Documentation MLflow - Tracking et Model Registry

Table des matières

Introduction à MLflow

MLflow est une plateforme open-source pour gérer le cycle de vie complet du machine learning. Elle comprend quatre composants principaux :

- **MLflow Tracking** : Enregistrement et interrogation des expériences
- **MLflow Projects** : Packaging du code ML pour la reproductibilité
- **MLflow Models** : Gestion et déploiement de modèles
- **MLflow Model Registry** : Stockage centralisé des modèles avec versioning

Cette documentation se concentre sur **MLflow Tracking** et **MLflow Model Registry**.

MLflow Tracking

Concepts clés

Run

Une exécution unique d'un code de machine learning. Chaque run enregistre :

- **Paramètres** : Valeurs d'entrée (hyperparamètres, configurations)
- **Métriques** : Valeurs numériques évaluées (accuracy, loss, etc.)
- **Artefacts** : Fichiers de sortie (modèles, graphiques, datasets)
- **Métadonnées** : Informations sur le run (timestamp, utilisateur, etc.)

Experiment

Regroupement logique de runs pour une tâche spécifique. Permet d'organiser et de comparer différentes exécutions.

Tracking Server

Serveur central qui stocke les données des runs. Peut être local ou distant.

Configuration

Installation

```
pip install mlflow
```

Démarrage du Tracking Server

Mode local :

```
mlflow ui
# Par défaut : http://localhost:5000
```

Mode serveur avec backend :

```
mlflow server \
    --backend-store-uri sqlite:///mlflow.db \
    --default-artifact-root ./mlruns \
    --host 0.0.0.0 \
    --port 5000
```

Avec PostgreSQL et S3 :

```
mlflow server \
    --backend-store-uri postgresql://user:password@localhost/mlflow \
    --default-artifact-root s3://my-bucket/mlflow \
    --host 0.0.0.0
```

Configuration du client

```
import mlflow

# Définir l'URI du tracking server
mlflow.set_tracking_uri("http://localhost:5000")
```

```
# Ou utiliser une variable d'environnement
# export MLFLOW_TRACKING_URI=http://localhost:5000
```

Logging des expériences

Structure de base

```
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Définir l'expérience
mlflow.set_experiment("mon_projet_classification")

# Démarrer un run
with mlflow.start_run(run_name="random_forest_v1"):
    # Paramètres du modèle
    n_estimators = 100
    max_depth = 10

    # Logger les paramètres
    mlflow.log_param("n_estimators", n_estimators)
    mlflow.log_param("max_depth", max_depth)

    # Entraîner le modèle
    model = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth
    )
    model.fit(X_train, y_train)

    # Faire des prédictions
    predictions = model.predict(X_test)
    accuracy = accuracy_score(y_test, predictions)

    # Logger les métriques
    mlflow.log_metric("accuracy", accuracy)
    mlflow.log_metric("test_samples", len(X_test))

    # Logger le modèle
    mlflow.sklearn.log_model(model, "model")

    print(f"Run ID: {mlflow.active_run().info.run_id}")
```

Logging de multiples paramètres et métriques

```
```

```
# Logger plusieurs paramètres à la fois
params = {
    "learning_rate": 0.01,
    "batch_size": 32,
    "epochs": 100,
    "optimizer": "adam"
}
mlflow.log_params(params)

# Logger plusieurs métriques à la fois
metrics = {
    "train_loss": 0.234,
    "val_loss": 0.456,
    "train_acc": 0.89,
    "val_acc": 0.85
}
mlflow.log_metrics(metrics)
```

Logging de métriques par étapes

```
# Pour le suivi pendant l'entraînement
for epoch in range(num_epochs):
    train_loss = train_one_epoch()
    val_loss = validate()

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

Logging d'artefacts

```
import matplotlib.pyplot as plt

# Créer et sauvegarder une figure
plt.figure(figsize=(10, 6))
plt.plot(history['loss'], label='Training Loss')
plt.plot(history['val_loss'], label='Validation Loss')
plt.legend()
plt.savefig("loss_plot.png")
plt.close()

# Logger l'image
mlflow.log_artifact("loss_plot.png")

# Logger un fichier texte
with open("notes.txt", "w") as f:
    f.write("Modèle entraîné avec succès")
mlflow.log_artifact("notes.txt")
```

```
# Logger un dossier complet
mlflow.log_artifacts("output_dir/")
```

Logging avec autolog

MLflow offre l'autolog pour plusieurs frameworks :

```
# Pour scikit-learn
mlflow.sklearn.autolog()

# Pour TensorFlow/Keras
mlflow.tensorflow.autolog()

# Pour PyTorch
mlflow.pytorch.autolog()

# Pour XGBoost
mlflow.xgboost.autolog()

# Puis votre code d'entraînement normal
# Les paramètres, métriques et modèles seront loggés automatiquement
```

Tags personnalisés

```
mlflow.set_tag("team", "data-science")
mlflow.set_tag("project", "customer-churn")
mlflow.set_tag("environment", "production")
```

Visualisation

Interface Web

L'interface MLflow UI permet de :

- Comparer plusieurs runs
- Visualiser les métriques en graphiques
- Télécharger les artefacts
- Rechercher et filtrer les runs

Accès : `http://localhost:5000` après avoir lancé `mlflow ui`

Recherche programmatique

```
from mlflow.tracking import MlflowClient
```

```
client = MlflowClient()

# Rechercher des runs dans une expérience
experiment = client.get_experiment_by_name("mon_projet_classification")
runs = client.search_runs(
    experiment_ids=[experiment.experiment_id],
    filter_string="metrics.accuracy > 0.8",
    order_by=["metrics.accuracy DESC"],
    max_results=5
)

for run in runs:
    print(f"Run ID: {run.info.run_id}")
    print(f"Accuracy: {run.data.metrics['accuracy']}")
    print(f"Parameters: {run.data.params}")
```

MLflow Model Registry

Concepts du Registry

Le Model Registry est un référentiel centralisé pour :

- Stocker et organiser les modèles ML
- Versionner les modèles
- Gérer le cycle de vie des modèles (staging, production)
- Collaborer entre équipes

Stages du modèle

- **None** : Modèle nouvellement enregistré
- **Staging** : Modèle en test/validation
- **Production** : Modèle déployé en production
- **Archived** : Modèle archivé/obsolète

Enregistrement de modèles

Méthode 1 : Depuis un run actif

```
with mlflow.start_run():
    # Entraîner votre modèle
    model = train_model()

    # Logger et enregistrer le modèle
    mlflow.sklearn.log_model(
        sk_model=model,
        artifact_path="model",
        registered_model_name="mon_modele_rf"
    )
```

Méthode 2 : Depuis un run existant

```
# Enregistrer un modèle depuis un run spécifique
model_uri = f"runs:/{run_id}/model"
mlflow.register_model(
    model_uri=model_uri,
    name="mon_modele_rf"
)
```

Méthode 3 : Via le client

```
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Créer une nouvelle version
result = client.create_model_version(
    name="mon_modele_rf",
    source=f"runs:/{run_id}/model",
    run_id=run_id
)

print(f"Version créée : {result.version}")
```

Gestion des versions

Lister les versions

```
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Obtenir toutes les versions d'un modèle
versions = client.search_model_versions(f"name='mon_modele_rf'")

for version in versions:
    print(f"Version {version.version}: Stage = {version.current_stage}")
```

Obtenir une version spécifique

```
# Charger la dernière version en production
model = mlflow.pyfunc.load_model(
    model_uri="models:/mon_modele_rf/Production"
```

```
)

# Charger une version spécifique
model = mlflow.pyfunc.load_model(
    model_uri="models:/mon_modele_rf/3"
)

# Utiliser le modèle
predictions = model.predict(X_test)
```

Ajouter des descriptions

```
# Description du modèle
client.update_registered_model(
    name="mon_modele_rf",
    description="Modèle Random Forest pour la classification des clients"
)

# Description d'une version
client.update_model_version(
    name="mon_modele_rf",
    version=1,
    description="Première version avec 100 estimators et max_depth=10"
)
```

Transitions de stages

Promouvoir un modèle

```
from mlflow.tracking import MlflowClient

client = MlflowClient()

# Passer en Staging
client.transition_model_version_stage(
    name="mon_modele_rf",
    version=3,
    stage="Staging"
)

# Passer en Production
client.transition_model_version_stage(
    name="mon_modele_rf",
    version=3,
    stage="Production",
    archive_existing_versions=True  # Archive les anciennes versions en
prod
)
```

```
# Archiver un modèle
client.transition_model_version_stage(
    name="mon_modele_rf",
    version=1,
    stage="Archived"
)
```

Workflow complet de déploiement

```
from mlflow.tracking import MlflowClient
import mlflow

client = MlflowClient()

# 1. Entraîner et enregistrer un nouveau modèle
with mlflow.start_run() as run:
    model = train_model()
    mlflow.sklearn.log_model(
        model,
        "model",
        registered_model_name="mon_modele_rf"
    )
    run_id = run.info.run_id

# 2. Obtenir la version créée
versions = client.search_model_versions(f"run_id='{run_id}'")
new_version = versions[0].version

# 3. Ajouter une description
client.update_model_version(
    name="mon_modele_rf",
    version=new_version,
    description=f"Modèle entraîné le {datetime.now()}"
)

# 4. Tester en Staging
client.transition_model_version_stage(
    name="mon_modele_rf",
    version=new_version,
    stage="Staging"
)

# 5. Valider le modèle
staging_model = mlflow.pyfunc.load_model(f"models:/mon_modele_rf/Staging")
test_accuracy = validate_model(staging_model, X_test, y_test)

# 6. Si validé, passer en Production
if test_accuracy > 0.85:
    client.transition_model_version_stage(
        name="mon_modele_rf",
```

```
        version=new_version,
        stage="Production",
        archive_existing_versions=True
    )
    print(f"Version {new_version} déployée en production")
else:
    print(f"Version {new_version} rejetée (accuracy: {test_accuracy})")
```

Ajouter des annotations

```
# Ajouter des tags à une version
client.set_model_version_tag(
    name="mon_modele_rf",
    version=3,
    key="validation_status",
    value="approved"
)

client.set_model_version_tag(
    name="mon_modele_rf",
    version=3,
    key="reviewer",
    value="john.doe@company.com"
)
```

Exemples pratiques

Exemple 1 : Pipeline complet scikit-learn

```
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
import pandas as pd

# Configuration
mlflow.set_tracking_uri("http://localhost:5000")
mlflow.set_experiment("customer_churn_prediction")

# Charger les données
df = pd.read_csv("customer_data.csv")
X = df.drop('churn', axis=1)
y = df['churn']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

```python
# Hyperparamètres à tester
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [10, 20, 30],
    'min_samples_split': [2, 5, 10]
}

# Activer l'autolog
mlflow.sklearn.autolog()

# Recherche d'hyperparamètres
with mlflow.start_run(run_name="rf_grid_search"):
    # Tags
    mlflow.set_tag("model_type", "RandomForest")
    mlflow.set_tag("dataset_size", len(df))

    # Grid Search
    rf = RandomForestClassifier(random_state=42)
    grid_search = GridSearchCV(
        rf,
        param_grid,
        cv=5,
        scoring='accuracy',
        n_jobs=-1
    )
    grid_search.fit(X_train, y_train)

    # Meilleur modèle
    best_model = grid_search.best_estimator_

    # Prédictions
    y_pred = best_model.predict(X_test)

    # Métriques supplémentaires
    metrics = {
        "test_accuracy": accuracy_score(y_test, y_pred),
        "test_precision": precision_score(y_test, y_pred),
        "test_recall": recall_score(y_test, y_pred),
        "test_f1": f1_score(y_test, y_pred)
    }
    mlflow.log_metrics(metrics)

    # Enregistrer dans le Model Registry
    mlflow.sklearn.log_model(
        best_model,
        "model",
        registered_model_name="customer_churn_classifier"
    )

    print(f"Meilleurs paramètres : {grid_search.best_params_}")
    print(f"Accuracy : {metrics['test_accuracy']:.4f}")
```

Exemple 2 : Deep Learning avec PyTorch

```
import mlflow
import mlflow.pytorch
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader

# Définir le modèle
class NeuralNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

# Configuration MLflow
mlflow.set_experiment("neural_network_classification")

# Hyperparamètres
params = {
    "input_size": 784,
    "hidden_size": 128,
    "num_classes": 10,
    "learning_rate": 0.001,
    "batch_size": 64,
    "num_epochs": 20
}

with mlflow.start_run():
    # Logger les paramètres
    mlflow.log_params(params)

    # Créer le modèle
    model = NeuralNetwork(
        params["input_size"],
        params["hidden_size"],
        params["num_classes"]
    )

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=params["learning_rate"])

    # Entraînement
    for epoch in range(params["num_epochs"]):
```

```
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for images, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        # Métriques par epoch
        epoch_loss = running_loss / len(train_loader)
        epoch_acc = 100 * correct / total

        mlflow.log_metric("train_loss", epoch_loss, step=epoch)
        mlflow.log_metric("train_accuracy", epoch_acc, step=epoch)

        # Validation
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for images, labels in val_loader:
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                val_total += labels.size(0)
                val_correct += (predicted == labels).sum().item()

        val_epoch_loss = val_loss / len(val_loader)
        val_epoch_acc = 100 * val_correct / val_total

        mlflow.log_metric("val_loss", val_epoch_loss, step=epoch)
        mlflow.log_metric("val_accuracy", val_epoch_acc, step=epoch)

        print(f"Epoch {epoch+1}/{params['num_epochs']}, "
              f"Train Loss: {epoch_loss:.4f}, Train Acc: {epoch_acc:.2f}%,
"
              f"Val Loss: {val_epoch_loss:.4f}, Val Acc:
{val_epoch_acc:.2f}%")

    # Sauvegarder le modèle
    mlflow.pytorch.log_model(model, "model")
```

```
    # Logger l'état du modèle
    torch.save(model.state_dict(), "model_weights.pth")
    mlflow.log_artifact("model_weights.pth")
```

Exemple 3 : Comparaison de modèles

```
import mlflow
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, roc_auc_score

mlflow.set_experiment("model_comparison")

# Modèles à comparer
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(n_estimators=100),
    "Gradient Boosting": GradientBoostingClassifier(n_estimators=100),
    "SVM": SVC(probability=True)
}

# Entraîner et évaluer chaque modèle
results = []

for model_name, model in models.items():
    with mlflow.start_run(run_name=model_name):
        # Tags
        mlflow.set_tag("model_type", model_name)

        # Entraînement
        model.fit(X_train, y_train)

        # Prédictions
        y_pred = model.predict(X_test)
        y_pred_proba = model.predict_proba(X_test)[:, 1]

        # Métriques
        accuracy = accuracy_score(y_test, y_pred)
        roc_auc = roc_auc_score(y_test, y_pred_proba)

        mlflow.log_metric("accuracy", accuracy)
        mlflow.log_metric("roc_auc", roc_auc)

        # Logger le modèle
        mlflow.sklearn.log_model(model, "model")

        results.append({
            "model": model_name,
            "accuracy": accuracy,
```

```
            "roc_auc": roc_auc
        })

        print(f"{model_name}: Accuracy={accuracy:.4f}, ROC-AUC=
{roc_auc:.4f}")

# Trouver le meilleur modèle
best_model_info = max(results, key=lambda x: x['roc_auc'])
print(f"\nMeilleur modèle : {best_model_info['model']}")
```

Bonnes pratiques

Organisation des expériences

1. **Nommage cohérent** : Utilisez une convention de nommage claire pour les expériences

```
# Bon
mlflow.set_experiment("projet_X/classification/modele_Y")

# Éviter
mlflow.set_experiment("test123")
```

2. **Un problème = Une expérience** : Regroupez tous les runs liés à un même problème

3. **Utiliser des tags** : Facilitez la recherche et le filtrage

```
mlflow.set_tag("team", "data-science")
mlflow.set_tag("priority", "high")
mlflow.set_tag("framework", "pytorch")
```

Logging efficace

1. **Logger ce qui compte** : Ne pas surcharger avec trop de métriques
    - Paramètres : Hyperparamètres et configurations importantes
    - Métriques : Métriques de performance clés
    - Artefacts : Visualisations et fichiers essentiels

2. **Nommer explicitement** : Utilisez des noms descriptifs

```
# Bon
mlflow.log_metric("validation_accuracy_epoch_50", 0.89)

# Moins bon
mlflow.log_metric("acc", 0.89)
```

3. **Logger à intervalles réguliers** : Pour les entraînements longs

```
if epoch % 10 == 0:
    mlflow.log_metric("loss", loss, step=epoch)
```

Gestion du Model Registry

1. **Nommage des modèles** : Utilisez des noms significatifs et cohérents

```
# Bon
"customer_churn_random_forest"
"fraud_detection_neural_network"

# Éviter
"model1"
"my_model"
```

2. **Documentation** : Ajoutez des descriptions détaillées

```
client.update_model_version(
    name="customer_churn_rf",
    version=2,
    description="""
    Modèle Random Forest entraîné sur 100k clients.
    Hyperparamètres: n_estimators=200, max_depth=15
    Performance: accuracy=0.89, precision=0.87, recall=0.91
    Date d'entraînement: 2024-01-15
    Jeu de données: customers_v3.csv
    """
)
```

3. **Workflow de validation** : Testez en Staging avant Production

```
# Toujours valider avant de promouvoir
staging_model = mlflow.pyfunc.load_model("models:/mon_modele/Staging")
performance = evaluate_model(staging_model)

if performance > threshold:
    transition_to_production(model_name, version)
```

4. **Versioning sémantique** : Utilisez les tags pour indiquer les changements majeurs

```
client.set_model_version_tag(
    name="mon_modele",
```

```
    version=5,
    key="change_type",
    value="major"  # major, minor, patch
)
```

Sécurité et collaboration

1. **Contrôle d'accès** : Configurez les permissions appropriées sur le serveur
2. **Backend séparé** : Utilisez une base de données dédiée pour la production

```
mlflow server \
    --backend-store-uri postgresql://prod_user:password@prod-db/mlflow
\
    --default-artifact-root s3://prod-bucket/mlflow
```

3. **Backup régulier** : Sauvegardez régulièrement la base de données MLflow
4. **Environnements séparés** : Utilisez des serveurs différents pour dev, staging et production

Performance

1. **Artifact storage** : Utilisez un stockage cloud (S3, Azure Blob, GCS) pour les gros artefacts
2. **Nettoyage** : Supprimez les anciennes expériences et runs inutiles

```
# Supprimer une expérience
mlflow.delete_experiment(experiment_id)

# Supprimer un run
mlflow.delete_run(run_id)
```

3. **Batch logging** : Loggez plusieurs métriques à la fois pour réduire les appels réseau

```
mlflow.log_metrics({
    "metric1": value1,
    "metric2": value2,
    "metric3": value3
})
```

Reproductibilité

1. **Logger l'environnement** : Sauvegardez les dépendances

```
# MLflow génère automatiquement requirements.txt et conda.yaml
mlflow.sklearn.log_model(
    model,
    "model",
    conda_env="conda.yaml",
    pip_requirements="requirements.txt"
)
```

2. **Fixer les seeds** : Pour la reproductibilité

```
import random
import numpy as np

seed = 42
random.seed(seed)
np.random.seed(seed)
mlflow.log_param("random_seed", seed)
```

3. **Logger le code source** : Utilisez Git et loggez le commit hash

```
import git

repo = git.Repo(search_parent_directories=True)
commit_hash = repo.head.object.hexsha
mlflow.set_tag("git_commit", commit_hash)
mlflow.log_param("git_branch", repo.active_branch.name)
```

Ressources supplémentaires

- **Documentation officielle** : https://mlflow.org/docs/latest/index.html
- **API Reference** : https://mlflow.org/docs/latest/python_api/index.html
- **GitHub** : https://github.com/mlflow/mlflow
- **Community Forum** : https://github.com/mlflow/mlflow/discussions

---

Commandes utiles

```
# Démarrer l'interface UI
mlflow ui

# Démarrer le serveur avec options
mlflow server --help

# Exporter une expérience
```

```
mlflow experiments export --experiment-id 1 --output-dir ./export

# Importer une expérience
mlflow experiments import --input-dir ./export

# Chercher des runs
mlflow runs list --experiment-id 1

# Voir les détails d'un run
mlflow runs describe --run-id <run_id>
```

Version : 1.0
Dernière mise à jour : Décembre 2024