptions.MlflowException:

print("Could not set experiment, ensure MLflow server is accessible or local ./mlruns is writable.")

# Fallback or exit if necessary

# --- 2. Project Echo: Start an MLflow Run ---

# Use a context manager to ensure the run is always closed.

with mlflow.start\_run(run\_name="Echo\_Model\_Training\_Run\_RF") as run:

run\_id = run.info.run\_uuid

print(f"MLflow Run ID: {run\_id}")

# --- 3. Project Echo: Log Parameters ---

# Example parameters for the Echo model

params = {

"n\_estimators": 150,

"max\_depth": 10,

"random\_state": 42,

"criterion": "gini"

}

mlflow.log\_params(params)

print("Logged parameters:", params)

# --- 4. Project Echo: Load Data (Dummy Example) ---

# Replace with your actual data loading for Project Echo

X = pd.DataFrame(np.random.rand(100, 5), columns=[f"feature\_{i}" for i in range(5)])

y = pd.Series(np.random.randint(0, 2, 100))

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=params["random\_state"])

# Log a sample of the training data or its characteristics if small enough and relevant

# For large datasets, consider logging descriptive statistics or a hash.

# mlflow.log\_input(mlflow.data.from\_pandas(X\_train.head(), source="X\_train\_head.csv"), context="training\_input\_sample")

# --- 5. Project Echo: Train Model ---

model = RandomForestClassifier(

n\_estimators=params["n\_estimators"],

max\_depth=params["max\_depth"],

random\_state=params["random\_state"],

criterion=params["criterion"]

)

model.fit(X\_train, y\_train)

print("Model training complete.")

# --- 6. Project Echo: Log Metrics ---

predictions = model.predict(X\_test)

acc = accuracy\_score(y\_test, predictions)

mlflow.log\_metric("accuracy", acc) # Log a single metric.

print(f"Logged metric: accuracy = {acc:.4f}")

# You can log metrics at different steps (e.g., epochs)

# for i in range(5):

# mlflow.log\_metric("loss\_epoch", np.random.rand(), step=i)

# --- 7. Project Echo: Log Artifacts (e.g., plots, data files) ---

# Example: Log a feature importance plot (if applicable)

# import matplotlib.pyplot as plt

# fig, ax = plt.subplots()

# ax.bar(range(len(model.feature\_importances\_)), model.feature\_importances\_)

# ax.set\_title("Feature Importances")

# mlflow.log\_figure(fig, "feature\_importances.png")

# Example: Log a text file

with open("echo\_model\_summary.txt", "w") as f:

f.write(f"Model: RandomForestClassifier\n")

f.write(f"Accuracy: {acc:.4f}\n")

f.write(f"Parameters: {params}\n")

mlflow.log\_artifact("echo\_model\_summary.txt", artifact\_path="run\_summary")

print("Logged artifact: echo\_model\_summary.txt")

# --- 8. Project Echo: Log and Register Model ---

# Log the trained model. MLflow saves it in a format that includes dependencies.

# `registered\_model\_name` will create a new registered model or a new version if it exists.

# Model signatures infer input/output schema.

input\_example = X\_train.sample(5) # A small sample of the input data

signature = mlflow.models.infer\_signature(X\_train, model.predict(X\_train)) # More robust signature

mlflow.sklearn.log\_model(

sk\_model=model,

artifact\_path="echo-random-forest-model", # Path within the run's artifacts

registered\_model\_name="EchoEngine\_RandomForest", # Name in Model Registry.

signature=signature,

input\_example=input\_example

)

print("Model logged and registered as 'EchoEngine\_RandomForest'.")

print("MLflow script finished.")

# --- 9. Project Echo: Load a Registered Model (Example for later use) ---

# This would typically be in a separate script for inference or serving

# model\_name = "EchoEngine\_RandomForest"

# model\_version = "1" # Or "Staging", "Production"

#

# try:

# loaded\_model = mlflow.pyfunc.load\_model(model\_uri=f"models:/{model\_name}/{model\_version}") #

# # loaded\_model = mlflow.sklearn.load\_model(model\_uri=f"models:/{model\_name}/{model\_version}") # For scikit-learn specific API

# print(f"\nSuccessfully loaded model '{model\_name}' version '{model\_version}'.")

# # Example prediction

# # sample\_prediction = loaded\_model.predict(X\_test.head(1))

# # print("Sample prediction:", sample\_prediction)

# except Exception as e:

# print(f"Error loading model: {e}")

**To run this script:**

python echo\_train\_mlflow.py

Then, refresh the MLflow UI (mlflow ui) to see the new experiment, run, parameters, metrics, artifacts, and registered model.

**3. Integration with Existing Workflows**

* **Training Scripts:** Wrap your existing training logic within with mlflow.start\_run():.
* **Parameter Logging:** Before training, log all relevant hyperparameters using mlflow.log\_param() or mlflow.log\_params().
* **Metric Logging:** After evaluation, log performance metrics using mlflow.log\_metric().
* **Model Logging:** Save your trained model using the appropriate mlflow.<flavor>.log\_model() function (e.g., mlflow.sklearn.log\_model, mlflow.tensorflow.log\_model). This bundles the model and its dependencies.
* **Model Registry:** Use the registered\_model\_name parameter in log\_model or mlflow.register\_model() to version models in the MLflow Model Registry. This allows you to manage model stages (e.g., Staging, Production).

**5. DVC (Data Version Control) Integration**

DVC is a version control system for machine learning projects, designed to handle large files, data sets, and ML models. It works alongside Git, using Git to store metadata and DVC to manage the actual data in a separate storage.

**1. Setup and Configuration**

**Installation:** Install DVC using pip. You might need to install support for specific remote storage types (e.g., s3, gcs, azure, ssh).

pip install dvc[all] # Installs DVC with all supported remote types

# Or for a specific remote, e.g., S3:

# pip install dvc[s3]

**Initialization (in your Git repository):** Navigate to your Project Echo Git repository root.

# 1. Ensure it's a Git repository

# git init (if not already)

# 2. Initialize DVC

dvc init

git commit -m "Initialize DVC" # This commits DVC configuration files like .dvc/config.

This creates a .dvc directory with configuration files and adds .dvc/cache and other DVC-specific files to .gitignore.

**Configure Remote Storage (Optional but Recommended):** DVC needs a place to store your large files. This can be local, or a cloud storage like S3, GCS, Azure Blob Storage, SSH, etc.

* **Local Remote (for testing/simplicity):** bash # Create a directory outside your project for DVC remote storage mkdir /tmp/dvc\_remote\_storage\_echo dvc remote add -d local\_echo\_remote /tmp/dvc\_remote\_storage\_echo # The -d flag sets this as the default remote. git add .dvc/config # DVC remote configuration is stored here git commit -m "Configure DVC local remote"
* **Cloud Remote (e.g., S3 - conceptual):** bash # dvc remote add my\_s3\_echo\_remote s3://your-echo-bucket/dvc-storage # You might need to configure credentials, e.g., using dvc remote modify or environment variables. # git add .dvc/config # git commit -m "Configure DVC S3 remote"

**2. Code Snippets and Basic Usage Patterns**

DVC is primarily command-line driven. Here's how you integrate it into your workflow:

**params.yaml (Example):**

train:

seed: 42

n\_estimators: 100

max\_depth: 5

output\_model\_path: models/echo\_model.pkl # DVC will track this output

output\_metrics\_path: metrics/train\_metrics.json # DVC will track this metric file

**echo\_train\_dvc.py (Conceptual - DVC itself doesn't require Python API for tracking):** This script would load parameters from params.yaml, load data, train a model, save the model to models/echo\_model.pkl, and save metrics to metrics/train\_metrics.json.

# Conceptual echo\_train\_dvc.py

import pickle

import yaml

import json

from sklearn.ensemble import RandomForestClassifier

import numpy as np

import os

# Create directories if they don't exist

os.makedirs("models", exist\_ok=True)

os.makedirs("metrics", exist\_ok=True)

# 1. Load parameters

with open("params.yaml", 'r') as f:

params = yaml.safe\_load(f)['train']

print(f"Parameters: {params}")

# 2. Load data (dummy - replace with actual data loading)

# Assume data/raw/features.csv and data/raw/labels.csv are tracked by DVC

# For this example, let's just create dummy data

X\_train = np.random.rand(100, params.get('n\_features', 5))

y\_train = np.random.randint(0, 2, 100)

print("Data loaded/generated.")

# 3. Train model

model = RandomForestClassifier(

n\_estimators=params['n\_estimators'],

max\_depth=params['max\_depth'],

random\_state=params['seed']

)

model.fit(X\_train, y\_train)

print("Model trained.")

# 4. Save model

with open(params['output\_model\_path'], 'wb') as f:

pickle.dump(model, f)

print(f"Model saved to {params['output\_model\_path']}")

# 5. Save metrics (dummy example)

metrics = {"accuracy": model.score(X\_train, y\_train)} # Replace with actual evaluation

with open(params['output\_metrics\_path'], 'w') as f:

json.dump(metrics, f, indent=4)

print(f"Metrics saved to {params['output\_metrics\_path']}")

**DVC Workflow Commands:**

1. **Add Data to DVC Tracking:** Let's say you have your raw dataset in data/raw/dataset.csv.
2. # Create dummy data for example
3. mkdir -p data/raw
4. echo "feature1,feature2,label" > data/raw/dataset.csv
5. for i in {1..100}; do echo "$RANDOM,$RANDOM,$(($RANDOM % 2))" >> data/raw/dataset.csv; done
6. dvc add data/raw/dataset.csv # This creates data/raw/dataset.csv.dvc.
7. git add data/raw/dataset.csv.dvc data/raw/.gitignore # Add the .dvc file (pointer) and .gitignore.
8. git commit -m "Add raw dataset v1"

The actual dataset.csv is now in .dvc/cache and .gitignore is updated to ignore the original file.

**Track an Experiment (using dvc exp run):** DVC experiments can track changes in code, data, parameters, and resulting metrics/models. Make sure params.yaml and echo\_train\_dvc.py are committed to Git.

1. # First, commit your initial training script and params.yaml
2. git add echo\_train\_dvc.py params.yaml
3. git commit -m "Add initial training script and parameters"
4. # Run an experiment
5. # DVC will execute `python echo\_train\_dvc.py`, detect changes in params.yaml,
6. # track outputs (models/echo\_model.pkl, metrics/train\_metrics.json) and metrics.
7. dvc exp run --set-param train.n\_estimators=100
8. # This command implicitly tracks outputs defined in params.yaml if they are created by the script.
9. # To be explicit, or if not using params.yaml for output paths, you'd use `dvc stage add` or track outputs manually.

To make DVC explicitly track outputs and metrics from the script as part of the experiment: You can define a dvc.yaml file or use DVCLive within your Python script for more direct experiment logging.

**Using DVCLive for metrics logging within echo\_train\_dvc.py:** Install DVCLive: pip install dvclive Modify echo\_train\_dvc.py:

# ... (imports and previous code) ...

from dvclive import Live # Add this import

# ... (parameter loading, data loading, model training) ...

# 5. Log metrics and save model using DVCLive

with Live(save\_dvc\_exp=True, cache\_images=False) as live: # save\_dvc\_exp=True integrates with dvc exp run

live.log\_params(params) # Log parameters from params.yaml

accuracy = model.score(X\_train, y\_train) # Replace with actual evaluation on a test set

live.log\_metric("accuracy", accuracy) # Log metrics.

print(f"Metrics logged with DVCLive: accuracy = {accuracy}")

# Log model as an artifact

model\_path = params['output\_model\_path']

with open(model\_path, 'wb') as f:

pickle.dump(model, f)

live.log\_artifact(model\_path, type="model", name="echo-model-dvclive")

print(f"Model saved to {model\_path} and logged as artifact by DVCLive.")

# You can also log the metrics.json file if you prefer manual JSON output

metrics\_data = {"accuracy": accuracy}

metrics\_file\_path = params['output\_metrics\_path']

with open(metrics\_file\_path, 'w') as f:

json.dump(metrics\_data, f, indent=4)

live.log\_artifact(metrics\_file\_path, type="metrics")

print(f"Metrics JSON saved to {metrics\_file\_path} and logged as artifact by DVCLive.")

Now, when you run dvc exp run, DVCLive will automatically log params, metrics, and artifacts.

**Versioning Models and Outputs:** After dvc exp run (especially with DVCLive or if outputs are defined in dvc.yaml), the outputs like models/echo\_model.pkl and metrics/train\_metrics.json are tracked as part of the experiment. If you're not using dvc exp run for everything and want to version a model (or any large file) manually:

1. # Assume echo\_train\_dvc.py created models/echo\_model.pkl
2. dvc add models/echo\_model.pkl # Track the model file.
3. git add models/echo\_model.pkl.dvc models/.gitignore # Commit the .dvc pointer file.
4. git commit -m "Version model v1.0 (RandomForest n\_est=100)"
5. **Pushing Data and Models to Remote Storage:**
6. dvc push # Pushes data/models tracked by DVC to the configured remote (e.g., local\_echo\_remote or S3).

And push your Git commits:

git push origin your-branch

**Reproducibility and Retrieving Versions:**

* **Switching between data/model versions:** bash git checkout <commit\_hash\_for\_desired\_version> # Switches .dvc files dvc checkout # Brings the corresponding data/model files from cache to your workspace. # Or, if files are not in local cache but in remote: # dvc pull
* **Reproducing an experiment from dvc.yaml (if you defined stages):** bash # dvc repro # This command re-runs stages if inputs/dependencies changed.
* **Managing experiments:** bash dvc exp show # Shows a table of all experiments, their params, and metrics. dvc exp diff <exp\_a> <exp\_b> # Compare two experiments dvc exp apply <experiment\_name> # Apply the changes from an experiment to your workspace

**Integration with Existing Workflows in the Repository**

* **Git Workflow:** DVC is designed to complement Git. You commit .dvc files (small metadata files) and dvc.yaml (pipeline definitions) to Git, while DVC manages the large actual files.
* **Data Preprocessing:** If you have data preprocessing scripts, define them as stages in dvc.yaml so DVC can track their inputs (raw data) and outputs (processed data).
* **Training:** Define your training script as a stage in dvc.yaml, with dependencies on processed data and parameters (from params.yaml), and outputs like model files and metrics files.
* **Versioning:** Use dvc add for individual large files/directories or dvc exp run for a more holistic experiment tracking that includes versioning of code, data, params, and outputs.
* **Collaboration:** Team members can git pull to get the latest .dvc files/dvc.yaml and then dvc pull to download the actual data/model files from the DVC remote.

**Example dvc.yaml (for pipeline definition):**

stages:

preprocess\_data: # Example stage

cmd: python scripts/preprocess.py --input data/raw/dataset.csv --output data/processed/features.pkl

deps:

- scripts/preprocess.py

- data/raw/dataset.csv

outs:

- data/processed/features.pkl

train\_model:

cmd: python echo\_train\_dvc.py # Assumes echo\_train\_dvc.py reads params.yaml

deps:

- echo\_train\_dvc.py

- data/processed/features.pkl # Output from preprocess\_data

- params.yaml

params: # Parameters from params.yaml to track

- train.seed

- train.n\_estimators

- train.max\_depth

metrics: # DVC will look for these files for metrics

- metrics/train\_metrics.json: # Path to metrics file

cache: false # Usually false for metrics files (small, text-based)

plots: # DVC can also render plots

- metrics/accuracy\_plot.png: # Example plot output by training script

cache: false

outs: # Outputs of the training script

- models/echo\_model.pkl # The trained model

To run this pipeline: dvc repro train\_model

**References**

MLflow.org, n.d. *Concepts - MLflow*. Available from: <https://mlflow.org/docs/1.23.1/concepts.html> [9 April 2025].

DVC.org, n.d. *Model Registry | Data Version Control*. Available from: <https://dvc.org/doc/use-cases/model-registry> [7 April 2025].

GuildAI.github.io, n.d. *Introduction to Guild AI for R*. Available from: <https://guildai.github.io/guildai-r/articles/guildai.html> [7 April 2025].

Metaflow.org, n.d. *Checkpointing Progress - Metaflow Docs*. Available from: <https://docs.metaflow.org/scaling/checkpointing> [7 April 2025].

CanvasBusinessModel.com, n.d. *How Does Weights & Biases Work?*. Available from: <https://canvasbusinessmodel.com/blogs/how-it-works/how-does-weights-and-biases-work> [7 April 2025].

Neptune.ai, n.d. *ML Model Registry: The Ultimate Guide*. Available from: <https://neptune.ai/blog/ml-model-registry> [5 April 2025].

MicrosoftLearn, n.d. *Register and work with models - Azure Machine Learning*. Available from: <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-manage-models> [5 April 2025].

DVC, n.d. *DVC Get Started*. Available from: [https://dvc.org/doc/start](https://www.google.com/url?sa=E&q=https%3A%2F%2Fdvc.org%2Fdoc%2Fstart) [2 May 2025].

DVC, n.d. *Data Versioning - DVC*. Available from: [https://dvc.org/doc/use-cases/data-versioning](https://www.google.com/url?sa=E&q=https%3A%2F%2Fdvc.org%2Fdoc%2Fuse-cases%2Fdata-versioning) [2 May 2025].

DVC, n.d. *Command Reference - dvc remote*. Available from: [https://dvc.org/doc/command-reference/remote](https://www.google.com/url?sa=E&q=https%3A%2F%2Fdvc.org%2Fdoc%2Fcommand-reference%2Fremote) [2 May 2025].

DVC, n.d. *Experiments - DVC*. Available from: [https://dvc.org/doc/user-guide/experiment-management](https://www.google.com/url?sa=E&q=https%3A%2F%2Fdvc.org%2Fdoc%2Fuser-guide%2Fexperiment-management) [2 May 2025].

DVC, n.d. *DVCLive*. Available from: [https://dvc.org/doc/dvclive](https://www.google.com/url?sa=E&q=https%3A%2F%2Fdvc.org%2Fdoc%2Fdvclive) [2 May 2025].

DVC, n.d. *Pipelines - DVC*. Available from: [https://dvc.org/doc/user-guide/pipelines](https://www.google.com/url?sa=E&q=https%3A%2F%2Fdvc.org%2Fdoc%2Fuser-guide%2Fpipelines) [2 May 2025].

MLflow, n.d. *MLflow Tracking*. Available from: [https://mlflow.org/docs/latest/tracking.html](https://www.google.com/url?sa=E&q=https%3A%2F%2Fmlflow.org%2Fdocs%2Flatest%2Ftracking.html) [2 May 2025].

MLflow, n.d. *MLflow Models*. Available from: [https://mlflow.org/docs/latest/models.html](https://www.google.com/url?sa=E&q=https%3A%2F%2Fmlflow.org%2Fdocs%2Flatest%2Fmodels.html) [2 May 2025].

MLflow, n.d. *MLflow Model Registry*. Available from: [https://mlflow.org/docs/latest/model-registry.html](https://www.google.com/url?sa=E&q=https%3A%2F%2Fmlflow.org%2Fdocs%2Flatest%2Fmodel-registry.html) [2 May 2025].

MLflow, n.d. *MLflow Pyfunc Model Flavor*. Available from: [https://mlflow.org/docs/latest/python\_api/mlflow.pyfunc.html#mlflow.pyfunc.load\_model](https://www.google.com/url?sa=E&q=https%3A%2F%2Fmlflow.org%2Fdocs%2Flatest%2Fpython_api%2Fmlflow.pyfunc.html%23mlflow.pyfunc.load_model) [2 May 2025].