Projet PLURI : Optimiser la durée des arrêts de tranche avec un modèle ML

By David Tchatchoua

Pour optimiser la durée des arrêts de tranche, un modèle de **régression supervisée** peut être utilisé. L'objectif est de prédire la durée en fonction des facteurs influents, tels que :

- **Type de maintenance** : préventive ou corrective.
- **Complexité des travaux**.
- **Historique des arrêts similaires**.
- **Effectif mobilisé**.
- **Disponibilité des ressources et pièces détachées**.

Un modèle comme **Gradient Boosting Regressor** (ex. : XGBoost ou LightGBM) est bien adapté à ce type de problème grâce à sa capacité à gérer des relations complexes entre variables.

Étapes du projet

1. **Création de données synthétiques** : Simuler un ensemble de données réalistes.
2. **Préparation et nettoyage des données**.
3. **Entraînement d'un modèle de régression**.
4. **Évaluation du modèle sur un ensemble de test**.

Code Python avec données synthétiques

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_absolute_error, r2_score
import matplotlib.pyplot as plt

# 1. Création de données synthétiques
np.random.seed(42)
data_size = 1000

# Variables indépendantes
```

```
types_maintenance = np.random.choice(["préventive", "corrective"], data_size, p=[0.7,
0.3])
complexité = np.random.randint(1, 10, data_size)  # 1 = faible, 10 = très élevée
effectif = np.random.randint(5, 50, data_size)  # Effectif mobilisé
ressources_disponibles = np.random.rand(data_size)  # 0.0 à 1.0 pour refléter la
disponibilité

# Durée (target) : générée en fonction des variables
durée = (
    5 * (types_maintenance == "corrective").astype(int)
    + 2 * complexité
    - 0.1 * effectif
    + 10 * (1 - ressources_disponibles)
    + np.random.normal(0, 5, data_size)  # Bruit aléatoire
)

# DataFrame
df = pd.DataFrame({
    "Type_Maintenance": types_maintenance,
    "Complexité": complexité,
    "Effectif": effectif,
    "Ressources_Disponibles": ressources_disponibles,
    "Durée": durée
})

# Encodage de la variable catégorielle
df = pd.get_dummies(df, columns=["Type_Maintenance"], drop_first=True)

# 2. Division des données en train/test
X = df.drop(columns=["Durée"])
y = df["Durée"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 3. Entraînement du modèle
model = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, max_depth=5,
random_state=42)
model.fit(X_train, y_train)

# 4. Évaluation du modèle
y_pred = model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Absolute Error: {mae:.2f}")
print(f"R2 Score: {r2:.2f}")

# 5. Visualisation des prédictions vs valeurs réelles
plt.scatter(y_test, y_pred, alpha=0.7)
plt.xlabel("Valeurs réelles (Durée)")
plt.ylabel("Prédictions (Durée)")
```

```
plt.title("Prédictions vs Réalité")
plt.show()
```

Explication du code

1. **Données synthétiques** : Simulent des facteurs influençant la durée des arrêts.
2. **Encodage** : La variable catégorielle "Type_Maintenance" est convertie en indicateurs binaires pour être utilisée par le modèle.
3. **Gradient Boosting Regressor** : Modèle puissant pour les régressions non linéaires.
4. **Évaluation** : Utilise le **MAE** (erreur absolue moyenne) et le **R²** pour mesurer les performances du modèle.
5. **Visualisation** : Compare les prédictions aux valeurs réelles.

Ce modèle peut être amélioré avec des données réelles ou supplémentaires, comme :

- **Temps moyen d'arrêt par tranche similaire**.
- **Historique de performances des équipes**.
- **Météo ou autres facteurs externes**.

Pour approfondir l'optimisation et la mise en production du modèle, voici les étapes détaillées :

---

1. Optimisation du Modèle

a. Hyperparameter Tuning

L'objectif est d'améliorer les performances en ajustant les hyperparamètres du modèle. Cela peut être fait avec des techniques comme **Grid Search**, **Random Search**, ou **Bayesian Optimization**.

Code pour Grid Search :

```
from sklearn.model_selection import GridSearchCV

# Définir les hyperparamètres à tester
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'subsample': [0.8, 1.0]
}

# Configurer le Grid Search
grid_search = GridSearchCV(
    GradientBoostingRegressor(random_state=42),
    param_grid,
    cv=5,  # Validation croisée
    scoring='neg_mean_absolute_error',
    verbose=2,
    n_jobs=-1
)
```

```
grid_search.fit(X_train, y_train)
best_model = grid_search.best_estimator_

print(f"Meilleurs paramètres : {grid_search.best_params_}")
print(f"Meilleure performance (MAE) : {-grid_search.best_score_:.2f}")
```

b. Feature Importance

Identifier les facteurs les plus influents sur la durée pour guider les interventions opérationnelles.

```
importances = best_model.feature_importances_
sorted_indices = np.argsort(importances)[::-1]

# Visualisation
plt.figure(figsize=(10, 6))
plt.bar(range(X.shape[1]), importances[sorted_indices], align='center')
plt.xticks(range(X.shape[1]), X.columns[sorted_indices], rotation=45)
plt.title("Importance des caractéristiques")
plt.show()
```

---

2. Mise en Production

Une fois le modèle optimisé, voici les étapes pour le déployer et le monitorer dans un environnement de production.

a. Containerisation avec Docker

Créer un conteneur Docker pour le modèle afin de simplifier son déploiement sur des plateformes comme **AWS SageMaker**, **Azure ML**, ou **Kubernetes**.

Dockerfile :

```
# Utiliser une image de base légère
FROM python:3.9-slim

# Installer les dépendances
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copier le code du modèle
COPY model.pkl app.py ./

# Lancer l'application Flask
CMD ["python", "app.py"]
```

Fichier `app.py` :

```
from flask import Flask, request, jsonify
import pickle
import numpy as np

app = Flask(__name__)
```

```
# Charger le modèle
with open('model.pkl', 'rb') as f:
    model = pickle.load(f)

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    X = np.array([data['features']])
    prediction = model.predict(X)[0]
    return jsonify({"prediction": prediction})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

b. Déploiement sur AWS SageMaker

1. **Uploader le modèle dans S3.**
2. **Créer un endpoint SageMaker** à partir de l'image Docker ou directement avec le modèle.

```
from sagemaker.model import Model

model = Model(
    image_uri='your-docker-image-uri',
    model_data='s3://your-bucket/model.tar.gz',
    role='your-sagemaker-role'
)

predictor = model.deploy(
    initial_instance_count=1,
    instance_type='ml.m5.large',
    endpoint_name='predict-arrets-tranche'
)
```

---

3. Monitoring et Mise à Jour

a. Monitoring des Performances

Utilisez des outils comme **AWS CloudWatch** ou **Prometheus + Grafana** pour suivre :

- Le temps de réponse des prédictions.
- La disponibilité des endpoints.
- Les erreurs ou anomalies.

Exemple de métrique CloudWatch :

```
from sagemaker.model_monitor import ModelMonitor

monitor = ModelMonitor.attach(endpoint_name='predict-arrets-tranche')
monitor.describe_schedule()
```

b. Automatisation des mises à jour

Configurez une **pipeline CI/CD** pour automatiser :

- Le réentraînement périodique avec des données nouvelles (par ex. via AWS Step Functions).
- Le déploiement des nouveaux modèles via **GitHub Actions** ou **AWS CodePipeline**.

Automatisation des mises à jour et CI/CD pour un modèle ML

Mettre en place une **pipeline CI/CD (Intégration et Déploiement Continus)** permet d'automatiser le réentraînement, la validation, et le déploiement de votre modèle. Voici un guide détaillé pour le faire, en prenant AWS comme exemple.

---

1. Architecture de la Pipeline CI/CD

La pipeline automatisera les étapes suivantes :

1. **Ingestion des nouvelles données** : collecter de nouvelles données pertinentes pour améliorer le modèle.
2. **Prétraitement et Feature Engineering** : nettoyer et transformer les données.
3. **Entraînement et Validation** : tester la nouvelle version du modèle.
4. **Déploiement en production** : remplacer le modèle actuel si la nouvelle version est meilleure.

Outils nécessaires :

- **AWS CodePipeline** : orchestration des étapes.
- **AWS CodeBuild** : exécution des scripts d'entraînement et de tests.
- **Amazon S3** : stockage des données et modèles.
- **AWS SageMaker** : réentraînement et déploiement.
- **GitHub Actions** (optionnel) : pour lier les changements dans le dépôt à la pipeline.

---

2. Étapes de Configuration

a. Configuration des Données

1. **Créer un bucket Amazon S3** pour stocker :
    - Données d'entraînement (`s3://your-bucket/data/`).
    - Artefacts du modèle (`s3://your-bucket/models/`).
2. **Uploader des données** dans le bucket (manuellement ou via une tâche automatisée).

---

b. Script d'Entraînement Automatisé

Un script Python pour réentraîner le modèle avec les nouvelles données :

```
import boto3
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
import joblib
```

```
# Télécharger les données depuis S3
s3 = boto3.client('s3')
s3.download_file('your-bucket', 'data/train.csv', 'train.csv')

# Charger les données
df = pd.read_csv('train.csv')
X = df.drop(columns=['Durée'])
y = df['Durée']

# Entraîner le modèle
model = GradientBoostingRegressor(n_estimators=100, max_depth=5, random_state=42)
model.fit(X, y)

# Sauvegarder le modèle
joblib.dump(model, 'model.pkl')

# Uploader le modèle vers S3
s3.upload_file('model.pkl', 'your-bucket', 'models/model.pkl')
```

---

c. Configuration de la Pipeline AWS

1. **Étape 1 : Extraction des données**
   - Une tâche dans AWS Step Functions pour surveiller S3 et extraire les nouvelles données.
2. **Étape 2 : Entraînement du modèle avec AWS CodeBuild**
   - Définir un fichier `buildspec.yml` pour entraîner le modèle :

```
version: 0.2

phases:
  install:
    commands:
      - pip install -r requirements.txt
  build:
    commands:
      - python train_model.py  # Script d'entraînement
artifacts:
  files:
    - model.pkl
  discard-paths: yes
  base-directory: ./
```

3. **Étape 3 : Validation automatique**
   - Ajouter un script pour comparer les performances des modèles. Exemple :

```
from sklearn.metrics import mean_squared_error

# Charger l'ancien et le nouveau modèle
```

```
old_model = joblib.load('old_model.pkl')
new_model = joblib.load('model.pkl')

# Calculer les performances
old_perf = mean_squared_error(y_test, old_model.predict(X_test))
new_perf = mean_squared_error(y_test, new_model.predict(X_test))

if new_perf < old_perf:
    print("Le nouveau modèle est meilleur !")
else:
    print("Conserver l'ancien modèle.")
```

4. **Étape 4 : Déploiement automatique avec AWS SageMaker**
    - Remplacez le modèle sur l'endpoint SageMaker si la validation est réussie :

```
from sagemaker.model import Model

model = Model(
    image_uri='your-docker-image-uri',
    model_data='s3://your-bucket/models/model.pkl',
    role='your-sagemaker-role'
)

predictor = model.deploy(
    initial_instance_count=1,
    instance_type='ml.m5.large',
    endpoint_name='predict-arrets-tranche'
)
```

---

3. Surveiller et Automatiser les Déploiements

a. Alertes avec CloudWatch

Configurez des métriques comme :

- Latence du modèle.
- Taux d’erreur (`4XX` , `5XX`).
- Performances des nouvelles prédictions.

b. Mise à jour continue avec GitHub Actions

Déclenchez des mises à jour automatiques quand un nouveau script ou modèle est poussé dans le dépôt :

```
name: Deploy Model

on:
  push:
    branches:
      - main
```

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Train and Deploy Model
        run: |
          pip install -r requirements.txt
          python train_model.py
          python deploy_model.py
```

Conclusion

Avec une pipeline CI/CD, votre système devient autonome :

1. Nouvelles données → Entraînement automatique.
2. Validation → Déploiement si performances améliorées.
3. Monitoring → Alertes en cas d'anomalie.

Guide détaillé pour orchestrer la pipeline avec AWS Step Functions

AWS Step Functions permet de coordonner plusieurs services AWS pour automatiser une pipeline de bout en bout. Voici comment l'utiliser pour automatiser l'entraînement et le déploiement de votre modèle.

---

1. Architecture de la Pipeline Orchestrée

Avec AWS Step Functions, la pipeline comprend :

1. **Étape de Prétraitement** : Nettoyage et préparation des données (via AWS Lambda ou SageMaker Processing).
2. **Étape d'Entraînement** : Exécution du script d'entraînement sur AWS SageMaker.
3. **Étape de Validation** : Comparaison des performances des modèles (Lambda).
4. **Étape de Déploiement** : Déploiement du modèle sur un endpoint SageMaker si les performances sont satisfaisantes.

---

2. Configuration des Services AWS

a. AWS Lambda

Lambda est utilisé pour exécuter des tâches légères comme le déclenchement ou la validation.

- **Script Lambda pour valider les performances :**

```
import boto3
import joblib
from sklearn.metrics import mean_squared_error

def lambda_handler(event, context):
```

```
    s3 = boto3.client('s3')

    # Télécharger les modèles depuis S3
    s3.download_file('your-bucket', 'models/old_model.pkl',
'/tmp/old_model.pkl')
    s3.download_file('your-bucket', 'models/new_model.pkl',
'/tmp/new_model.pkl')

    # Charger les modèles
    old_model = joblib.load('/tmp/old_model.pkl')
    new_model = joblib.load('/tmp/new_model.pkl')

    # Charger les données de test
    s3.download_file('your-bucket', 'data/test.csv', '/tmp/test.csv')
    import pandas as pd
    test_data = pd.read_csv('/tmp/test.csv')
    X_test = test_data.drop(columns=['Durée'])
    y_test = test_data['Durée']

    # Calculer les performances
    old_perf = mean_squared_error(y_test, old_model.predict(X_test))
    new_perf = mean_squared_error(y_test, new_model.predict(X_test))

    if new_perf < old_perf:
        return {"deploy": True, "new_perf": new_perf, "old_perf": old_perf}
    else:
        return {"deploy": False, "new_perf": new_perf, "old_perf": old_perf}
```

b. SageMaker Training Job

L'entraînement est réalisé avec AWS SageMaker en configurant un job d'entraînement.

- **Configuration SageMaker :**

```
import boto3

sagemaker = boto3.client('sagemaker')

response = sagemaker.create_training_job(
    TrainingJobName='training-job-arrets-tranche',
    AlgorithmSpecification={
        'TrainingImage': 'your-training-image',
        'TrainingInputMode': 'File'
    },
    RoleArn='arn:aws:iam::your-account-id:role/your-sagemaker-role',
    InputDataConfig=[
        {
            'ChannelName': 'training',
            'DataSource': {
                'S3DataSource': {
                    'S3DataType': 'S3Prefix',
                    'S3Uri': 's3://your-bucket/data/train/',
                    'S3DataDistributionType': 'FullyReplicated'
```

```
                }
            },
            'ContentType': 'text/csv'
        }
    ],
    OutputDataConfig={
        'S3OutputPath': 's3://your-bucket/models/'
    },
    ResourceConfig={
        'InstanceType': 'ml.m5.large',
        'InstanceCount': 1,
        'VolumeSizeInGB': 10
    },
    StoppingCondition={'MaxRuntimeInSeconds': 3600}
)
```

---

3. Définir une Machine d’État Step Functions

Définition JSON de la pipeline

La machine d'état AWS Step Functions suit une séquence d'étapes définies dans un fichier JSON. Exemple de définition :

```
{
  "Comment": "Pipeline ML pour optimiser les arrêts de tranche",
  "StartAt": "Prétraitement des données",
  "States": {
    "Prétraitement des données": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:region:account-id:function:preprocess-data",
      "Next": "Entraînement du modèle"
    },
    "Entraînement du modèle": {
      "Type": "Task",
      "Resource": "arn:aws:states:region:sagemaker:createTrainingJob",
      "Parameters": {
        "TrainingJobName.$": "$.TrainingJobName",
        "AlgorithmSpecification": {
          "TrainingImage": "your-training-image",
          "TrainingInputMode": "File"
        },
        "RoleArn": "arn:aws:iam::account-id:role/your-sagemaker-role",
        "InputDataConfig": [{
          "ChannelName": "training",
          "DataSource": {
            "S3DataSource": {
              "S3Uri": "s3://your-bucket/data/train/",
              "S3DataType": "S3Prefix",
              "S3DataDistributionType": "FullyReplicated"
            }
          }
```

```
        }],
        "OutputDataConfig": {
          "S3OutputPath": "s3://your-bucket/models/"
        },
        "ResourceConfig": {
          "InstanceType": "ml.m5.large",
          "InstanceCount": 1,
          "VolumeSizeInGB": 10
        },
        "StoppingCondition": {
          "MaxRuntimeInSeconds": 3600
        }
      },
      "Next": "Validation du modèle"
    },
    "Validation du modèle": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:region:account-id:function:validate-model",
      "Next": "Déploiement ou Terminaison"
    },
    "Déploiement ou Terminaison": {
      "Type": "Choice",
      "Choices": [{
        "Variable": "$.deploy",
        "BooleanEquals": true,
        "Next": "Déploiement"
      }],
      "Default": "Fin"
    },
    "Déploiement": {
      "Type": "Task",
      "Resource": "arn:aws:states:region:sagemaker:createModel",
      "End": true
    },
    "Fin": {
      "Type": "Succeed"
    }
  }
}
```

---

4. Déploiement du Modèle

Le modèle validé est automatiquement déployé sur un endpoint SageMaker ou intégré dans une application via API.

- **Créer un Endpoint SageMaker :**

```
from sagemaker.model import Model

model = Model(
    image_uri='your-model-image',
    model_data='s3://your-bucket/models/model.tar.gz',
```

```
    role='arn:aws:iam::account-id:role/your-sagemaker-role'
)

predictor = model.deploy(
    initial_instance_count=1,
    instance_type='ml.m5.large',
    endpoint_name='optimisation-arrets-tranche'
)
```

Pour configurer **AWS Lambda** dans votre pipeline d'orchestration, voici un guide étape par étape :

---

1. Création d’une Fonction AWS Lambda

1. **Accéder à la Console AWS Lambda** :
   - Allez sur la console AWS Lambda.
   - Cliquez sur **Créer une fonction**.
2. **Choisir les Paramètres de la Fonction** :
   - **Type** : Créer une fonction à partir de zéro.
   - **Nom de la fonction** : Donnez un nom explicite, par exemple `validate_model_performance`.
   - **Runtime** : Sélectionnez `Python 3.x`.
   - **Role IAM** : Choisissez un rôle existant avec les permissions nécessaires, ou créez-en un nouveau (voir point 2).
3. **Créer la Fonction**.

---

2. Configuration des Permissions Lambda

Lambda nécessite des permissions pour accéder aux services AWS comme S3, SageMaker, ou CloudWatch.

Créer un Rôle IAM pour Lambda :

1. Accédez à la console IAM.
2. Cliquez sur **Créer un rôle**.
3. **Sélectionnez Lambda** comme cas d’utilisation.
4. Ajoutez les politiques suivantes :
   - `AmazonS3FullAccess` (accès S3 pour lire/écrire les données ou modèles).
   - `AmazonSageMakerFullAccess` (si nécessaire pour interagir avec SageMaker).
   - `CloudWatchLogsFullAccess` (pour surveiller les journaux Lambda).
5. Nommez le rôle, par exemple : `lambda-sagemaker-role`.

---

3. Développer et Tester le Code Lambda

Exemple : Fonction de Validation des Performances

Ce script compare les performances d'un nouveau modèle à un ancien modèle stocké dans S3.

Code Python pour Lambda :

```
import boto3
import joblib
import pandas as pd
from sklearn.metrics import mean_squared_error

def lambda_handler(event, context):
    s3 = boto3.client('s3')

    # Informations sur le bucket et les fichiers
    bucket_name = "your-bucket"
    old_model_key = "models/old_model.pkl"
    new_model_key = "models/new_model.pkl"
    test_data_key = "data/test.csv"

    # Télécharger les fichiers depuis S3
    s3.download_file(bucket_name, old_model_key, '/tmp/old_model.pkl')
    s3.download_file(bucket_name, new_model_key, '/tmp/new_model.pkl')
    s3.download_file(bucket_name, test_data_key, '/tmp/test.csv')

    # Charger les modèles et les données
    old_model = joblib.load('/tmp/old_model.pkl')
    new_model = joblib.load('/tmp/new_model.pkl')
    test_data = pd.read_csv('/tmp/test.csv')

    X_test = test_data.drop(columns=['Durée'])
    y_test = test_data['Durée']

    # Calculer les performances
    old_perf = mean_squared_error(y_test, old_model.predict(X_test))
    new_perf = mean_squared_error(y_test, new_model.predict(X_test))

    # Résultat : Déployer ou non
    if new_perf < old_perf:
        return {
            "statusCode": 200,
            "body": {
                "deploy": True,
                "new_perf": new_perf,
                "old_perf": old_perf
            }
        }
    else:
        return {
            "statusCode": 200,
            "body": {
                "deploy": False,
                "new_perf": new_perf,
```

```
                "old_perf": old_perf
            }
        }
```

---

4. Ajouter la Fonction Lambda à Step Functions

Dans Step Functions, ajoutez Lambda comme tâche dans votre définition JSON.

Exemple :

```
{
  "States": {
    "Validation du modèle": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:region:account-
id:function:validate_model_performance",
      "Next": "Déploiement ou Terminaison"
    },
    ...
  }
}
```

---

5. Tester la Fonction Lambda

1. Accédez à la console Lambda.
2. Cliquez sur **Tester** et configurez un événement de test :

   ```
   {
       "bucket_name": "your-bucket",
       "test_data_key": "data/test.csv",
       "old_model_key": "models/old_model.pkl",
       "new_model_key": "models/new_model.pkl"
   }
   ```

3. Cliquez sur **Exécuter** pour vérifier les résultats.

---

6. Surveiller Lambda avec CloudWatch Logs

- Les journaux Lambda sont automatiquement envoyés à **CloudWatch Logs**.
- Accédez à CloudWatch pour visualiser les journaux :
  1. Console AWS > CloudWatch > Logs.
  2. Trouvez le groupe de journaux correspondant à votre fonction Lambda.
  3. Vérifiez les erreurs et performances.