MLflow Logging

Complete Study Guide

Master MLflow's Tracking Components

📑 Table of Contents

🧭 1. Parameters

Definition: Parameters are the inputs or hyperparameters used for your model or experiment. They define the configuration settings that control how your model learns.

What Are Parameters?

Parameters represent the configuration choices you make before and during training. These include:

- **Learning Rate:** Controls how quickly the model adapts
- **Batch Size:** Number of samples processed together
- **Number of Layers:** Architecture decisions
- **Regularization:** Penalty terms to prevent overfitting
- **Optimizer Type:** Algorithm used for optimization (Adam, SGD, etc.)

Examples:

```
mlflow.log_param("learning_rate", 0.01)
mlflow.log_param("batch_size", 32)
mlflow.log_param("num_layers", 3)
mlflow.log_param("optimizer", "adam")
mlflow.log_param("dropout_rate", 0.2)
```

✅ Useful for comparing different runs or configurations to find optimal hyperparameters

✅ Enables systematic hyperparameter tuning and experiment tracking

📊 2. Metrics

Definition: Metrics are numerical performance values that can change during training. They measure how well your model is performing on various aspects.

Types of Metrics

- **Accuracy:** Percentage of correct predictions
- **Loss:** Error measure during training
- **Precision & Recall:** Classification quality measures
- **F1 Score:** Harmonic mean of precision and recall
- **AUC-ROC:** Area under the receiver operating characteristic curve

Basic Metrics Logging:

```
mlflow.log_metric("accuracy", 0.95)
mlflow.log_metric("loss", 0.24)
mlflow.log_metric("f1_score", 0.91)
mlflow.log_metric("precision", 0.93)
mlflow.log_metric("recall", 0.89)
```

Tracking Metrics Over Time (Per Epoch):

```
for epoch in range(10):
    train_loss = compute_loss(model, train_data)
    val_loss = compute_loss(model, val_data)

    mlflow.log_metric("train_loss", train_loss, step=epoch)
    mlflow.log_metric("val_loss", val_loss, step=epoch)
    mlflow.log_metric("accuracy", accuracy, step=epoch)
```

> ✅ Useful for plotting trends and analyzing model performance visually in the MLflow UI

> ✅ Enables early stopping decisions based on validation metrics

> ✅ Helps identify overfitting by comparing training vs validation metrics

📁 3. Artifacts

Definition: Artifacts are output files generated during training or evaluation. They include any file or directory you want to save alongside your experiment.

Common Artifact Types

- **Model Files:** .pkl, .pt, .h5, .onnx
- **Visualizations:** Confusion matrix, ROC curves, training plots
- **Feature Importance:** Graphs showing feature contributions
- **Data Files:** Test predictions, processed datasets
- **Configuration Files:** JSON/YAML configs
- **Logs:** Training logs, error reports
- **Notebooks:** Analysis notebooks

Logging Individual Files:

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDis

# Create and save a plot
cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(cm)
disp.plot()
plt.savefig("confusion_matrix.png")

# Log the artifact
```

```
mlflow.log_artifact("confusion_matrix.png")
mlflow.log_artifact("feature_importance.png")
mlflow.log_artifact("training_history.json")
```

Logging Entire Directories:

```
# Log all files in a directory
mlflow.log_artifacts("models/")
mlflow.log_artifacts("plots/")
mlflow.log_artifacts("outputs/")
```

✅ Keeps all outputs from a run (plots, checkpoints, reports) in one centralized place

✅ Makes experiment results reproducible and shareable

✅ Enables detailed post-training analysis and reporting

🧠 4. Models

Definition: You can log entire models with their metadata, which allows for easy loading, serving, and deployment. MLflow provides framework-specific model logging.

Why Log Models?

- Packages model with dependencies and environment info
- Enables easy model loading and inference

- Supports model versioning and registry
- Facilitates deployment to various platforms
- Includes model signature (input/output schema)

Logging Different Framework Models:

```
# Scikit-learn model
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(X_train, y_train)
mlflow.sklearn.log_model(model, "random_forest_model")

# PyTorch model
import torch
pytorch_model = MyNeuralNetwork()
mlflow.pytorch.log_model(pytorch_model, "pytorch-model")

# TensorFlow/Keras model
import tensorflow as tf
keras_model = tf.keras.models.Sequential([...])
mlflow.tensorflow.log_model(keras_model, "tf-model")

# XGBoost model
import xgboost as xgb
xgb_model = xgb.XGBClassifier()
mlflow.xgboost.log_model(xgb_model, "xgboost-model")
```

Loading a Logged Model:

```
# Load model for inference
model_uri = "runs:/<run_id>/model"
loaded_model = mlflow.sklearn.load_model(model_uri)

# Make predictions
predictions = loaded_model.predict(new_data)
```

✅ MLflow supports model registry for versioning and stage transitions (Staging, Production)

✅ Models can be deployed directly from MLflow to various platforms

✅ Includes all dependencies needed for model inference

🧾 5. Tags

Definition: Tags are metadata key-value pairs used for organizing, searching, and filtering experiments and runs.

Common Use Cases for Tags

- **Author/Team:** Who created the experiment
- **Purpose:** Experiment objective (baseline, tuning, production)
- **Dataset Version:** Which data version was used
- **Environment:** Development, staging, production
- **Model Type:** Architecture or algorithm used
- **Project:** Associated project or initiative

Setting Tags:

```
mlflow.set_tag("author", "Ravan")
mlflow.set_tag("purpose", "baseline model")
mlflow.set_tag("dataset_version", "v2.1")
mlflow.set_tag("model_type", "CNN")
mlflow.set_tag("environment", "production")
```

```
mlflow.set_tag("project", "customer_churn")
mlflow.set_tag("priority", "high")
```

✅ Very handy for filtering runs in the MLflow UI

✅ Enables quick identification of experiment context and purpose

✅ Helps teams collaborate by adding organizational metadata

6. Source Information

Definition: MLflow automatically logs source code information and environment details to ensure reproducibility.

Automatically Logged Information

- **Script Name:** The Python file that was executed
- **Git Commit Hash:** If in a Git repository, the commit ID
- **Git Branch:** Current branch name
- **Git Remote URL:** Repository location
- **Python Version:** Version of Python used
- **Dependencies:** Library versions (from conda.yaml or requirements.txt)
- **Start Time:** When the run began
- **End Time:** When the run completed
- **User:** Who executed the run

Example of Logged Source Info:

```
# Automatically logged (no code needed)
{
    "source_type": "LOCAL",
    "source_name": "train_model.py",
    "git_commit": "a3f2b1c",
    "git_branch": "feature/new-model",
    "python_version": "3.9.7",
    "start_time": "2025-10-23 10:30:00",
```

```
    "end_time": "2025-10-23 11:15:00",
    "user": "ravan"
}
```

✅ Helps you reproduce results later by knowing exact code version and environment

✅ Tracks code changes over time automatically

✅ Ensures experiments can be recreated with the same dependencies

⚙️ 7. Automatic Logging (Autologging)

Definition: MLflow can automatically log parameters, metrics, models, and artifacts for many popular ML frameworks without manual logging code.

Supported Frameworks

- Scikit-learn
- TensorFlow / Keras
- PyTorch
- XGBoost
- LightGBM
- Spark MLlib
- Fastai

- Statsmodels

Enabling Autologging:

```
# Scikit-learn autologging
mlflow.sklearn.autolog()
model = RandomForestClassifier()
model.fit(X_train, y_train)  # Automatically logged!

# PyTorch autologging
mlflow.pytorch.autolog()
# Your PyTorch training code
# Parameters, metrics, and model automatically logged!

# TensorFlow/Keras autologging
mlflow.tensorflow.autolog()
model.fit(X_train, y_train, epochs=10)  # Auto-logged!

# Universal autologging (enables for all supported frameworks)
mlflow.autolog()

# XGBoost autologging
mlflow.xgboost.autolog()
xgb_model.fit(X_train, y_train)
```

What Gets Automatically Logged?

- **Parameters:** All model hyperparameters
- **Metrics:** Training and validation metrics per epoch
- **Models:** Trained model with signature
- **Artifacts:** Plots like training curves, feature importance
- **Model Summary:** Architecture details for deep learning models

- ✅ Saves significant time by eliminating manual logging code
- ✅ Ensures consistency across runs by logging standard metrics
- ✅ Reduces chance of forgetting to log important information

Complete Autologging Example:

```
import mlflow
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Enable autologging
mlflow.sklearn.autolog()

# Your normal ML code
X_train, X_test, y_train, y_test = train_test_split(X, y)

with mlflow.start_run():
    # Train model - everything is auto-logged!
    model = RandomForestClassifier(n_estimators=100, max_depth=5
    model.fit(X_train, y_train)

    # Even predictions and scores are logged
    score = model.score(X_test, y_test)

# Check MLflow UI to see all logged information!
```

🧰 8. Summary Reference Table

Category	Description	Example Code
Parameters	Inputs/hyperparameters that configure your model	`mlflow.log_param("lr", 0.001)`
Metrics	Numerical performance results	`mlflow.log_metric("accuracy", 0.93)`
Artifacts	Output files and directories	`mlflow.log_artifact("plot.png")`
Models	Trained models with metadata	`mlflow.sklearn.log_model(model, "model")`
Tags	Custom metadata for organization	`mlflow.set_tag("stage", "testing")`
Source Info	Code version and environment	*Automatically logged*
Autologging	Auto-logs params, metrics, models	`mlflow.sklearn.autolog()`

💡 Best Practices

1. Start Your Runs Explicitly

```
with mlflow.start_run(run_name="baseline_model"):
    # Your logging code here
    mlflow.log_param("learning_rate", 0.01)
    # ... train model ...
    mlflow.log_metric("accuracy", accuracy)
```

2. Organize with Experiments

```
# Create or set experiment
mlflow.set_experiment("customer_churn_prediction")

# All subsequent runs go to this experiment
with mlflow.start_run():
    # ... your code ...
```

3. Use Nested Runs for Complex Workflows

```
with mlflow.start_run(run_name="hyperparameter_tuning"):
    for params in param_grid:
        with mlflow.start_run(nested=True):
            model = train_model(**params)
            mlflow.log_params(params)
            mlflow.log_metric("accuracy", score)
```

4. Log Early and Often

- Log parameters before training starts
- Log metrics during training (per epoch/batch)
- Log final metrics and artifacts after training
- Use consistent naming conventions

5. Combine Manual and Automatic Logging

```
mlflow.sklearn.autolog()  # Enable autologging

with mlflow.start_run():
    model = RandomForestClassifier()
    model.fit(X_train, y_train)  # Auto-logged

    # Add custom logs
    mlflow.set_tag("experiment_type", "baseline")
    mlflow.log_artifact("custom_analysis.html")
```

🎯 Quick Reference Commands

```
# Initialize MLflow
import mlflow

# Set tracking URI (optional)
mlflow.set_tracking_uri("http://localhost:5000")

# Set experiment
```

```
mlflow.set_experiment("my_experiment")

# Start a run
with mlflow.start_run(run_name="my_run"):

    # Log single parameter
    mlflow.log_param("param_name", value)

    # Log multiple parameters
    mlflow.log_params({"param1": val1, "param2": val2})

    # Log single metric
    mlflow.log_metric("metric_name", value)

    # Log multiple metrics
    mlflow.log_metrics({"metric1": val1, "metric2": val2})

    # Log metric with step (for time series)
    mlflow.log_metric("loss", loss_value, step=epoch)

    # Log artifact (file)
    mlflow.log_artifact("path/to/file.png")

    # Log artifacts (directory)
    mlflow.log_artifacts("path/to/directory")

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

    # Set tag
    mlflow.set_tag("tag_name", "tag_value")

    # Set multiple tags
    mlflow.set_tags({"tag1": "val1", "tag2": "val2"})

# Enable autologging
mlflow.autolog()  # or framework-specific
mlflow.sklearn.autolog()
mlflow.tensorflow.autolog()
mlflow.pytorch.autolog()
```