Mise en situation 2 (E3)

Intégrer des modèles et des services d'intelligence artificielle

Projet AniMOV : Surveillance et Analyse Comportementale des Chèvres avec l'IA

Formation Développeur en Intelligence Artificielle
RNCP 37827
Promotion 2023-2024

Manuel CALDEIRA

SIMPLON.CO

académie Orléans-Tours
GRETA VAL DE LOIRE
FORMEZ-VOUS À DEMAIN

Microsoft en France
Ecole IA

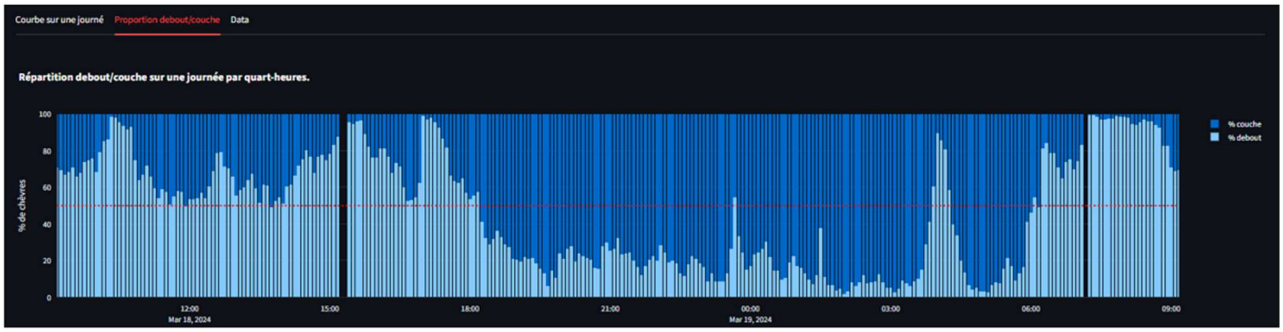
Table des matières

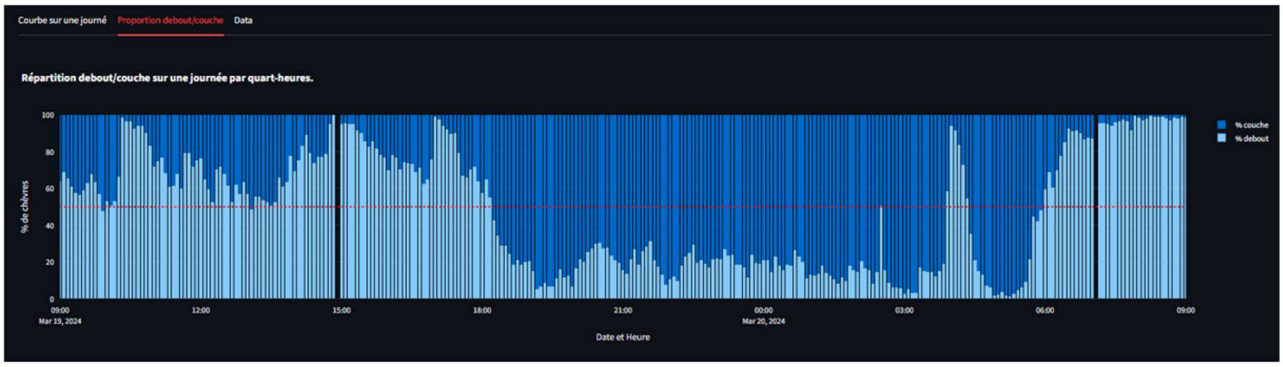
1. INTRODUCTION

1.1. OBJECTIFS DU RAPPORT

Ce rapport a pour but de présenter de manière détaillée les différentes étapes suivies pour le développement, l'intégration, le monitoring, les tests, et le déploiement d'un modèle SARIMAX. Ce modèle est utilisé pour prédire le ratio debout/couché des chèvres en se basant sur des données historiques. Le document servira de support pour démontrer les compétences acquises en développement de projets en Intelligence Artificielle (IA), tout en offrant une vision complète des défis rencontrés et des solutions apportées.

1.2. ANALYSE DES COMPORTEMENTS CYCLIQUES ET JUSTIFICATION DE L'USAGE DU MODELE SARIMAX



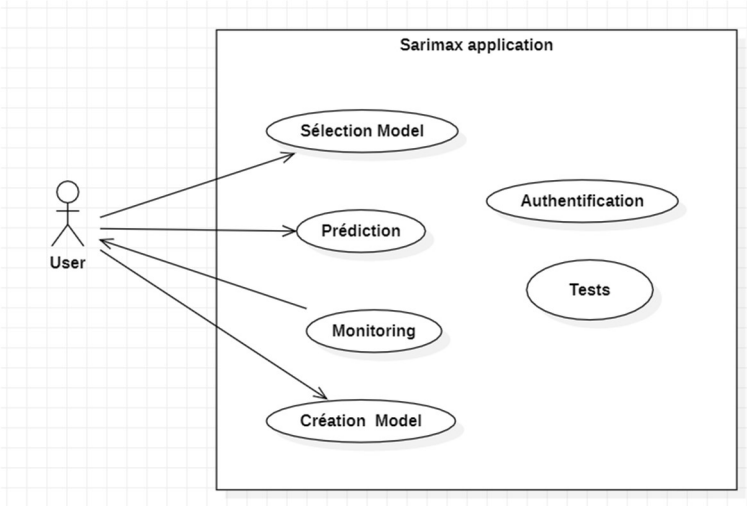


L'observation des données comportementales des chèvres sur une période de 24 heures révèle des variations significatives dans le pourcentage de temps passé debout ou couché. Ces variations suivent des cycles bien définis, avec des périodes de repos prédominantes durant les heures nocturnes et des périodes d'activité plus marquées en journée.

Le modèle SARIMAX (Seasonal Autoregressive Integrated Moving Average with Exogenous Regressors) a été choisi pour capturer ces comportements cycliques. Ce modèle est particulièrement adapté pour modéliser et prévoir des séries temporelles présentant des tendances saisonnières, en tenant compte des effets des variables exogènes. Par exemple, les conditions météorologiques ou les cycles lunaires, qui ne sont pas intrinsèquement liés aux données internes de la série temporelle, peuvent influencer les comportements des chèvres. L'intégration de ces variables exogènes dans le modèle permet d'améliorer la précision des prévisions, en offrant une approche robuste pour anticiper les changements de comportement.

2. USER CASE

2.1. CAS D'UTILISATION : APPLICATION SARIMAX AVEC AUTHENTIFICATION



Le cas d'utilisation présenté concerne une application dédiée à la modélisation et à la prédiction de séries temporelles en utilisant le modèle SARIMAX. Cette application intègre une fonctionnalité d'authentification, garantissant que seules les personnes autorisées peuvent accéder aux données et aux outils de modélisation.

2.2. DESCRIPTION GENERALE

L'application permet à l'utilisateur de gérer tout le cycle de vie de la modélisation, depuis la sélection du modèle jusqu'à l'analyse des prédictions. Le système d'authentification assure que les accès aux fonctions critiques, telles que l'entraînement et la prédiction, sont sécurisés.

2.3. ACTEURS

- **Utilisateur :**

  L'acteur principal de ce cas d'utilisation. L'utilisateur doit d'abord s'authentifier pour accéder aux fonctionnalités de l'application, ce qui inclut la gestion des modèles et l'exécution de prédictions.

2.4. FONCTIONS PRINCIPALES

L'application offre plusieurs fonctionnalités clés :

- **Authentification :**

  Avant toute interaction avec l'application, l'utilisateur doit passer par un processus d'authentification pour sécuriser l'accès aux données et aux outils de modélisation.

- **Sélection du Modèle :**

  Une fois authentifié, l'utilisateur peut choisir un modèle SARIMAX existant ou en créer un nouveau, en configurant les paramètres appropriés, tels que les ordres ARIMA et les variables exogènes.

- **Création du Modèle :**

  Si aucun modèle existant ne convient, l'utilisateur a la possibilité de créer un nouveau modèle.

- **Prédiction :**

  Après sélection ou création du modèle, l'utilisateur peut générer des prédictions basées sur les données historiques et les facteurs exogènes.

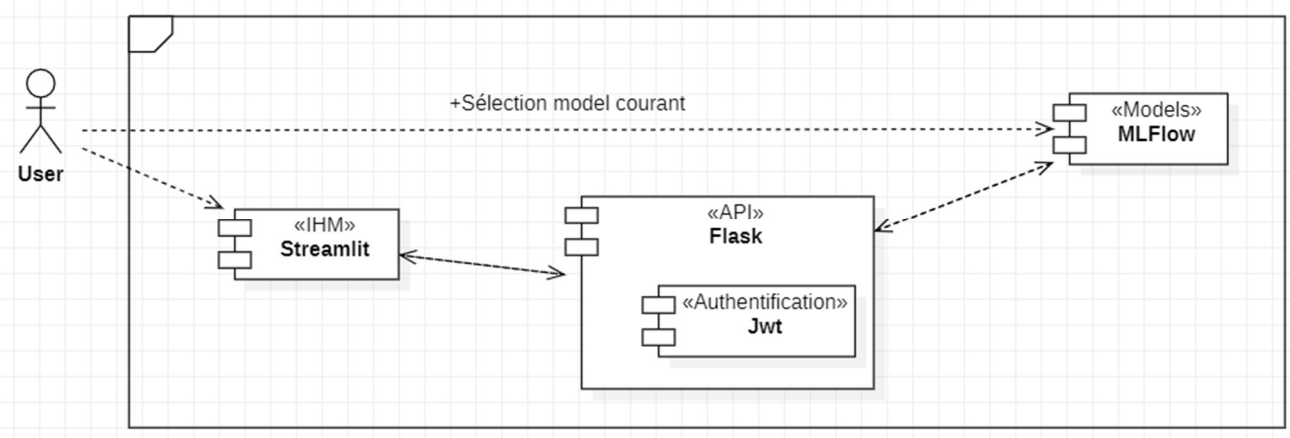
- **Monitoring :**

  L'application permet de suivre la performance du modèle au fil du temps, en vérifiant la précision des prédictions et en ajustant le modèle si nécessaire.

- **Tests :**

  L'utilisateur peut tester le modèle avant de l'utiliser en production.

3. ARCHITECTURE



3.1. CONTEXTE

L'architecture de l'application repose sur plusieurs composants clés, chacun ayant un rôle précis dans le processus de modélisation et de prédiction. L'interface utilisateur (IHM) est développée avec Streamlit, offrant une interaction fluide pour l'utilisateur. L'API, implémentée avec Flask, assure la communication entre les différents composants et gère les requêtes. Un système de gestion des modèles est intégré via MLFlow, permettant de suivre et de gérer les versions des modèles utilisés. Enfin, un système d'authentification basé sur JSON Web Tokens (JWT) garantit que seules les personnes autorisées peuvent accéder aux fonctionnalités de l'application.

3.2. COMPOSANTS DE L'ARCHITECTURE

- **Utilisateur :**

  L'acteur central qui interagit avec l'application via l'interface Streamlit.

- **Streamlit (IHM) :**

  Fournit une interface utilisateur intuitive pour la sélection du modèle et la génération de prédictions.

- **Flask (API) :**

  Moteur principal de l'application côté serveur, gérant les requêtes et l'interaction avec MLFlow.

- **Authentification JWT :**

  Assure la sécurité en vérifiant que l'utilisateur est autorisé avant toute interaction avec l'API.

- **MLFlow (Gestion des Modèles) :**

  Plateforme utilisée pour suivre, stocker et gérer les modèles de machine learning, facilitant leur récupération pour les prédictions.

3.3. INTERACTION ENTRE LES COMPOSANTS

L'interaction entre les composants suit un flux de travail bien défini :

- **Authentification :**

  L'utilisateur s'authentifie via l'interface Streamlit. Une fois validé, un token JWT est généré et stocké pour des communications futures.

- **Sélection du Modèle :**

  L'utilisateur choisit un modèle via Streamlit, qui envoie une requête à Flask pour récupérer le modèle sélectionné depuis MLFlow.

- **Prédiction :**

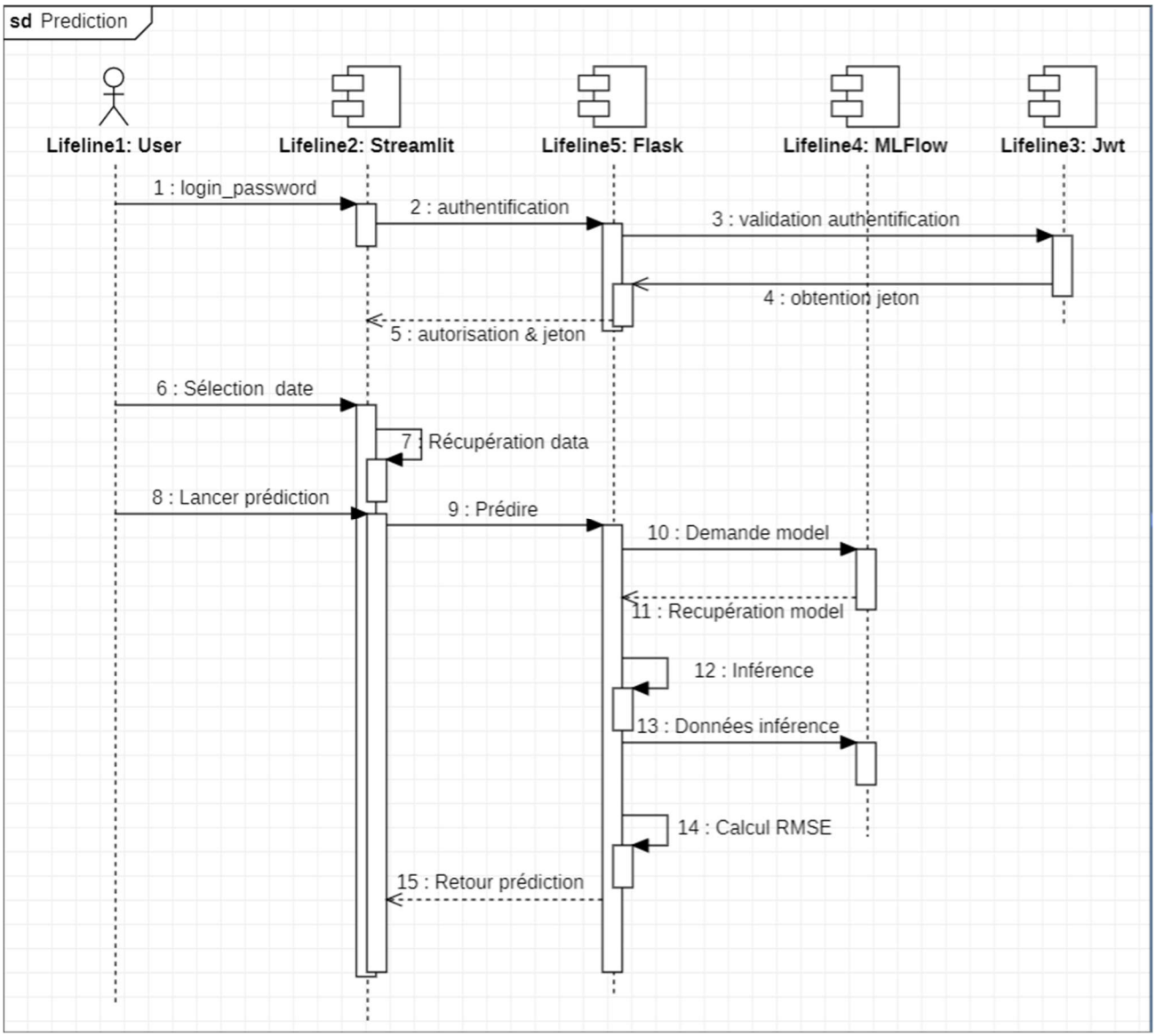
  Flask utilise le modèle récupéré pour effectuer les prédictions demandées, qui sont ensuite renvoyées et affichées dans l'interface Streamlit.

- **Monitoring :**

  Les performances du modèle sont suivies et affichées, permettant des ajustements si nécessaire.

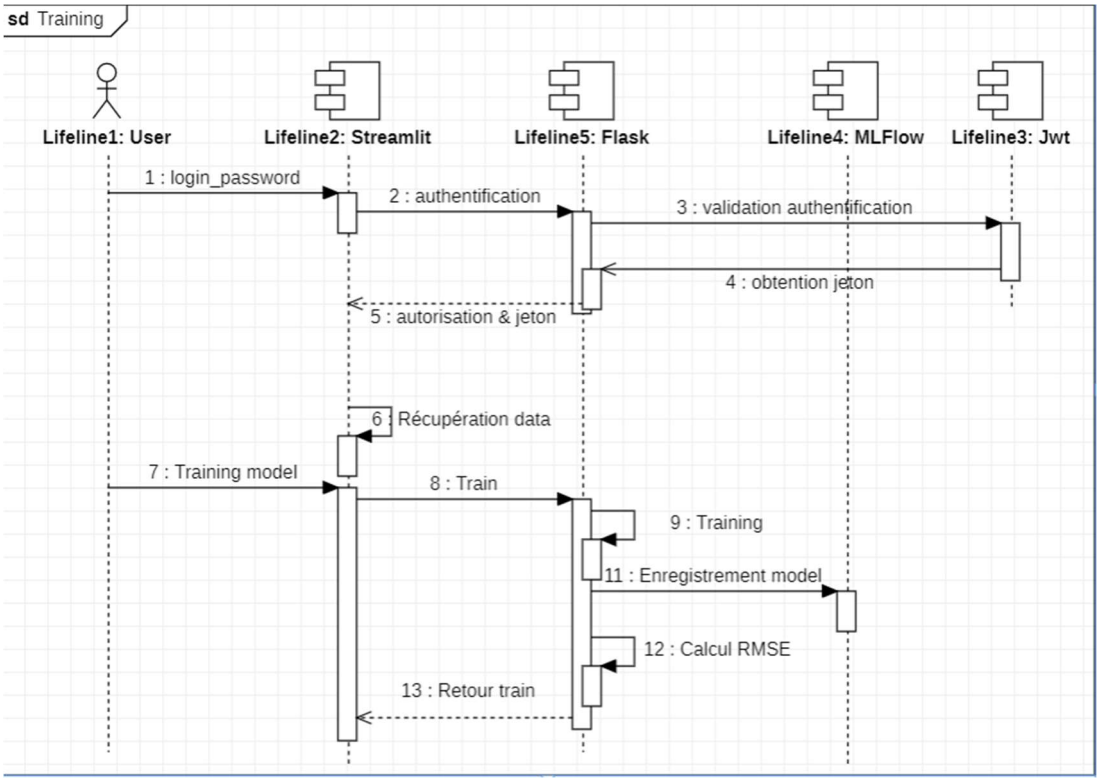
4. DIAGRAMMES DE SEQUENCES

4.1. DIAGRAMME DE SEQUENCE : PROCESSUS DE PREDICTION DANS L'APPLICATION SARIMAX



Le diagramme de séquence décrit les étapes suivies lors de la prédiction. L'utilisateur s'authentifie d'abord, puis sélectionne les données pour la prédiction. Ces données sont envoyées à l'API Flask, qui récupère le modèle SARIMAX via MLFlow, exécute la prédiction et renvoie les résultats à l'utilisateur.

4.2. PROCESSUS D'ENTRAINEMENT DANS L'APPLICATION SARIMAX



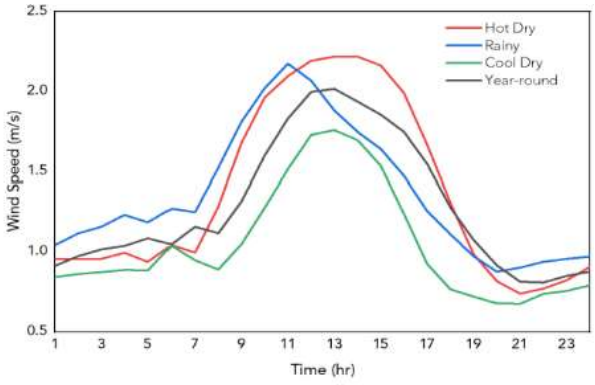
Ce diagramme montre les interactions entre les composants lors de l'entraînement du modèle. Après authentification, l'utilisateur envoie les données d'entraînement via Streamlit. Flask traite ces données et utilise MLFlow pour entraîner le modèle. Une fois l'entraînement terminé, les résultats sont retournés à l'utilisateur, complétant ainsi le cycle de formation du modèle.

5. LE MODEL SARIMAX

Le modèle SARIMAX, ou Seasonal Autoregressive Integrated Moving Average with Exogenous Regressors, est un outil puissant pour la prévision de séries temporelles. Ce modèle combine plusieurs aspects essentiels :

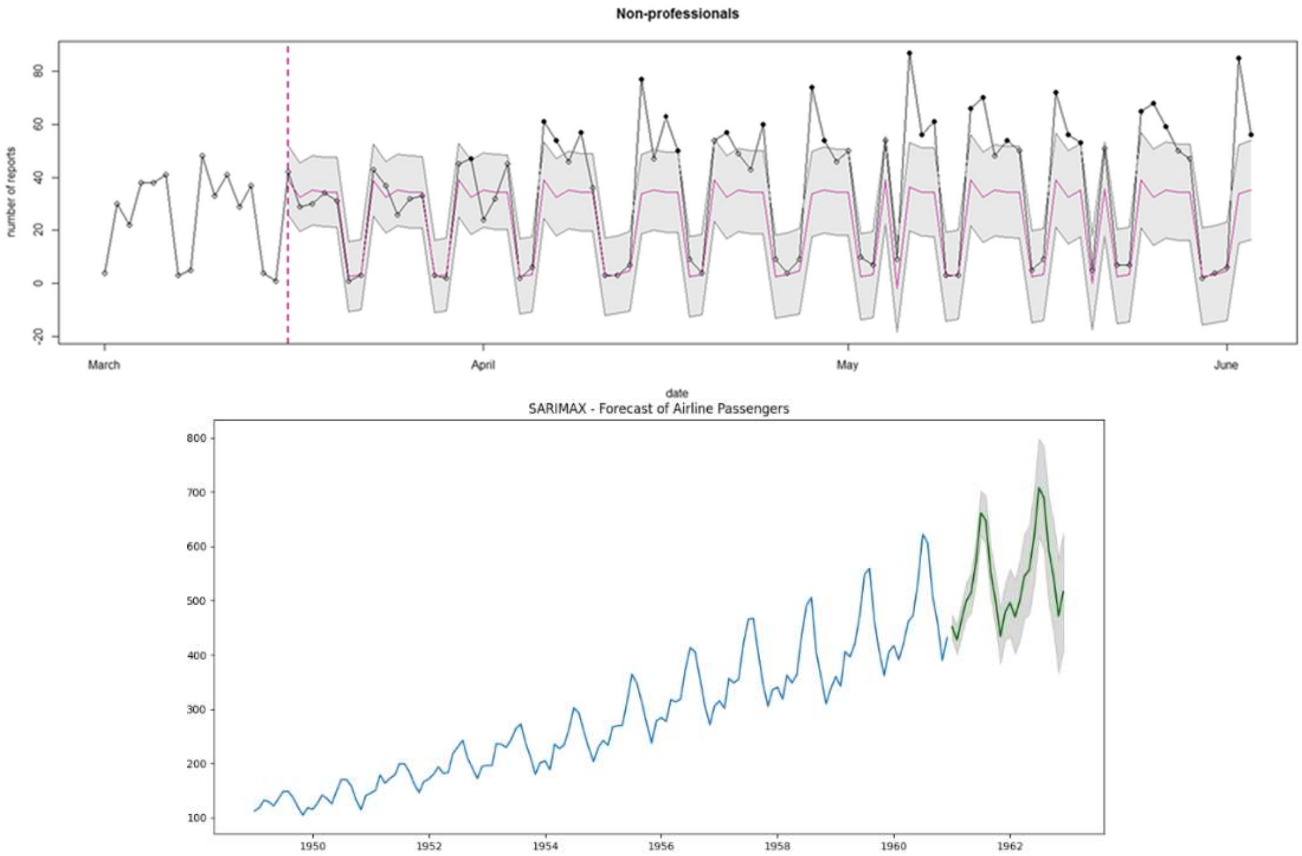
- **Tendances Passées :**

  Analyse des données historiques pour identifier des motifs récurrents.



- **Saisonnalité :**

  Intègre les variations saisonnières, ajustant les prédictions en fonction des périodes spécifiques.



- **Variables Exogènes :**

  Prend en compte des facteurs externes influençant les données, comme les conditions météorologiques ou des événements spécifiques.

SARIMAX permet ainsi de réaliser des prédictions plus précises en tenant compte de l'ensemble des facteurs influençant les séries temporelles analysées.

6. Developpement d'une API REST pour le Modele SARIMAX

6.1. Choix du Framework et des Outils

Le développement de l'API REST a nécessité la sélection d'outils et de frameworks pour répondre aux besoins spécifiques du projet. Voici les choix principaux :

- **Flask :**

  Choisi pour sa légèreté et sa flexibilité, idéal pour gérer les requêtes HTTP et s'intégrer avec les bibliothèques Python.

- **Flasgger :**

  Intégré pour créer une interface Swagger, facilitant la documentation et les tests de l'API.

- **MLFlow :**

  Utilisé pour la gestion des modèles de machine learning, permettant de suivre les versions, les paramètres, et les métriques associées.

- **SQLite :**

  Base de données légère utilisée pour stocker les informations d'authentification des utilisateurs et les tokens.

6.2. ARCHITECTURE DE L'API

L'architecture de l'API repose sur une structure modulaire pour maximiser la flexibilité et la maintenabilité. Elle est organisée en plusieurs couches :

- **Couche de Présentation (Endpoints) :**

  Gère les interactions directes avec les utilisateurs via des endpoints RESTful. Les principales fonctionnalités incluent l'authentification, l'entraînement du modèle, la prédiction, et la gestion des historiques des erreurs de prédiction (RMSE).

- **Couche Logique (Business Logic) :**

  Encapsule la logique métier, incluant les processus d'entraînement et de prédiction du modèle SARIMAX.

- **Couche d'Intégration avec le Machine Learning :**

  Responsable de l'interaction avec MLFlow pour le suivi des modèles.

6.3. IMPLEMENTATION DE L'API

6.3.1. Chargement du modèle

Le chargement du modèle SARIMAX dans l'API est une étape cruciale. L'API se connecte au serveur MLFlow pour accéder au registre des modèles, récupérant la version la plus récente et stable. Le modèle est ensuite chargé en mémoire, prêt à traiter les requêtes de prédiction avec une performance optimale.

6.3.2. Exemple de requête

- **Exemple de requête pour une prédiction**

```python
import requests

# URL de l'API
url = "http://localhost:5100/predict"

# Jeton d'authentification (reçu lors de la connexion)
headers = {
    'x-access-tokens': 'votre_token_d_authentification_ici',
    'Content-Type': 'application/json'
}

# Données à prédire (exemple de données au format JSON)
data = [
    {"date": "2024-03-19 00:00:00", "ratio_debout": 0.65},
    {"date": "2024-03-19 00:15:00", "ratio_debout": 0.68},
    {"date": "2024-03-19 00:30:00", "ratio_debout": 0.70},]
    # Ajoutez d'autres points de données si nécessaire
```

```
# Envoyer la requête POST à l'API pour obtenir les prédictions
response = requests.post(url, json={"data": data}, headers=headers)

# Afficher la réponse de l'API
if response.status_code == 200:
    predictions = response.json()
    print("Prédictions :")
    print(predictions)
else:
    print(f"Erreur lors de la prédiction : {response.status_code}")
    print(response.json())
```

7. Integration de l'API SARIMAX dans une Application

7.1. Documentation de l'API

Pour intégrer efficacement l'API SARIMAX dans une application, il est essentiel de comprendre la documentation fournie. Celle-ci décrit les endpoints disponibles, les paramètres requis, et les formats de réponse attendus. Une bonne maîtrise de ces aspects garantit une utilisation correcte de l'API.

La documentation de l'API SARIMAX, créée avec l'aide de Flasgger pour générer une interface Swagger, permet une exploration interactive des fonctionnalités de l'API. Cela inclut des informations sur l'authentification, l'entraînement du modèle, les prédictions, et le suivi des métriques telles que le RMSE (Root Mean Square Error).

7.2. Utilisation

On commence par l'authentification de l'utilisateur. Le processus nécessite l'envoi des informations d'identification (nom d'utilisateur et mot de passe) via l'endpoint `/login`. Si l'authentification est réussie, un token JWT est renvoyé et doit être inclus dans les en-têtes de toutes les requêtes suivantes.

7.3. Implementation des appels API

7.3.1. Authentification

- **Coté Font-end**

  Ce code définit une fonction authentication_form() qui crée un formulaire d'authentification avec Streamlit, une bibliothèque utilisée pour créer des interfaces utilisateur en Python. Le formulaire comprend des champs de saisie pour le nom d'utilisateur et le mot de passe, ce dernier étant masqué pour plus de sécurité. Lorsqu'un utilisateur clique sur le bouton "Se connecter", les informations saisies sont envoyées via une requête POST à un serveur d'authentification spécifié par une URL. Si l'authentification réussit (indiquée par un code de statut HTTP 200), le serveur renvoie un jeton (token) qui est stocké dans l'état de session de l'application pour les futures interactions. En cas d'échec, un message d'erreur est affiché.

```
```python
def authentication_form():
    # Affiche le titre du formulaire d'authentification
    st.title("Authentification")
```
```

```
    # Champ de saisie pour le nom d'utilisateur
    username = st.text_input("Nom d'utilisateur")

    # Champ de saisie pour le mot de passe avec masquage des caractères
    password = st.text_input("Mot de passe", type="password")

    # Condition vérifiant si le bouton "Se connecter" est cliqué
    if st.button("Se connecter"):
        # URL du serveur d'authentification
        url_login = 'http://xxxxx:5xxx/login'

        # Données d'authentification à envoyer au serveur (nom d'utilisateur
        #et mot de  passe)
        auth_data = {'username': username, 'password': password}


        # Envoie d'une requête POST au serveur avec les données d'authentification
        response_login = requests.post(url_login, json=auth_data)

        # Vérification si la réponse du serveur indique une authentification réussie
        if response_login.status_code == 200:
            # Récupération du token d'authentification depuis la réponse du serveur
            token = response_login.json()['token']

            # Affiche un message de succès
            st.success("Authentification réussie")

            # Stocke le token dans l'état de session de l'application
            st.session_state.token = token
        else:
            # Affiche un message d'erreur en cas d'échec d'authentification
            st.error(f"Erreur d'authentification: {response_login.status_code}")
```

- **Coté Back-end**

  Ce code implémente une application Flask simple qui gère l'authentification utilisateur et la protection d'une route à l'aide de jetons JWT (JSON Web Tokens). La base de données SQLite est utilisée pour stocker les informations des utilisateurs, y compris un hachage sécurisé de leur mot de passe. Lorsqu'un utilisateur s'authentifie via la route `/login`, un jeton JWT est généré et renvoyé, ce qui permet à l'utilisateur d'accéder à la route protégée `/predict`. Cette route protégée, accessible uniquement avec un jeton valide, permet d'effectuer une prédiction basée sur les données d'entrée fournies, bien que dans cet exemple, la prédiction soit simulée par une simple chaîne de texte.

```
from flask import Flask, request, jsonify, g
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import generate_password_hash, check_password_hash
import jwt
import datetime
from functools import wraps

app = Flask(__name__)

# Configuration de la base de données SQLite
app.config['SECRET_KEY'] = 'votre_cle_secrete'
```

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
db = SQLAlchemy(app)
# Modèle d'utilisateur
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), unique=True, nullable=False)
    password_hash = db.Column(db.String(128), nullable=False)
    token = db.Column(db.String(500), nullable=True)

# Création de la base de données
with app.app_context():
    db.create_all()

# Fonction pour vérifier le jeton
def token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = request.headers.get('x-access-tokens')
        if not token:
            return jsonify({'message': 'Token manquant!'}), 401

        try:
            data = jwt.decode(token, app.config['SECRET_KEY'], algorithms=["HS256"])
            current_user = User.query.filter_by(username=data['username']).first()
        except:
            return jsonify({'message': 'Token invalide!'}), 401

        return f(current_user, *args, **kwargs)
    return decorated

# Route pour l'authentification
@app.route('/login', methods=['POST'])
def login():
    auth_data = request.json

    if not auth_data or not auth_data['username'] or not auth_data['password']:
        return jsonify({'message': 'Nom d\'utilisateur ou mot de passe manquant!'}), 400

    user = User.query.filter_by(username=auth_data['username']).first()

    if not user or not check_password_hash(user.password_hash, auth_data['password']):
        return jsonify({'message': 'Authentification échouée!'}), 401

    token = jwt.encode({
        'username': user.username,
        'exp': datetime.datetime.utcnow() + datetime.timedelta(minutes=30)
    }, app.config['SECRET_KEY'], algorithm="HS256")

    user.token = token
    db.session.commit()

    return jsonify({'token': token})

# Route protégée pour la prédiction
@app.route('/predict', methods=['POST'])
@token_required
def predict(current_user):
    # Exemple simple de prédiction - Vous pouvez remplacer cela par un modèle de ML
    data = request.json
```

```
    input_data = data.get('input_data')

    if not input_data:
        return jsonify({'message': 'Données d\'entrée manquantes!'}), 400

    # Logique de prédiction ici (c'est juste un exemple fictif)
    prediction = f"Prédiction pour {input_data}"

    return jsonify({'user': current_user.username, 'prediction': prediction})

# Exécution de l'application Flask
if __name__ == '__main__':
    app.run(debug=True)
```

7.3.2. Entraînement du Modèle

- **Coté Font-end**

  Cette fonction `train` est conçue pour envoyer des données à une API afin de lancer un processus d'entraînement de modèle de machine learning. Le DataFrame `df` fourni est converti en une liste de dictionnaires, où chaque ligne devient un dictionnaire de données. Un jeton d'accès est récupéré depuis l'état de session pour authentifier la requête HTTP POST envoyée à l'URL spécifiée. Si l'entraînement est réussi (indiqué par un code de réponse 200), un message de succès est affiché et les prédictions retournées par l'API sont affichées sous forme de tableau interactif. En cas d'échec, un message d'erreur est affiché.

```
```python
def train(df):
    # Préparation des données pour l'entraînement
    # Conversion du DataFrame en une liste de dictionnaires où chaque ligne devient
    #un dictionnaire
    data = df.to_dict(orient='records')

    # Préparation des en-têtes pour la requête HTTP
    # Le token d'accès est récupéré depuis l'état de session pour l'authentification
    headers = {
        'x-access-tokens': st.session_state.token,  # Token d'accès pour l'authentification
        'Content-Type': 'application/json'  # Type de contenu spécifié comme JSON
    }

    # URL de l'API pour lancer l'entraînement
    url_train = 'http://xxxxx:5xxx/train'

    # Envoie d'une requête POST à l'API avec les données d'entraînement
    # et les en-têtes d'authentification
    response_train = requests.post(url_train, headers=headers, json=data)

    # Vérification si la réponse du serveur indique que l'entraînement s'est bien déroulé
    if response_train.status_code == 200:
        # Affiche un message de succès si l'entraînement est réussi
        st.success("Entraînement réussi avec succès !")

        # Gestion des prédictions renvoyées par l'API après l'entraînement
        data_response = response_train.json()

        # Extraction des prédictions depuis la réponse, si elles existent
        predictions_json = data_response.get('predictions', None)
```
```

```
    if predictions_json:
        # Conversion des prédictions JSON en DataFrame pandas
        predictions_df = pd.read_json(predictions_json)

        # Affiche les prédictions sous forme de tableau interactif
        st.dataframe(predictions_df)
    else:
        # Affiche un message d'erreur si l'entraînement échoue
        st.error(f"Erreur lors de l'entraînement: {response_train.status_code}")```
```

- **Coté Back-end**

  Cette fonction `train_model` entraîne un modèle SARIMAX pour prédire le ratio debout/couché à partir d'un ensemble de données fourni. Elle commence par transformer les données en DataFrame pandas, puis applique un nettoyage via un objet `DataCleaner`. Les données sont ensuite divisées en un ensemble d'entraînement (jusqu'à deux jours avant la date maximale) et un ensemble de validation (les deux derniers jours). Un modèle SARIMAX est intégré dans un pipeline avec le nettoyeur de données et est entraîné sur l'ensemble d'entraînement.

  Les prédictions sont effectuées sur l'ensemble de validation, et la performance du modèle est évaluée en calculant la racine carrée de l'erreur quadratique moyenne (RMSE) entre les valeurs réelles et prédites. Un graphique comparant les données d'entraînement, les vérités terrain, et les prédictions est généré et sauvegardé.

  Le modèle entraîné est ensuite enregistré dans MLflow, où différents paramètres et métriques sont également enregistrés. Le modèle est enfin enregistré sous le nom "SARIMAXModel". Les fichiers temporaires créés (graphique et données CSV) sont supprimés après enregistrement. Finalement, la fonction renvoie les prédictions sous forme de JSON, le RMSE de validation, et l'ID de la session MLflow. En cas d'erreur lors du processus, l'exception est capturée et un message d'erreur est logué.

```
```python
def train_model(data):
    # Convertir les données en DataFrame pandas
    df = pd.DataFrame(data)

    # Nettoyage des données à l'aide d'un objet DataCleaner
    cleaner = DataCleaner()
    df_cleaned = cleaner.transform(df)

    # Déterminer la date maximale dans les données nettoyées
    max_date = df_cleaned['date'].max()
    # Fixer une date limite pour séparer les données d'entraînement et de validation
    # (2 jours avant la date maximale)
    cutoff_date = max_date - pd.Timedelta(days=2)
    # Sélectionner les données d'entraînement (avant ou égales à la date limite)
    train_data = df_cleaned[df_cleaned['date'] <= cutoff_date]
    # Sélectionner les données de validation (après la date limite)
    validation_data = df_cleaned[df_cleaned['date'] > cutoff_date]
```
```

```
# Calculer le nombre de jours dans les données d'entraînement
  num_days_train = (train_data['date'].max() - train_data['date'].min()).days
  # Calculer le nombre de jours dans les données de validation
  num_days_test = (validation_data['date'].max() - validation_data['date'].min()).days
   # Initialiser le modèle SARIMAX
  sarimax_model = SARIMAXModel()
  # Créer un pipeline avec le nettoyeur de données et le modèle SARIMAX
  pipeline = Pipeline([('cleaner', cleaner), ('sarimax', sarimax_model)], verbose=True)

  # Entraîner le pipeline sur les données d'entraînement
  pipeline.fit(train_data)

  # Créer un modèle personnalisé basé sur le pipeline
  custom_model = CustomModel(pipeline)

  # Faire des prédictions sur les données de validation
  predictions = sarimax_model.predict(validation_data)
  # Extraire les valeurs réelles des données de validation
  actuals = validation_data.set_index('date')['ratio_debout']

  # Calculer la RMSE pour la validation
  validation_rmse = np.sqrt(mean_squared_error(actuals, predictions))

  # Tracer les résultats
  plt.figure(figsize=(14, 7))
  plt.plot(train_data['date'], train_data['ratio_debout'], color='blue', label='Entraînement (5 jours)')
  plt.plot(validation_data['date'], actuals, color='green', label='Vérité Terrain (jours 6 et 7)')
  plt.plot(validation_data['date'], predictions, color='yellow', label='Prédictions (jours 6 et 7)')
  plt.xlabel('Date et Heure')
  plt.ylabel('Ratio Debout (%)')
  plt.title('Prédictions SARIMAX du Ratio Debout')
  plt.legend()
  plt.grid(True)

  # Sauvegarder le graphique des prédictions
  plot_file = 'predictions_plot.png'
  plt.savefig(plot_file)
  plt.close()

  # Sauvegarder les données d'entraînement et de validation sous forme de fichiers CSV
  train_data_file = 'train_data.csv'
  validation_data_file = 'validation_data.csv'
  train_data.to_csv(train_data_file, index=False)
  validation_data.to_csv(validation_data_file, index=False)

  # Enregistrer le modèle, les paramètres, et les métriques dans MLflow
  with mlflow.start_run() as run:
      # Enregistrement du modèle dans MLflow
      mlflow.pyfunc.log_model(
          artifact_path="model",
          python_model=custom_model
      )

      # Enregistrement des hyperparamètres du modèle
      mlflow.log_param("order", sarimax_model.order)
      mlflow.log_param("seasonal_order", sarimax_model.seasonal_order)
      mlflow.log_param("num_data_points", len(df_cleaned))
      mlflow.log_param("num_days_train", num_days_train)
      mlflow.log_param("num_days_test", num_days_test + 1)
```

```
        # Enregistrement des métriques (RMSE) pour l'entraînement et la validation
        mlflow.log_metric("train_rmse", -sarimax_model.score(train_data, train_data['ratio_debout']))
        mlflow.log_metric("validation_rmse", validation_rmse)

        # Enregistrement des artefacts (fichiers graphiques et CSV)
        mlflow.log_artifact(plot_file)
        mlflow.log_artifact(train_data_file)
        mlflow.log_artifact(validation_data_file)

        # Enregistrement du modèle dans MLflow avec un nom spécifique
        run_id = run.info.run_id
        mlflow.register_model(
            model_uri=f"runs:/{run.info.run_id}/model",
            name="SARIMAXModel"
        )

    # Nettoyer les fichiers temporaires
    if os.path.exists(plot_file):
        os.remove(plot_file)
    if os.path.exists(train_data_file):
        os.remove(train_data_file)
    if os.path.exists(validation_data_file):
        os.remove(validation_data_file)

    # Essayer de préparer les données de prédictions en format JSON
    try:
        # Sélectionner les colonnes pertinentes et réinitialiser les index
        predictions_df = validation_data[["date", 'ratio_debout']]
        predictions_df.reset_index(inplace=True)
        print(predictions_df.columns)  # Cela montrera ['index', 'date', 'ratio_debout']
        # Renommer les colonnes pour une meilleure compréhension
        predictions_df.columns = ['Index', 'Date', 'Valeurs']
        # Convertir la colonne 'Date' en format datetime
        predictions_df['Date'] = pd.to_datetime(predictions_df['Date'])
        # Convertir le DataFrame en JSON
        predictions_json = predictions_df.to_json(orient='records', date_format='iso')

    # Gestion des erreurs possibles
    except Exception as e:
        logging.error(f"Error during training: {e}")

    # Retourner les prédictions au format JSON, la RMSE de validation, et l'ID du run MLflow
    return predictions_json, validation_rmse, run_id
```

7.3.3. Prédiction

- **Coté Font-end**

La fonction prediction est conçue pour envoyer des données à l'API afin d'obtenir des prédictions. Elle convertit un DataFrame filtré (filtered_df) en une liste de dictionnaires où chaque ligne du DataFrame devient un dictionnaire de données. Un jeton d'accès, récupéré depuis l'état de session, est utilisé pour authentifier la requête HTTP POST envoyée à l'URL spécifiée pour la prédiction. Si la requête est réussie (indiquée par un code de réponse 200), un message de succès est affiché et les prédictions retournées par l'API sont converties en un DataFrame pandas. Les prédictions sont ensuite affichées

sous forme de graphique linéaire, utilisant la colonne Date comme index. En cas d'échec, un message d'erreur est affiché.

```python
def prediction(filtered_df, ratio_df, median_df):
    # Préparation des données pour la prédiction
    # Convertit le DataFrame filtré en une liste de dictionnaires, où chaque ligne devient
    # un dictionnaire
    data = filtered_df.to_dict(orient='records')

    # Préparation des en-têtes pour la requête HTTP
    # Le token d'accès est récupéré depuis l'état de session pour l'authentification
    headers = {
        'x-access-tokens': st.session_state.token,  # Token d'accès pour l'authentification
        'Content-Type': 'application/json'  # Type de contenu spécifié comme JSON
    }

    # URL de l'API pour la prédiction
    url_predict = 'http://xxxxx:5xxx/predict'

    # Envoie d'une requête POST à l'API avec les données à prédire et
    # les en-têtes d'authentification
    response_predict = requests.post(url_predict, headers=headers, json=data)

    # Vérification si la réponse du serveur indique que la prédiction s'est bien déroulée
    if response_predict.status_code == 200:
        # Affiche un message de succès si la prédiction est réussie
        st.success("Requête de prédiction réussie avec succès !")

        # Récupération des données de prédiction depuis la réponse du serveur
        response_data = response_predict.json()

        # Conversion des prédictions JSON en DataFrame pandas
        df_response = pd.DataFrame(json.loads(response_data['predictions']))

        # Affichage des prédictions sous forme de graphique linéaire,
        # en utilisant la colonne 'Date' comme index
        st.line_chart(df_response.set_index('Date')['Valeurs'])
    else:
        # Affiche un message d'erreur si la requête de prédiction échoue
        st.error(f"Erreur lors de la requête de prédiction: {response_predict.status_code}")
```

- **Coté Back-end**

  La fonction `predict_model` est conçue pour effectuer des prédictions sur un ensemble de données en utilisant un modèle SARIMAX précédemment enregistré dans MLflow. Les données fournies sont d'abord transformées en un DataFrame pandas, puis nettoyées à l'aide d'un objet `DataCleaner`. Le modèle SARIMAX est récupéré depuis le registre de modèles MLflow, spécifiquement la version en production du modèle nommé "SARIMAXModel".

  Une session MLflow est ouverte pour traquer les prédictions. Le modèle chargé est utilisé pour prédire les valeurs sur les données nettoyées. Les prédictions sont ensuite organisées dans un DataFrame, formatées avec des colonnes "Date" et "Valeurs", et converties en JSON pour un retour structuré.

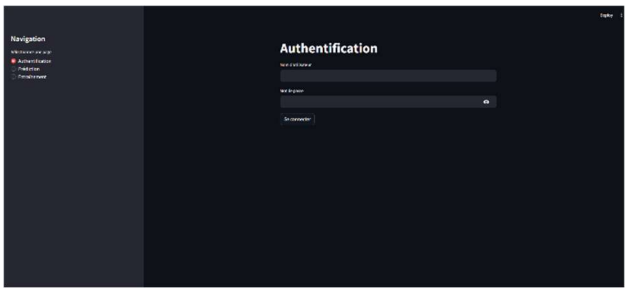
La fonction logue également le nombre de points de données utilisés dans la prédiction au sein de MLflow.

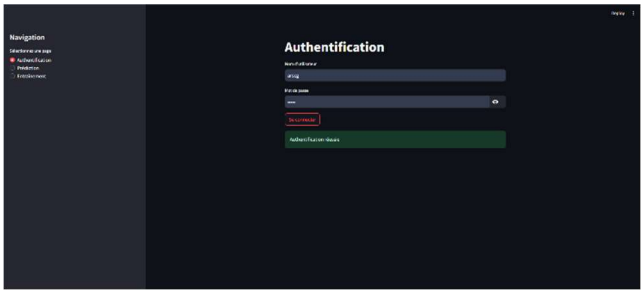
Enfin, la fonction retourne le JSON des prédictions, le nom du modèle utilisé, et la version du modèle SARIMAX en production. Cette structure permet de maintenir un suivi précis des modèles et des prédictions réalisées, tout en utilisant les capacités de suivi et d'enregistrement de MLflow.

```python
def predict_model(data):
    # Convertir les données en DataFrame pandas
    df = pd.DataFrame(data)

    # Nettoyage des données à l'aide d'un objet DataCleaner
    cleaner = DataCleaner()
    df_cleaned = cleaner.transform(df)

    # Nom du modèle à charger depuis MLflow
    model_name = 'SARIMAXModel'
    # Initialisation du client MLflow pour interagir avec le système de gestion des modèles
    client = mlflow.tracking.MlflowClient()
    # Chemin du modèle en production
    logged_model = f"models:/{model_name}/Production"

    # Charger le modèle SARIMAX en production depuis MLflow
    model = mlflow.pyfunc.load_model(logged_model)

    # Démarrer une nouvelle exécution dans MLflow pour enregistrer les paramètres
    # et les métriques
    with mlflow.start_run() as run:
        # Récupérer la version la plus récente du modèle en production
        model_version = client.get_latest_versions(model_name, stages=["Production"])[0].version
        # Faire des prédictions sur les données nettoyées
        predictions = model.predict(df_cleaned)

        # Convertir les prédictions en DataFrame avec une colonne nommée 'Valeurs'
        predictions_df = predictions.to_frame(name='Valeurs')
        # Réinitialiser les index du DataFrame
        predictions_df.reset_index(inplace=True)
        # Renommer les colonnes pour une meilleure compréhension
        predictions_df.columns = ['Date', 'Valeurs']
        # Convertir la colonne 'Date' en format datetime
        predictions_df['Date'] = pd.to_datetime(predictions_df['Date'])
        # Convertir le DataFrame en format JSON avec les dates au format ISO
        predictions_json = predictions_df.to_json(orient='records', date_format='iso')

        # Enregistrer le nombre de points de données utilisés pour la prédiction dans MLflow
        mlflow.log_param("num_data_points", len(df_cleaned))

    # Retourner les prédictions en JSON, le nom du modèle et la version du modèle utilisée
    return predictions_json, model_name, model_version
```

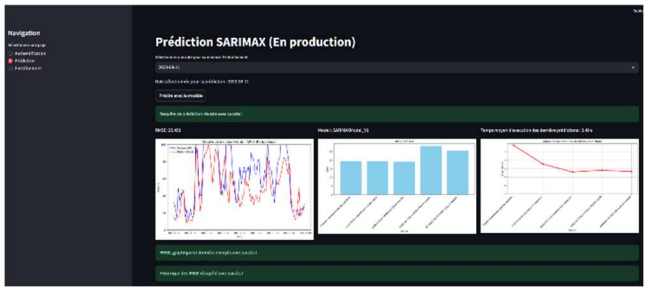
7.4. CAPTURE D'ECRAN DE L'APPLICATION

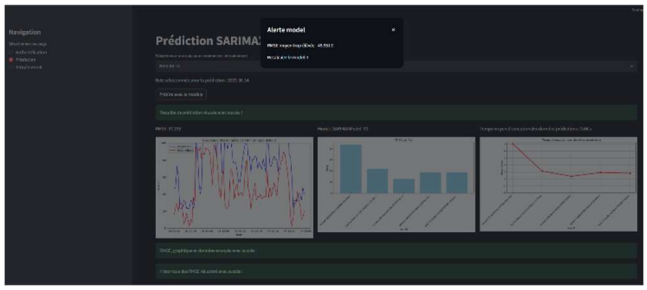
7.4.1. Authentification



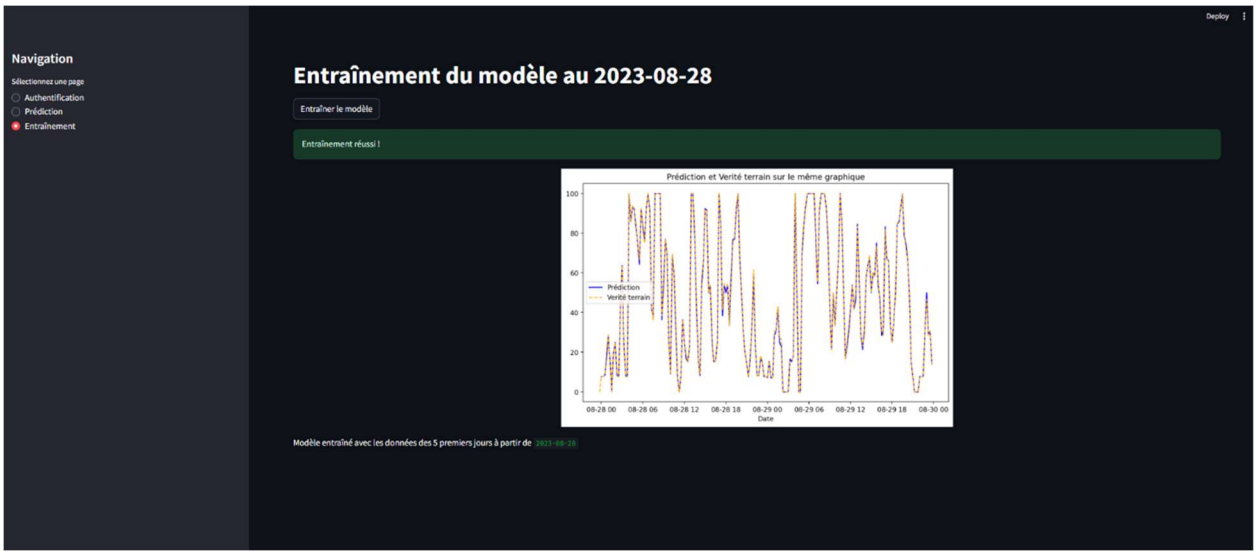


7.4.2. Prédiction





7.4.3. Entrainement



8. TESTS AUTOMATISES DU MODELE SARIMAX

8.1. REGLES DE VALIDATION

Les tests automatisés jouent un rôle crucial dans la validation des fonctionnalités de l'API. Ils permettent de s'assurer que chaque composant fonctionne correctement et que l'API est prête à être déployée en production.

- **Authentification :**

  Vérification que l'API accepte des identifiants valides et renvoie un token JWT, ou rejette les tentatives avec des identifiants incorrects.

- **Entraînement du Modèle :**

  S'assurer que l'API entraîne correctement le modèle SARIMAX avec des données valides, en renvoyant les métriques appropriées comme le RMSE.

- **Prédiction :**

  Validation de la capacité de l'API à fournir des prédictions précises lorsque des données valides sont soumises.

- **Gestion des Erreurs :**

  Vérification que l'API gère correctement les erreurs, en renvoyant des messages d'erreur clairs et appropriés.

8.2. FRAMEWORKS DE TEST UTILISES

- **Pytest :**

  Utilisé pour automatiser les tests unitaires et d'intégration, Pytest est choisi pour sa simplicité et sa capacité à s'intégrer avec Flask, permettant de tester efficacement les endpoints de l'API sans avoir besoin de déployer un serveur réel.

1. Exemple de tests automatisés

Ce script de tests utilise `pytest` pour valider la fonctionnalité d'un modèle SARIMAX appliqué à un jeu de données temporelles. Le test `test_model_prediction` vérifie que le modèle SARIMAX peut prédire 48 valeurs futures après avoir été entraîné sur un ensemble de 100 valeurs aléatoires. Un DataFrame simulé est créé avec 100 observations horaires commençant le 1er janvier 2023. Le modèle SARIMAX est ensuite configuré avec des paramètres spécifiques et ajusté aux données pour effectuer des prédictions. Le test vérifie que le nombre de prédictions générées est correct (48). Une fixture nommée `setup_data` est utilisée pour fournir un DataFrame standardisé à d'autres tests, comme dans `test_data_preparation`, qui vérifie que le DataFrame contient bien 100 lignes, assurant ainsi que la préparation des données est correcte.

```python
import pytest
import pandas as pd
import numpy as np
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Test pour vérifier la prédiction du modèle SARIMAX
def test_model_prediction():
```

```
    # Création d'un DataFrame avec des données simulées
    data = pd.DataFrame({
        'timestamp': pd.date_range(start='1/1/2023', periods=100, freq='H'),  # 100 heures consécutives à partir du 1er janvier 2023
        'value': np.random.rand(100)  # Génération de 100 valeurs aléatoires
    })

    # Configuration du modèle SARIMAX avec des paramètres spécifiques
    model = SARIMAX(data['value'], order=(1, 1, 1), seasonal_order=(1, 1, 1, 24))

    # Ajustement du modèle aux données
    model_fit = model.fit(disp=False)

    # Prédiction des 48 prochaines valeurs
    predictions = model_fit.forecast(steps=48)
    # Vérifie que le nombre de prédictions est bien de 48
    assert len(predictions) == 48

# Fixture pour fournir un jeu de données standardisé aux tests
@pytest.fixture
def setup_data():
    # Retourne un DataFrame avec des données simulées
    return pd.DataFrame({
        'timestamp': pd.date_range(start='1/1/2023', periods=100, freq='H'),  # 100 heures consécutives à partir du 1er janvier 2023
        'value': np.random.rand(100)  # Génération de 100 valeurs aléatoires
    })

# Test pour vérifier la préparation des données
def test_data_preparation(setup_data):
    # Vérifie que le DataFrame fourni par la fixture contient bien 100 lignes
    assert len(setup_data) == 100
```

9. CONCLUSION

9.1. RESUME DES REALISATIONS

Ce projet a permis de développer une API REST pour un modèle SARIMAX, de l'intégrer dans une application web, de monitorer les performances du modèle, et de mettre en place des tests automatisés pour garantir sa fiabilité. Une chaîne de livraison continue a également été implémentée pour assurer la maintenance et les mises à jour régulières du modèle.

9.2. LEÇONS APPRISES

Parmi les leçons clés retenues, l'importance de la surveillance continue des modèles d'IA a été particulièrement notable. L'automatisation des tests et des déploiements s'est avérée essentielle pour maintenir un haut niveau de qualité tout au long du cycle de vie du projet.

9.3. PERSPECTIVES FUTURES

Les perspectives futures incluent l'amélioration de l'interface utilisateur de l'application pour une meilleure expérience, l'ajout de nouvelles fonctionnalités au modèle pour en augmenter la performance, et l'optimisation des processus de prédiction pour réduire les temps de réponse.

ANNEXES

I. Guide d'Installation et de Configuration d'un Environnement Virtuel sous Anaconda

Voici une procédure détaillée pour créer un environnement virtuel sous Anaconda sur Windows, installer les dépendances nécessaires à partir des fichiers fournis, et exécuter le projet.

1. Installation de Anaconda

Si vous n'avez pas encore Anaconda installé sur votre machine, vous pouvez le télécharger et l'installer depuis le site officiel.

2. Création de l'environnement virtuel

Une fois Anaconda installé, ouvrez l'Anaconda Prompt (vous pouvez le trouver en le recherchant dans le menu Démarrer).

a. Ouvrir l'Anaconda Prompt :

Cliquez sur l'icône Anaconda dans le menu Démarrer pour ouvrir l'Anaconda Prompt.

b. Créer un nouvel environnement :

Utilisez la commande suivante pour créer un nouvel environnement Python. Remplacez `myenv` par le nom de votre choix pour l'environnement.

```bash
conda create --name myenv python=3.8
```

Cette commande créera un environnement nommé `myenv` avec Python 3.8.

c. Activer l'environnement :

```bash
conda activate myenv
```

3. Installation des dépendances

Après avoir activé votre environnement virtuel, installez les bibliothèques nécessaires à partir du fichier `requirements.txt` que vous allez générer à partir des fichiers Python fournis.

a. Générer le fichier `requirements.txt` :

- Créez un fichier nommé `requirements.txt` dans le même répertoire que vos fichiers `.py`.

- Ajoutez-y les bibliothèques nécessaires, identifiées à partir des imports dans vos fichiers `.py`. Par exemple :

```
flask
flasgger
```

```
pandas
numpy
matplotlib
scikit-learn
mlflow
streamlit
requests
pytest
```

b. Installer les dépendances :

```bash
pip install -r requirements.txt
```

4. Exécution du projet

a. Lancer l'application Flask :

Vous pouvez lancer l'application Flask en exécutant le fichier `appjwt_refactoring.py`. Assurez-vous que l'application MLflow est en cours d'exécution si nécessaire.

```bash
python appjwt_refactoring.py
```

b. Exécuter les tests :

Utilisez `pytest` pour exécuter les tests définis dans `test_app.py`.

```bash
pytest test_app.py
```

c. Utiliser l'interface Streamlit :

Vous pouvez également lancer l'application Streamlit en exécutant le fichier `app_courbe.py`.

```bash
streamlit run app_courbe.py
```

5. Désactiver l'environnement virtuel

Lorsque vous avez terminé, vous pouvez désactiver l'environnement avec la commande :

```bash
conda deactivate
```

Cette procédure vous permettra de configurer correctement l'environnement pour exécuter et tester le projet sous Windows avec Anaconda.