# Comprehensive MLOps Setup: DVC, MLflow, and DagsHub

This guide details the integration of DVC (Data Version Control) and MLflow (Experiment Tracking) using DagsHub as the centralized remote storage and tracking server. This setup is mandatory for reproducibility and auditing your ML pipeline.

## 1. Prerequisites and Initial Setup

### A. Install Necessary Tools

Ensure the following packages are installed in your development environment.

pip install dvc dvc-s3 # DVC core + S3 support (DagsHub uses S3 compatible storage)

pip install mlflow

pip install dagshub

### B. Configure DagsHub (The MLOps Remote)

Assuming you have created a DagsHub repository mirroring your GitHub project:

1. Authorize DagsHub: Run this command and follow the prompts. This authenticates your local machine for pushing metrics/data.

dagshub login

1. Initialize DVC: This creates the basic DVC structure (.dvc/config).

dvc init

dvc add <raw\_data\_file\_path>

Then

git add <raw\_data\_file\_path>.dvc

git add <raw\_data\_file\_path>.gitignore

git commit -m “Added raw data”

Prepare dvc.yaml with all components

Once done then execute the below cmd to run entire pipeline

dev repro

Add the Dagshub DVC remote:-> to push the dvc trace files to s3

dvc remote add origin s3://dvc

dvc remote modify origin endpointurl <URL link check in dagshub>

Then add Setup Credentials

dvc remote modify origin --local access\_key\_id <access\_key>

dvc remote modify origin --local secret\_access\_key <secret\_access\_key>

Then

dvc pull -r origin

dvc push -r origin

Last step:

git add .

git commit -m “final changes”

git push origin main

1. Configure DVC Remote: Set DagsHub as the default DVC remote storage. Replace user/repo with your actual DagsHub username and repository name.

dvc remote add origin [https://dagshub.com/](https://dagshub.com/)<user>/<repo>.dvc

dvc remote default origin

1. Configure MLflow Tracking: Set the environment variable so MLflow sends all run metadata to DagsHub.

export MLFLOW\_TRACKING\_URI=[https://dagshub.com/](https://dagshub.com/)<user>/<repo>.mlflow

## 2. DVC Integration (Data and Model Versioning)

DVC will track the raw datasets (output of Data Ingestion) and the final trained model (output of Model Trainer/Pusher).

### Stage: Data Ingestion

The goal is to version the raw data files (e.g., train.csv, test.csv) so that subsequent components always use the same snapshot.

|  |  |  |
| --- | --- | --- |
| **Step** | **Command/Action** | **Location** |
| 1. Track Data | Use dvc add on the output folder of Data Ingestion. | After data\_ingestion.py runs |
| 2. Commit Metadata | Commit the generated .dvc files to Git. | Git Commit |
| 3. Push Data | Push the actual data files to the DagsHub remote storage. | Terminal |

Terminal Commands:

# Example: Versioning the data in the artifacts/data\_ingestion/ingested directory

dvc add artifacts/data\_ingestion/ingested

git add artifacts/data\_ingestion/ingested.dvc

git commit -m "DVC: Tracked ingested data files"

# Push the data to DagsHub storage

dvc push

## 3. MLflow Integration (Experiment Tracking)

MLflow is primarily used in the Model Trainer and Model Evaluation components to record parameters, metrics, and models.

### Stage: Model Trainer

Modify your model\_trainer.py component to wrap the training logic within an MLflow run.

|  |  |
| --- | --- |
| **Action** | **MLflow Code Implementation (Example)** |
| 1. Start Run | import mlflow; with mlflow.start\_run(run\_name="Training\_Run") as run: |
| 2. Log Parameters | mlflow.log\_param("model\_name", "RandomForest"); mlflow.log\_param("n\_estimators", 100) |
| 3. Log Metrics | Log the calculated metrics using the artifact data. |
| 4. Log Model | Save the model directly to the MLflow registry. |

Code Snippet for model\_trainer.py:

import mlflow

import mlflow.sklearn # or mlflow.pyfunc

from sklearn.ensemble import RandomForestClassifier

# ... other imports

# Inside the ModelTrainer class's initiate\_model\_trainer method:

with mlflow.start\_run(run\_name="RandomForest\_Training\_v1") as run:

# 1. Log Hyperparameters

mlflow.log\_param("n\_estimators", 100)

mlflow.log\_param("max\_depth", 5)

# 2. Train the model (simplified)

model = RandomForestClassifier(n\_estimators=100, max\_depth=5)

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

# 3. Log Metrics (Use the data from your ClassificationMetricArtifact)

train\_f1 = self.model\_trainer\_artifact.train\_metric\_arifact.f1\_score

test\_f1 = self.model\_trainer\_artifact.test\_metric\_arifact.f1\_score

mlflow.log\_metric("train\_f1\_score", train\_f1)

mlflow.log\_metric("test\_f1\_score", test\_f1)

# 4. Log the Trained Model to MLflow

mlflow.sklearn.log\_model(

sk\_model=model,

artifact\_path="model",

registered\_model\_name="USVisaClassifier"

)

# 5. DVC track the model (for file-based deployment/versioning)

dvc\_model\_path = self.model\_trainer\_artifact.trained\_model\_file\_path

# This command should run \*after\* the python script finishes:

# Use dvc commit to record the new state of the file produced by the stage

dvc commit artifacts\model\_trainer\trained\_model\model.pkl

# git add <dvc\_model\_path>.dvc

# git commit -m "Trained model V1"

### Stage: Model Evaluation

Use MLflow to log the current performance and, critically, the comparison against the production model.

Code Snippet for model\_evaluation.py:

# Inside the ModelEvaluation class's initiate\_model\_evaluation method:

with mlflow.start\_run(run\_name="Model\_Evaluation", nested=True): # Use nested run

# Use the result from self.evaluate\_model()

eval\_response = self.evaluate\_model()

mlflow.log\_metric("current\_model\_test\_f1", eval\_response.test\_model\_f1\_score)

if eval\_response.best\_model\_f1\_score is not None:

mlflow.log\_metric("production\_model\_f1", eval\_response.best\_model\_f1\_score)

mlflow.log\_metric("f1\_difference", eval\_response.difference)

mlflow.log\_param("is\_model\_accepted", eval\_response.is\_model\_accepted)

## 4. DVC/MLflow Stage (Model Pusher)

The Model Pusher component handles promoting the model if the evaluation passes. This involves marking the model as "production" in both DVC and the MLflow Registry.

### Stage: Model Pusher

|  |  |  |
| --- | --- | --- |
| **Tool** | **Action** | **Implementation** |
| MLflow Registry | Promote the accepted model version to the Production stage. | Use mlflow.tracking.MlflowClient to update the model stage. |
| DVC | Tag the accepted model file for easy rollback/retrieval. | Use dvc tag and git tag. |

1. MLflow Model Promotion (Code inside model\_pusher.py):

from mlflow.tracking import MlflowClient

# Get the latest version of the model from the registry

client = MlflowClient()

latest\_version = client.get\_latest\_versions("USVisaClassifier")[0] # Assuming latest is the one being evaluated

if is\_model\_accepted:

client.transition\_model\_version\_stage(

name="USVisaClassifier",

version=latest\_version.version,

stage="Production"

)

# Log the promotion event

print(f"Model version {latest\_version.version} promoted to Production.")

2. DVC Production Tagging (Terminal Command):

If the model is accepted, you should tag the DVC-tracked model file's state in Git. This creates an easy, persistent reference point for deploying that specific model version.

# After running the pipeline and if is\_model\_accepted is True:

git tag -a "prod-model-v$(date +%Y%m%d)" -m "Accepted model promoted to production"

git push --tags

dvc push -r origin # Push the latest DVC cache