Université Chouaib Doukkali
Ecole Nationale des Sciences Appliquées d'El Jadida
Département Télécommunications, Réseaux et Informatique

PROJET
Filière : 2ITE
Niveau : 3ème Année

Sujet :

Mise en place d'un pipeline d'entrainement continu en utilisant Apache Airflow et Streamlit

Réalisé Par :
Arbaoui Salma,
Fouguir Akram,
Moughtanim Mouhib

Encadré Par :
Prof. FAHD KALLOUBI

Année Universitaire : 2022/2023

Table des matières

Liste des figures

I. Objectif

L'objectif de ce projet est de mettre en place un pipeline d'entraînement continu en utilisant Apache Airflow et Streamlit ce projet couvre plusieurs aspects liés à la collecte de données météorologiques via l'API **OpenWeatherMap** à l'application des techniques de prétraitement sur les données collectées (nettoyage, gestion des valeurs manquantes, conversion de types).
à l'entraînement de modèles prédictifs sur ces données, à la mise en œuvre d'une application web pour consommer le modèle, et à l'automatisation du processus à l'aide d'Apache Airflow. L'objectif global de ce projet est d'introduire et de mettre en pratique plusieurs concepts et compétences.

II. Prérequis pour le projet - Installation et Configuration

Ce projet nécessite l'installation et la configuration préalable de l'outil Apache Airflow et l'environnement docker. Suivez attentivement ces étapes pour garantir un déroulement sans problème du projet.

1. Environnement de Développement

Docker :

- ✓ Téléchargez et installez Docker en suivant les instructions spécifiques à votre système d'exploitation : lien vers Docker https://docs.docker.com/engine/install/
- ✓ Vérifiez l'installation avec la commande : docker –version

Docker Compose :

- ✓ Téléchargez et installez Docker Compose en suivant les instructions spécifiques à votre système d'exploitation .
- ✓ Vérifiez l'installation avec la commande : docker-compose –version

2. Installation et Configuration de Apache Airflow

Note : Nous pouvons personnaliser davantage Airflow selon nos besoins en utilisant le fichier Dockerfile ci-dessous.

```
FROM apache/airflow:2.6.1
ADD requirements.txt .
RUN pip install -r requirements.txt
```

Dans le répertoire qui contient le fichier Dockerfile, créez un fichier nommé .env et placez-y les lignes suivantes :

```
AIRFLOW_UID=5000
AIRFLOW_GID=0
```

Toujours dans le répertoire qui contient le fichier Dockerfile, créez un fichier nommé requirements.txt et placez-y les lignes suivantes :

```
scikit-learn
pandas
xgboost
streamlit
imbalanced-learn
matplotlib
seaborn
plotly
```

Ensuite, créez trois repertoires toujours dans le meme répértoire :

./dags, ./logs, ./plugins, et ./config.

Puis créez un fichier docker-compose.yml pour définir les services Docker nécessaires pour Airflow, on peut télécharger le fichier docker-compose.yml à partir du lien suivant :

https://airflow.apache.org/docs/apache-airflow/2.7.3/docker-compose.yaml

Mettez à jour le fichier docker-compose.yaml en modifiant ce qui suit :

```
image: ${AIRFLOW_IMAGE_NAME:-apache/airflow:2.6.1}
# build: .
```

```
# image: ${AIRFLOW_IMAGE_NAME:-apache/airflow:2.6.1}
build: .
```

Exécutez la commande suivante pour démarrer Apache Airflow dans Docker.

```
docker-compose up -d
```

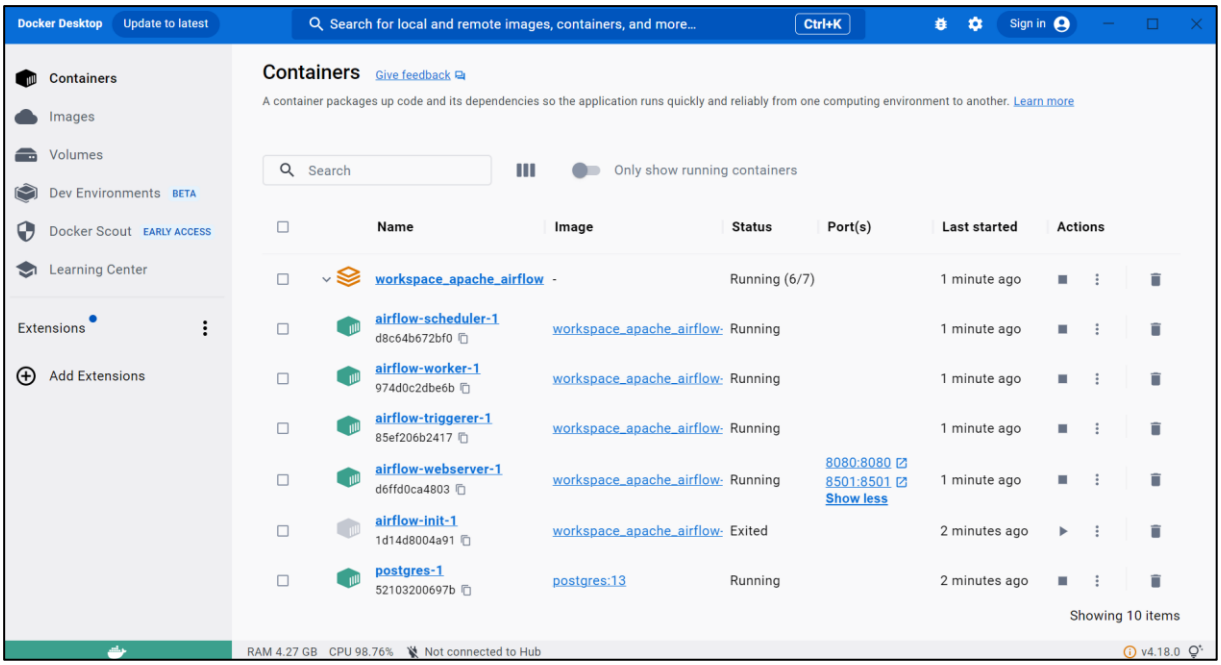


Figure 1:interface docker

Ouvrez votre navigateur et accédez à **http://localhost:8080** pour accéder à l'interface web d'Airflow.

III. Construction du dataset

1. Description des données disponibles via l'API.

L'API **OpenWeatherMap** fournit un ensemble complet de données météorologiques, y compris des informations telles que la température, la pression atmosphérique, l'humidité, la vitesse du vent, les conditions météorologiques actuelles.
Ces données sont disponibles pour des emplacements spécifiques et sont mises à jour régulièrement.Voici une analyse rapide des champs clés :

"main" : Contient des informations sur la température, la pression, l'humidité, etc.
"temp" : Température actuelle.
"feels_like" : Température ressentie.
"temp_min" et "temp_max" : Températures minimale et maximale.
"humidity" : Humidité en pourcentage.
"weather" : Informations sur les conditions météorologiques.
"id" : Identifiant de la condition météorologique.
"main" : Catégorie principale (ex. "Rain").
"description" : Description détaillée (ex. "light rain").
"clouds" : Pourcentage de couverture nuageuse.
"wind" : Informations sur le vent.
"speed" : Vitesse du vent.

"deg" : Direction du vent en degrés.
"gust" : Rafales de vent.
"visibility" : Visibilité en mètres.
"pop" : Probabilité de précipitation.
"rain" : Quantité de pluie prévue sur 3 heures.
"sys" : Informations système.
"dt_txt" : Date et heure au format texte.

2. Méthode d'interrogation de l'API pour collecter les données.

La méthode d'interrogation de l'API OpenWeatherMap pour collecter les données implique l'envoi de requêtes HTTP à l'API, qui répondra avec les informations météorologiques demandées.

Voici un processus général pour interroger l'API et construire votre dataset :

a. Obtention d'une Clé API :

Allez sur le site d'OpenWeatherMap et créez un compte.

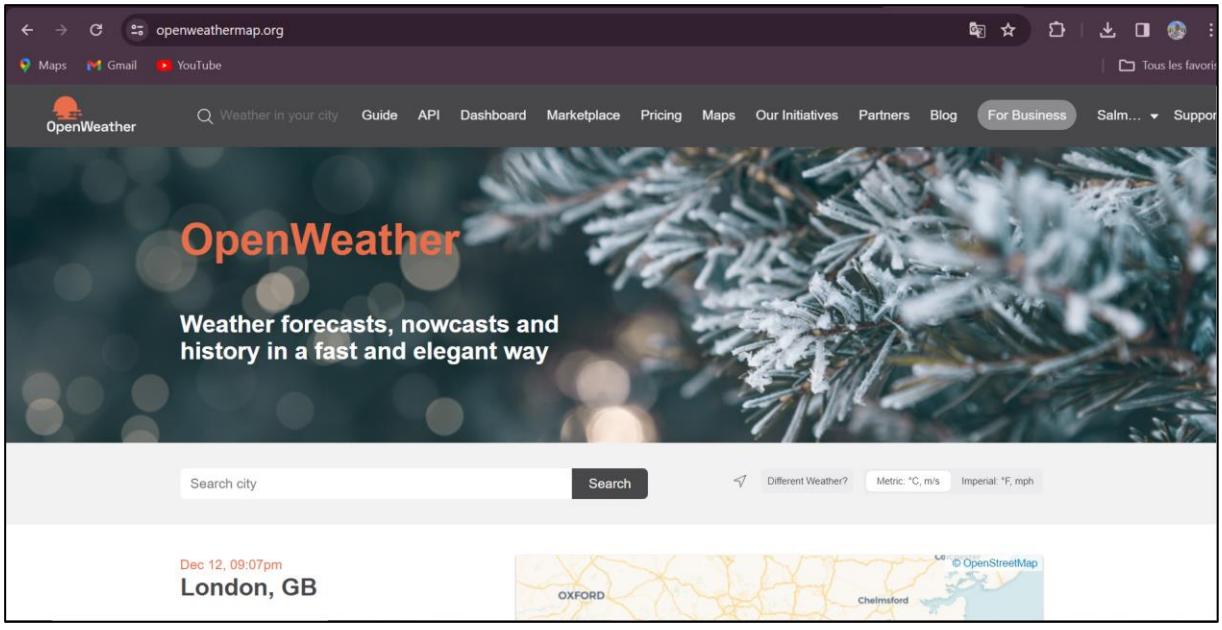


Figure 2:interface openweathermap

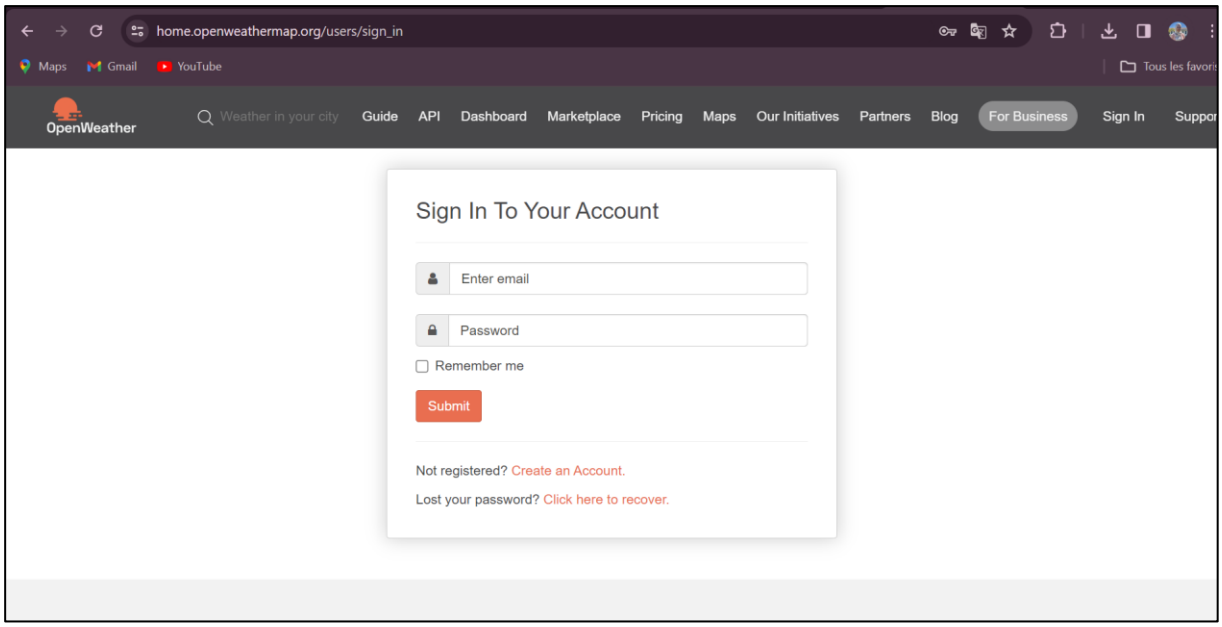


Figure 3:interface login openweathermap

Générez une clé API qui sera utilisée pour authentifier vos requêtes.

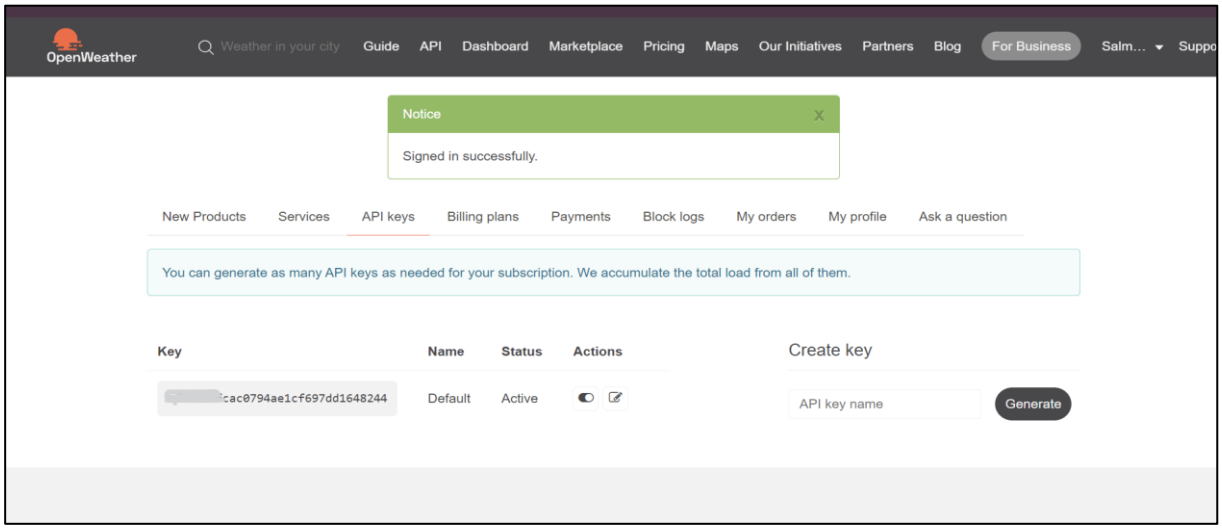


Figure 4:génération clé api

b. Construction de l'URL de Requête :

Utilisez la documentation OpenWeatherMap pour comprendre la structure des requêtes API. Construisez l'URL de requête en spécifiant les paramètres nécessaires (ville, pays, type de données, etc.).

c. Envoi de la Requête à l'API :

On va utiliser une bibliothèque HTTP **Requests** en Python pour envoyer la requête à l'API. Incluez votre clé API dans la requête pour l'authentification.

d. Réception et Traitement de la Réponse :

Vous recevez la réponse de l'API, généralement au format JSON.Vous devez parser la réponse pour extraire les données météorologiques pertinentes.

e. Stockage des Données dans un Dataset :

Créez un dataframe en utilisant la bibliothèque pandas en Python pour stocker les données. Ajoutez les données météorologiques extraites à ce dataframe.

Voici le code complet.

```
import requests
import pandas as pd
from datetime import datetime, timedelta
api_key = 'Api_key'
endpoint = 'https://api.openweathermap.org/data/2.5/forecast'
cities = [

 "Tokyo", "Mexico", "Séoul", "Jakarta", "New Delhi", "Le Caire"]
city_data_list = []
for city in cities:
    url = f'{endpoint}?q={city}&APPID={api_key}'
    response = requests.get(url)
    if response.status_code == 200:
        weather_data = response.json()
        # Check if 'list' key is present in the response
        if 'list' in weather_data:
            city_forecast = weather_data['list']

            for forecast in city_forecast:
                timestamp = forecast['dt']
                date = datetime.utcfromtimestamp(timestamp).strftime('%Y-%m-%d
%H:%M:%S')

                city_info = {
                    'City': city,
                    'Date': date,
                    'Temperature': forecast['main']['temp'],
                    'feels_like': forecast['main']['feels_like'],
                    'Temp_Min': forecast['main']['temp_min'],
                    'Temp_Max': forecast['main']['temp_max'],
                    'Pressure': forecast['main']['pressure'],
                    'Sea_Level': forecast['main']['sea_level'] if 'sea_level' in forecast['main']
else None,
                    'grnd_level': forecast['main']['grnd_level'],
                    'Humidity': forecast['main']['humidity'],
```

```
                'clouds all': forecast['clouds']['all'],
                'Description': forecast['weather'][0]['main'],
                'Wind Speed': forecast['wind']['speed'],
                'Wind deg': forecast['wind']['deg'],
                'Wind gust': forecast['wind']['gust'],
                'Country': weather_data['city']['country'],
                'Lon': weather_data['city']['coord']['lon'],
                'Lat': weather_data['city']['coord']['lat'],
            }
            city_data_list.append(city_info)
        else:
            print(f"No forecast data for {city}")
    else:
        print(f"Error {city}: {response.status_code}")

df = pd.DataFrame(city_data_list)
df.to_csv('weather_5.csv',index=False)
```

IV. Prétraitement des données

1. Exploration et analyse des données brutes.

Le processus de prétraitement commence par une exploration approfondie des données brutes. Utilisez le fichier CSV généré précédemment et créé un DataFrame data en utilisant Pandas. L'analyse exploratoire initiale inclut l'affichage des premières et dernières lignes du jeu de données, ainsi que des informations détaillées sur les types de données et les valeurs manquantes. Cette étape permet de comprendre la structure des données et d'identifier d'éventuels problèmes initiaux.
Importez les librairies necessaires.

```
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Chargement du jeu de données
data = pd.read_csv("Votre_fichier_CSV.csv")
data.head()
data.tail()
data.info()
```

Chargez les données à partir du fichier CSV dans un DataFrame Pandas appelé data.

La visualisation des données est une étape cruciale dans l'analyse exploratoire des données (EDA). Les visualisations permettent de comprendre les données, identification des Tendances et des Modèles ainsi la détection des Valeurs Aberrantes.
Essayer de faire différents visualisations.

```
plt.figure(figsize=(10,5))
sns.set_theme()
sns.countplot(x = 'Description',data = data,palette="ch:start=.2,rot=-.3")
plt.xlabel("weather",fontweight='bold',size=13)
plt.ylabel("Count",fontweight='bold',size=13)
plt.show()
```

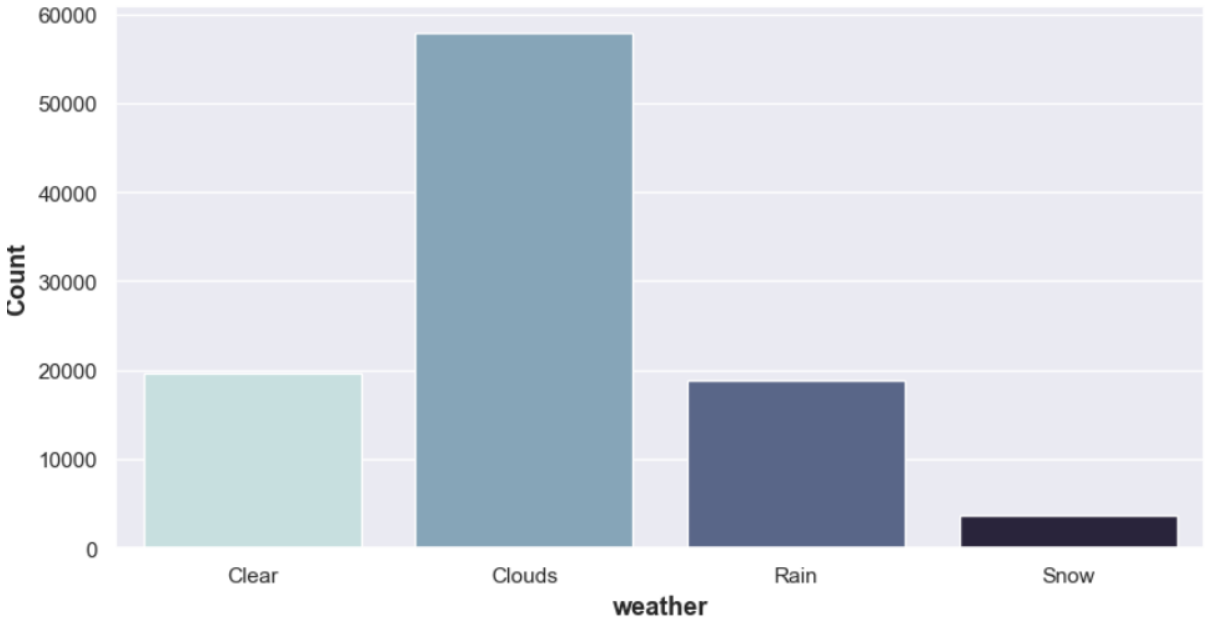


Figure 5:Nombre d'occurrences par type

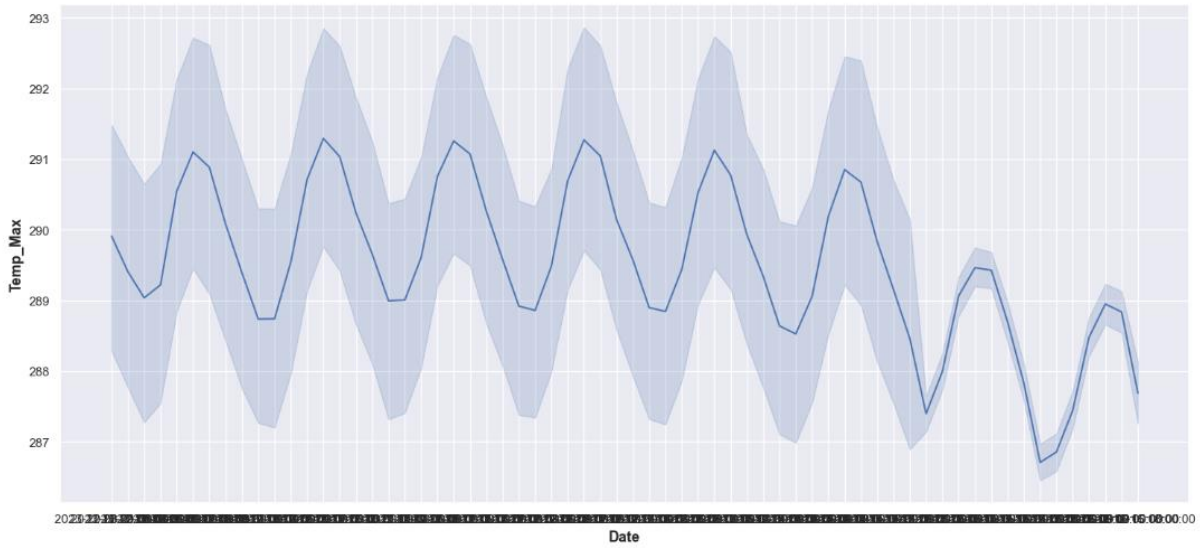
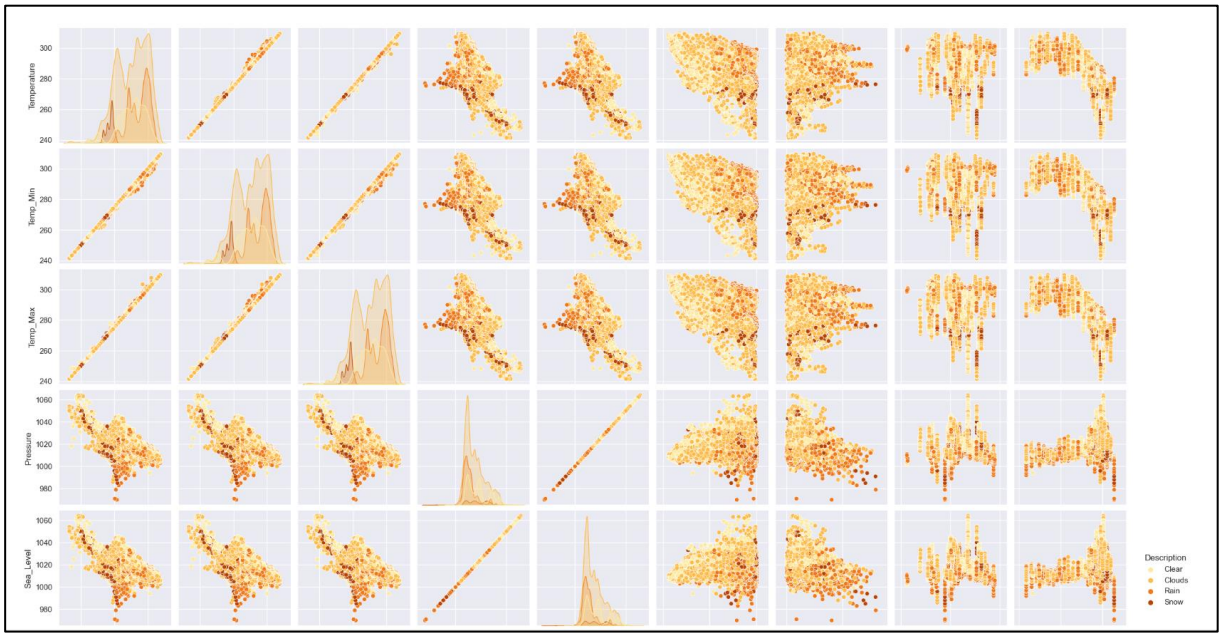


Figure 6:Variation de la Température Maximale au Fil du Temps

```
plt.figure(figsize=(14,8))
sns.pairplot(data.drop('Date',axis=1),hue='Description',palette="YlOrBr")
plt.show()
```



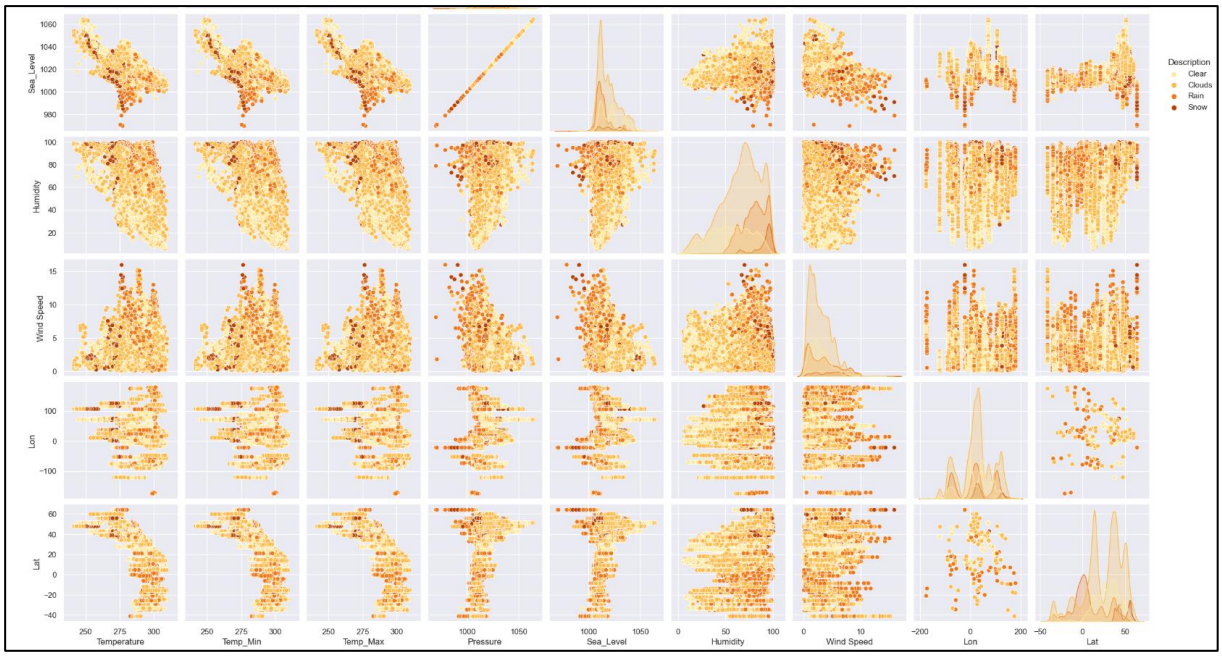


Figure 7:Pairplot avec différenciation par type

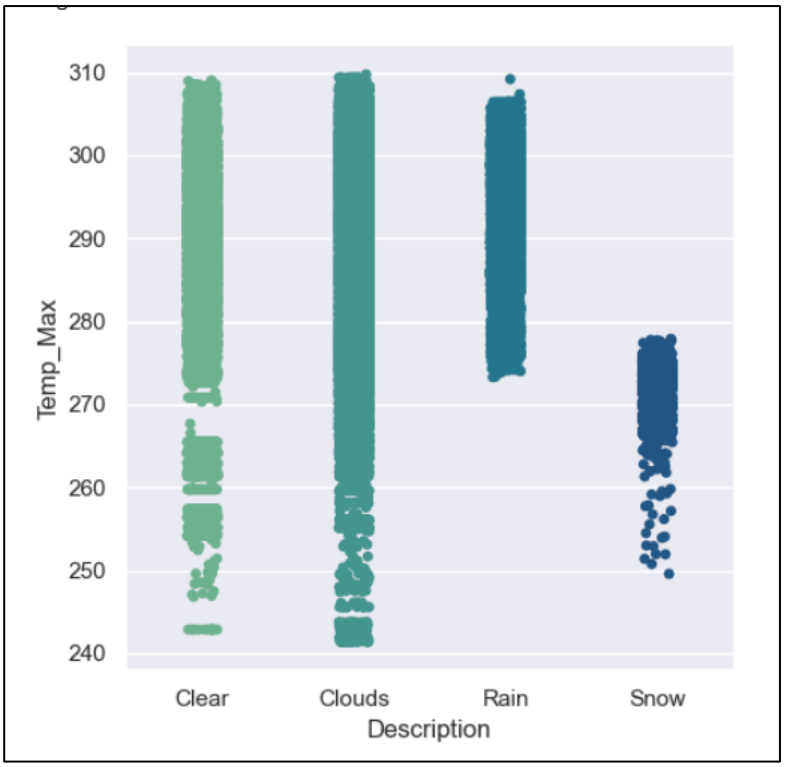


Figure 8:Température Maximale par Type

2. Traitement des valeurs manquantes et des données aberrantes.

La prochaine étape consiste à traiter les valeurs manquantes

```
data = data.dropna()
data['Date'] = pd.to_datetime(data['Date'])
data = data.drop(['City', 'Country'], axis=1)
```

Supprimez les lignes contenant des valeurs nulles et convertissez la colonne "Date" en format datetime pour faciliter la manipulation temporelle. Les colonnes inutiles telles que "City", et "Country" ont été supprimées.

Analyse de la distribution des descriptions :

```
data["Description"].value_counts()
```

Ce bout de code affiche le nombre des différentes catégories de la colonne cible.
Vous allez remarquer que les classes dans la colonne cible ne sont pas équilibrées, ce qui peut potentiellement biaiser les performances des modèles d'apprentissage automatique.

```
Description
Clouds    144774
Rain       45632
Clear      45439
Snow        9205
Name: count, dtype: int64
```

Figure 9:nombre des différentes catégories de la colonne cible

En suréchantillonnant les classes sous-représentées, on s'efforce de créer un ensemble de données équilibré pour une meilleure généralisation du modèle.
Voici un code qui fournit une fonction appelée equilibre_classes qui vise à équilibrer les classes d'un ensemble de données en suréchantillonnant les classes sous-représentées.

```
import pandas as pd
from sklearn.utils import resample

def equilibre_classes(df, label_col):
classes = df[label_col].unique()
max_samples = df[label_col].value_counts().max()

balanced_df = pd.DataFrame()

for classe in classes:
classe_df = df[df[label_col] == classe]
classe_df_resampled = resample(classe_df, replace=True, n_samples=max_samples,
random_state=42)
balanced_df = pd.concat([balanced_df, classe_df_resampled])

return balanced_df

data = equilibre_classes(data, 'Description')
```

3. Encodage des variables catégorielles.

Les variables catégorielles, comme la colonne "Description", ont été encodées en utilisant la technique d'encodage des étiquettes **Label Encoding** pour les convertir en valeurs numériques. Cela facilite l'utilisation de ces données dans les modèles d'apprentissage automatique qui nécessitent des entrées numériques.

```
From sklearn import preprocessing
def LABEL_ENCODING(c1):
label_encoder = preprocessing.LabelEncoder()
data[c1] = label_encoder.fit_transform(data[c1])
LABEL_ENCODING("Description")
```

4. Choix de la variable cible (par exemple, weather main).

La variable cible a été sélectionnée , dans notre cas, la colonne "Description". C'est la variable que nous chercherons à prédire à l'aide de modèles d'apprentissage automatique.
Le reste des colonnes du DataFrame a été considéré comme des variables indépendantes (X).

```
# Séparation des variables indépendantes (X) et de la variable cible (y)
x = data.drop('Description', axis=1)
y = data['Description']
```

V. Entrainement des modèles

Divisez la dataset en ensembles d'entraînement et de test.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 42)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

On va utiliser différents modèles .
Débutez par le feature scaling voici le code qui standardise les données en soustrayant la moyenne et en divisant par l'écart type, ce qui rend chaque caractéristique centrée autour de zéro et avec une variance égale à un.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
```

```
X_train = sc.fit_transform(X_train)

X_test = sc.transform(X_test)
```

La régression linéaire

```
from sklearn.linear_model import LogisticRegression
classifier1 = LogisticRegression(random_state = 42)
classifier1.fit(X_train, y_train)
y_pred = classifier1.predict(X_test)
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
sns.heatmap(cm,annot=True)
plt.show()
acc1 = accuracy_score(y_test, y_pred)
print(f"Accuracy score: {acc1}")
```

SVM

```
from sklearn.svm import SVC
classifier2 = SVC(kernel = 'linear', random_state = 42)
classifier2.fit(X_train, y_train)
y_pred = classifier2.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print(cm)
acc2 = accuracy_score(y_test, y_pred)
print(f"Accuracy score: {acc2}")
```

KNN

```
from sklearn.neighbors import KNeighborsClassifier
classifier3 = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 2)
classifier3.fit(X_train, y_train)
y_pred = classifier3.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print(cm)
acc3 = accuracy_score(y_test, y_pred) print(f"Accuracy score: {acc3}")
```

Naive bayes

```
from sklearn.naive_bayes import GaussianNB
classifier4 = GaussianNB()
classifier4.fit(X_train, y_train)
y_pred = classifier4.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print(cm)
acc4 = accuracy_score(y_test, y_pred)
print(f"Accuracy score : {acc4}")
```

Decision tree classification

```
from sklearn.tree import DecisionTreeClassifier
classifier5 = DecisionTreeClassifier(criterion = 'entropy', random_state = 42)
classifier5.fit(X_train, y_train)
y_pred = classifier5.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print(cm)
acc5 = accuracy_score(y_test, y_pred)
print(f"Accuracy score: {acc5}")
```

Random forest classifier

```
from sklearn.ensemble import RandomForestClassifier
forest= RandomForestClassifier(n_estimators =40, random_state = 42)
forest.fit(X_train,y_train)
RandomForestClassifier(n_estimators=40, random_state=42)
y_pred = forest.predict(X_test)
cm = confusion_matrix(y_test,y_pred)
from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred))
acc6 = forest.score(X_test,y_test)
print(acc6)
```

XGBoost

```
from xgboost import XGBClassifier
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
```

```
# Create and fit the XGBoost classifier
classifier6 = XGBClassifier()
classifier6.fit(X_train, y_train_encoded)
y_test_encoded = label_encoder.transform(y_test)

# Make predictions on the test set
y_pred = classifier6.predict(X_test)

# Calculate confusion matrix and accuracy score
cm = confusion_matrix(y_test_encoded, y_pred)
acc7 = accuracy_score(y_test_encoded, y_pred)

print("Confusion Matrix:")
print(cm)
print(f"Accuracy Score: {acc7}")
print(acc7)
```

Comparaison des performances des différents modèles.

```
mylist=[]
mylist2=[]
mylist.append(acc1)
mylist2.append("Logistic Regression")
mylist.append(acc2)
mylist2.append("SVM")
mylist.append(acc3)
mylist2.append("KNN")
mylist.append(acc4)
mylist2.append("Naive Bayes")
mylist.append(acc5)
mylist2.append("DTR")
mylist.append(acc6)
mylist2.append("RF")
mylist.append(acc7)
mylist2.append("XGBoost")
plt.rcParams['figure.figsize']=8,6
sns.set_style("darkgrid")
plt.figure(figsize=(22,8))
ax = sns.barplot(x=mylist2, y=mylist, palette = "mako", saturation =1.5)
plt.xlabel("Classification Models", fontsize = 20 )
plt.ylabel("Accuracy", fontsize = 20)
plt.title("Accuracy of different Classification Models", fontsize = 20)
plt.xticks(fontsize = 11, horizontalalignment = 'center', rotation = 8)
plt.yticks(fontsize = 13)
for p in ax.patches:
    width, height = p.get_width(), p.get_height()
    x, y = p.get_xy()
    ax.annotate(f'{height:.2%}', (x + width/2, y + height*1.02), ha='center',
```

```
        fontsize = 'x-large') plt.show()


from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
def label_encode(y_train, y_test):

 label_encoder = LabelEncoder()
    y_train_encoded = label_encoder.fit_transform(y_train
)
```

```
 y_test_encoded = label_encoder.transform(y_test)
    return y_train_encoded, y_test_encoded

def compare_models(X_train, y_train, X_test, y_test):
    y_train_encoded, y_test_encoded = label_encode(y_train, y_test)

    models = {
        "Logistic Regression": LogisticRegression(),
       "SVM": SVC(),
        "KNN": KNeighborsClassifier(),
        "Naive Bayes": GaussianNB(),
        "Decision Tree": DecisionTreeClassifier(),
        "Random Forest": RandomForestClassifier(n_estimators=40, random_state=0),
        "XGBoost": XGBClassifier()
    }

    results = {}

    for model_name, model in models.items():
        model.fit(X_train, y_train_encoded)
        y_pred = model.predict(X_test)
        accuracy = accuracy_score(y_test_encoded, y_pred)
        results[model_name] = accuracy

    best_model_name = max(results, key=results.get)
    best_accuracy = results[best_model_name]

    return best_model_name, best_accuracy, results

# Assuming X_train, y_train, X_test, y_test are your training and test data
best_model, best_accuracy, all_results = compare_models(X_train, y_train, X_test, y_test)
```

```
print("Results:")
for model, accuracy in all_results.items():
    print(f"{model}: {accuracy:.2%}")

print(f"Best Model: {best_model} with Accuracy: {best_accuracy:.2%}")
```

Le code effectue une comparaison des performances de différents modèles de classification, visualisez les résultats avec un graphique à barres.

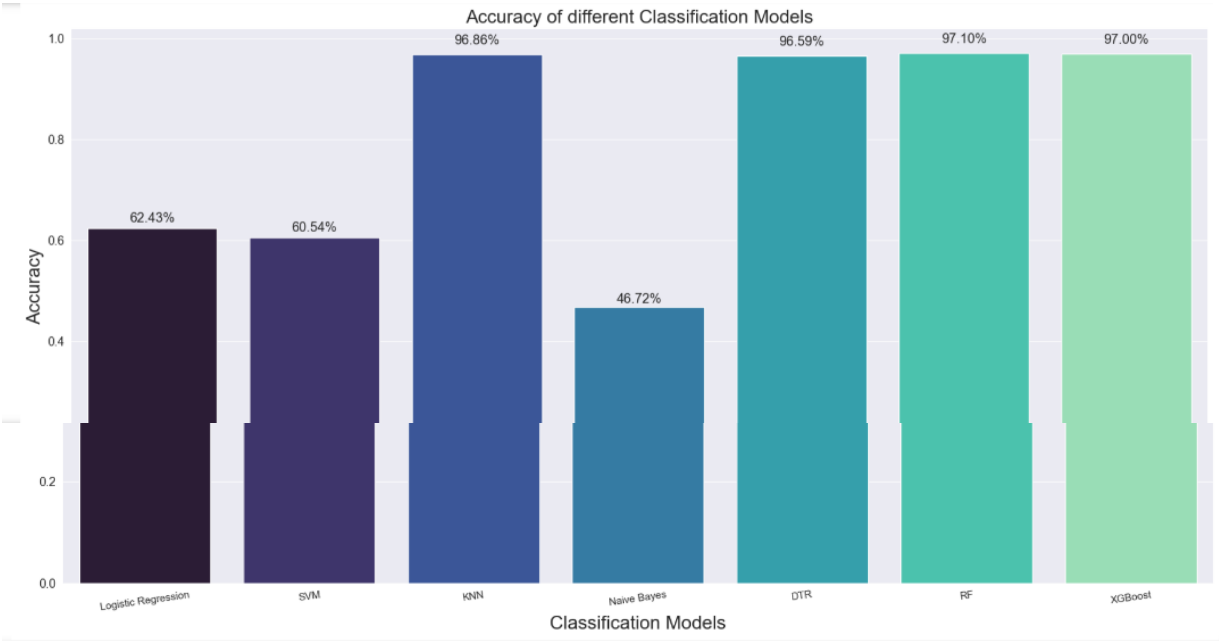


Figure 10:comparaison des modèles

Enregistrez le modèle entraîné qui a une meilleure performance dans un fichier au format pickle.Le modèle ensuite va être chargé à partir de ce fichier pour une utilisation dans notre interface Streamlit.

```
with open('votre_fichier.pkl', 'rb') as fichier:
model_charge = pickle.load(fichier)
```

VI. Développement de l'application web

Intégration du modèle sérialisé au format Pickle dans l'application.

```
import streamlit as st
import pickle
import pandas as pd
from sklearn import preprocessing
import numpy as np

# Load the pre-trained model
```

```
with open('modele_classification13.pkl', 'rb') as model_file:
    model = pickle.load(model_file)
```

Pour créer un formulaire pour la saisie des données météorologiques dans une interface Streamlit, vous pouvez utiliser le code suivant comme point de départ.
Streamlit simplifie la création d'interfaces utilisateur interactives en Python.
N'oubliez pas d'ajuster le formulaire en fonction des caractéristiques spécifiques que vous souhaitez collecter. Vous pouvez également améliorer ou personnaliser davantage l'interface en fonction de vos besoins.

Voici un exemple de formulaire pour la saisie des données météorologiques en utilisant Streamlit :

```
def main():
    st.title("Weather Prediction App")

    # User input for each feature
    temperature = st.slider("Temperature", min_value=0.0, max_value=400.0)
    temp_min = st.slider("Minimum Temperature", min_value=0.0, max_value=400.0)
    temp_max = st.slider("Maximum Temperature", min_value=0.0, max_value=400.0)
    pressure = st.slider("Pressure", min_value=900, max_value=1100)
    sea_level = st.slider("Sea Level", min_value=900, max_value=1100)
    humidity = st.slider("Humidity", min_value=0, max_value=100)
    wind_speed = st.slider("Wind Speed", min_value=0.0, max_value=20.0)
    lon = st.number_input("Longitude")
    lat = st.number_input("Latitude")
    # Create a DataFrame with user input
    user_data = pd.DataFrame({

        'Temperature': [temperature],
        'Temp_Min': [temp_min],
        'Temp_Max': [temp_max],
        'Pressure': [pressure],
        'Sea_Level': [sea_level],
        'Humidity': [humidity],
        'Wind Speed': [wind_speed],
        'Lon': [lon],

      'Lat': [lat]
      })
      # Load the label_encoder
      with open('label_encoder.pkl', 'rb') as label_encoder_file:
          label_encoder1 = pickle.load(label_encoder_file)
      # Make a prediction
      if st.button("Predict"):
          scaler = preprocessing.StandardScaler()
          processed_data = scaler.fit_transform(user_data)
```

```
        prediction = model.predict(processed_data)
        print("predictions:",prediction)

        predicted_labels = label_encoder1.inverse_transform(prediction)

        print("Raw predictions:", predicted_labels)    st.success(f"Prediction:
{predicted_labels}"  )    if __name__ == "__main__":  main()
```

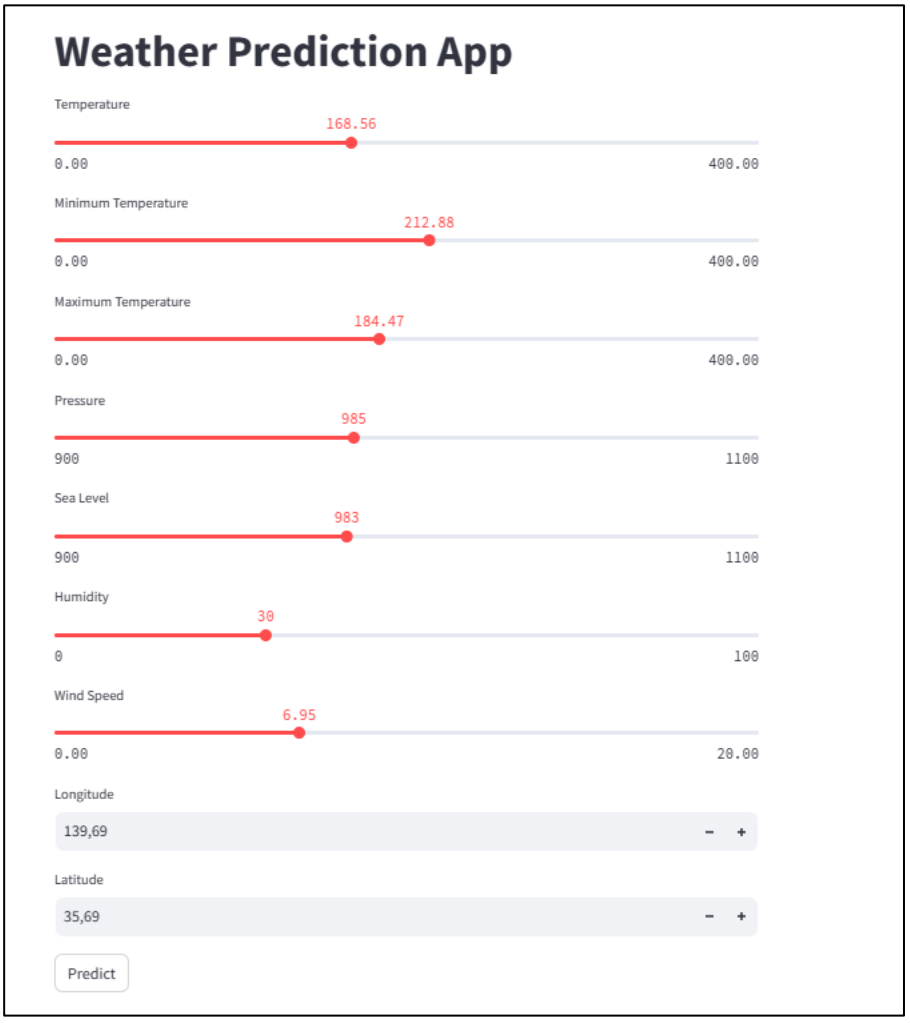


Figure 11:interface web

VII. Mise en place avec Apache Airflow

1. Construction d'une tâche pour chaque étape du pipeline.

La première tâche c'est de mettre à jour votre fichier CSV contenant des données météorologiques. Il utilise l'API OpenWeatherMap pour récupérer les prévisions météorologiques futures de plusieurs villes à travers le monde.

Les données actuelles du fichier CSV sont préalablement chargées, puis le script interroge l'API pour obtenir les prévisions météorologiques les plus récentes pour chacune des villes spécifiées.

Le script compare ensuite la date de chaque prévision avec la date maximale existante dans le fichier CSV pour déterminer s'il y a de nouvelles données à ajouter.
Si une prévision a une date ultérieure à la date maximale existante, elle est ajoutée au fichier CSV, actualisant ainsi le jeu de données avec les prévisions météorologiques les plus récentes.

Voici le contenu du fichier Task 1.py

```
import requests
import pandas as pd
from datetime import datetime

api_key = 'votre api key'
endpoint = 'https://api.openweathermap.org/data/2.5/forecast'

existing_data = pd.read_csv('/opt/airflow/dags/scripts/weather_5.csv')

existing_data['Date'] = pd.to_datetime(existing_data['Date'], errors='coerce')

cities = [
    "Tokyo", "Mexico", "Séoul", "Jakarta", "New Delhi", "Le Caire",
"Moscou",……."Melekeok"]//remplir par les villes que  vous souhaitez

new_data_list = []

for city in cities:
    url = f'{endpoint}?q={city}&APPID={api_key}'
    response = requests.get(url)

    if response.status_code == 200:
        weather_data = response.json()

        if 'list' in weather_data:
            city_forecast = weather_data['list']

            for forecast in city_forecast:
                timestamp = forecast['dt']
                date = datetime.utcfromtimestamp(timestamp).strftime('%Y-%m-%d %H:%M:%S')
                forecast_date = pd.to_datetime(date)
```

```
            if forecast_date > existing_data['Date'].max():
                city_info = {
                    'City': city,
                    'Date': date,
                    'Temperature': forecast['main']['temp'],
                    'Temp_Min': forecast['main']['temp_min'],
                    'Temp_Max': forecast['main']['temp_max'],
                    'Pressure': forecast['main']['pressure'],
                    'Sea_Level': forecast['main']['sea_level']
                    if 'sea_level' in forecast['main'] else None,
                    'Humidity': forecast['main']['humidity'],
                    'Description': forecast['weather'][0]['main'],
                    'Wind Speed': forecast['wind']['speed'],
                    'Country': weather_data['city']['country'],
                    'Lon': weather_data['city']['coord']['lon'],
                    'Lat': weather_data['city']['coord']['lat'],
                }
                new_data_list.append(city_info)
        else:
            print(f"No forecast data for {city}")
    else:
        print(f"Error {city}: {response.status_code}")

new_data_df = pd.DataFrame(new_data_list)

if not new_data_df.empty:
    new_data_df.to_csv('/opt/airflow/dags/scripts/weather_5.csv', index=False, mode='a',
header=False)
else:
    print("Aucune nouvelle donnée à ajouter.")
```

Le fichier du task 2 contient le script qui effectue un flux de travail complet, de la préparation des données jusqu'à la sauvegarde du meilleur modèle pour une utilisation ultérieure .

```
data = pd.read_csv("/opt/airflow/votre_Fichier_CSV.csv")

# Drop unnecessary columns

# Label encoding
from sklearn import preprocessing
label_encoder = preprocessing.LabelEncoder()
data['Description'] = label_encoder.fit_transform(data['Description'])

data = data.drop(['Date', 'City', 'Country'], axis=1).dropna()
X = data.drop('Description', axis=1)
y = data['Description']
```

```
# Split the dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Feature Scaling
sc = StandardScaler()
X_train = sc.fit_transform(X_train) X_test = sc.transform(X_test)


    # Train different models
    models = {
        "Logistic Regression": LogisticRegression(random_state=42),
        "SVM": SVC(kernel='linear', random_state=42),
        "KNN": KNeighborsClassifier(n_neighbors=5),
        "Naive Bayes": GaussianNB(),
        "Decision Tree": DecisionTreeClassifier(criterion='entropy', random_state=42),
        "Random Forest": RandomForestClassifier(n_estimators=40, random_state=42),
        "XGBoost": XGBClassifier()
    }

    results = {}

    for model_name, model in models.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        accuracy = accuracy_score(y_test, y_pred)
        results[model_name] = accuracy

    # Find the best model
    best_model_name = max(results, key=results.get)
    best_accuracy = results[best_model_name]

    print(f"Results:")
    for model, accuracy in results.items():
        print(f"{model}: {accuracy:.2%}")

    print(f"Best Model: {best_model_name} with Accuracy: {best_accuracy:.2%}")

    # Save the best model as a pickle file
    file_name = '/opt/airflow/dags/scripts/best_model.pkl'
    with open(file_name, 'wb') as file:
        pickle.dump(models[best_model_name], file)
```

Task3 constitue une application permettant aux utilisateurs de saisir des données météorologiques et d'obtenir une prédiction de la description météorologique

(« Clear» « Run» « Clouds» « Snow») correspondante à l'aide de votre modèle et on a aussi une interface pour la visualisation .

```
import streamlit as st
import pickle
import pandas as pd
from sklearn import preprocessing
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px
from matplotlib.ticker import MaxNLocator
import filter
import visualization as viz

# Load the pre-trained model
with open('/opt/airflow/dags/scripts/pickle_files/votre_model.pkl', 'rb') as
model_file:
    model = pickle.load(model_file)

label_encoder = preprocessing.LabelEncoder()
# Main Streamlit app
def main():
    st.title("Weather Prediction App")
    # Sidebar navigation
    page = st.sidebar.selectbox("Choose a page", ["Home", "Prediction"])
    if page == "Home":
        data = pd.read_csv("/opt/airflow/dags/scripts/Weather_Data.csv")
        cities = filter.get_unique_cities_and_countries(data)
        city_selected = filter.create_city_country_dropdown(cities)

        # Call the function to get filtered records
        filtered_records = filter.filter_records_by_city(data, city_selected)

        # Display the selected city in the map
        viz.map(filtered_records)

        # Dataset features
        all_features = ['Temperature', 'Temp_Min', 'Temp_Max', 'Pressure',
'Humidity', 'Wind Speed']

        viz.create_viz(filtered_records,all_features)


    elif page == "Prediction":

        st.write("Make a weather prediction:")
        # User input for each feature
```

```
        temperature = st.slider("Temperature", min_value=0.0, max_value=400.0)
        temp_min = st.slider("Minimum Temperature", min_value=0.0,
max_value=400.0)
        temp_max = st.slider("Maximum Temperature", min_value=0.0,
max_value=400.0)
        pressure = st.slider("Pressure", min_value=900, max_value=1100)
        sea_level = st.slider("Sea Level", min_value=900, max_value=1100)
        humidity = st.slider("Humidity", min_value=0, max_value=100)
        wind_speed = st.slider("Wind Speed", min_value=0.0, max_value=20.0)
        lon = st.number_input("Longitude")
        lat = st.number_input("Latitude")
        # Create a DataFrame with user input
        user_data = pd.DataFrame({
            'Temperature': [temperature],
            'Temp_Min': [temp_min],
            'Temp_Max': [temp_max],
            'Pressure': [pressure],
            'Sea_Level': [sea_level],
            'Humidity': [humidity],
            'Wind Speed': [wind_speed],
            'Lon': [lon],
            'Lat': [lat]
        })
        # Load the label_encoder
        with open('/opt/airflow/dags/pickle_files/label_encoder.pkl', 'rb') as
label_encoder_file:
            label_encoder1 = pickle.load(label_encoder_file)
        # Make a prediction
        if st.button("Predict"):
            # scaler = preprocessing.StandardScaler()
            # processed_data = scaler.fit_transform(nouvelles_donnees1)
            print(user_data)
            prediction = model.predict(user_data)
            print("Raw predictions:",prediction)
            st.success(f"Prediction: {prediction}")

if __name__ == "__main__":
    main()
```

Créez votre Dag **weather_Dag**

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime, timedelta
from pytz import timezone

# Set the timezone to Casablanca
casablanca_tz = timezone('Africa/Casablanca')

default_args = {
    'owner': 'salma_arbaoui',
    'depends_on_past': False,
    'start_date': datetime.now(casablanca_tz),
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 0,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'Weather_app',
    default_args=default_args,
    description='An Airflow DAG to fetch weather data and update the CSV file',
    schedule_interval='0 */3 * * *',  # executer chaque 3h
    max_active_runs=1,
    catchup=False,
)


bash_command_main = f'python /opt/airflow/dags/scripts/main.py'

# Task to execute the Python script
execute_script_task_1 = BashOperator(
    task_id='task_1',
    bash_command=bash_command_main,
    dag=dag

)

# Task to execute the Python script
execute_script_task_2 = BashOperator(
    task_id='task_2',
    bash_command='python /opt/airflow/dags/scripts/Task2.py',
    dag=dag
)
```

```
# Task to execute the Python script
execute_script_task_3 = BashOperator(
    task_id='task_3',
    bash_command='streamlit run --server.address 0.0.0.0
/opt/airflow/dags/scripts/streamlit_app.py',
    dag=dag
)

execute_script_task_1 >> execute_script_task_2 >> execute_script_task_3
```

Ce script Python crée votre Dag dans Apache Airflow pour automatiser l'exécution de tâches liées à une application météorologique

Pour la Configuration du DAG (Weather_app) :

default_args: Paramètres par défaut pour le DAG, notamment le propriétaire, la date de début, etc.
dag: Crée une instance du DAG avec des paramètres spécifiques, tels que la planification (schedule_interval), le nombre maximal d'exécutions actives (max_active_runs), et l'option de rattrapage (catchup).
Tâche d'installation des packages :

bash_command_main: Commande bash pour exécuter le script principal (main.py).

Les flèches >> indiquent les dépendances entre les tâches. install_packages_task doit être exécuté avant execute_script_task_1, et ainsi de suite.

Ces tâches sont organisées de manière à respecter les dépendances entre elles. Le DAG est planifié pour s'exécuter toutes les trois heures (schedule_interval='0 */3 * * *'), et il ne tient pas compte des périodes manquées depuis la date de début (catchup=False). Il assure également qu'une seule exécution est active à la fois (max_active_runs=1).

2. Déploiement du pipeline sur Apache Airflow

Veuillez ouvrir votre navigateur web et accéder à l'adresse http://localhost:8080 pour accéder à l'interface utilisateur d'Apache Airflow.

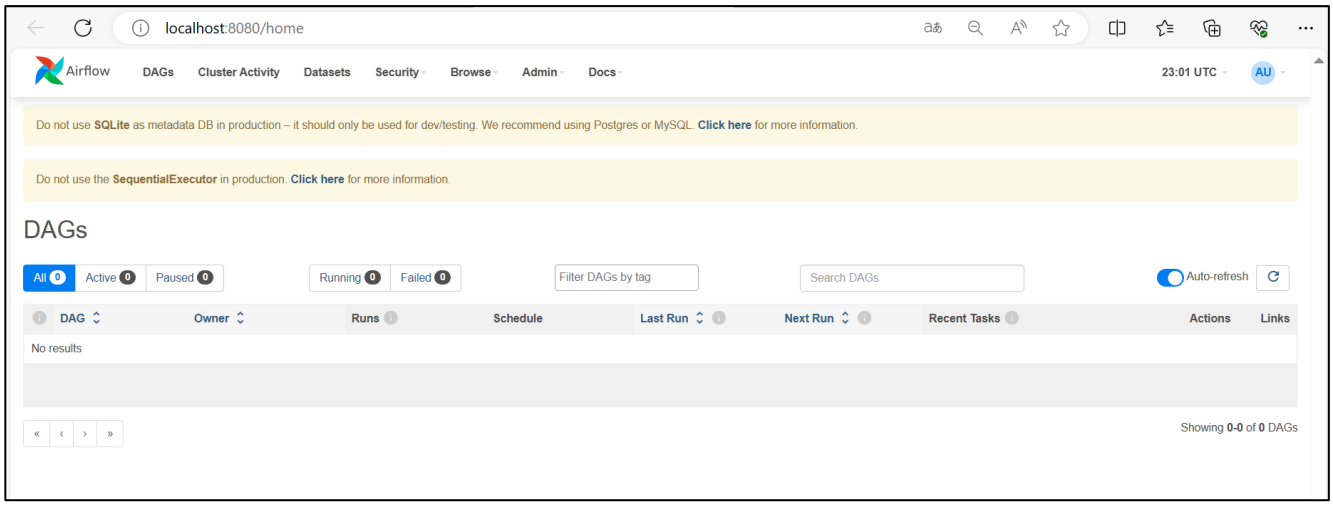


Figure 12:Interface avant le rechargement de votre dag

Dans la Figure 13, vous pouvez observer l'interface avant le rechargement de votre DAG

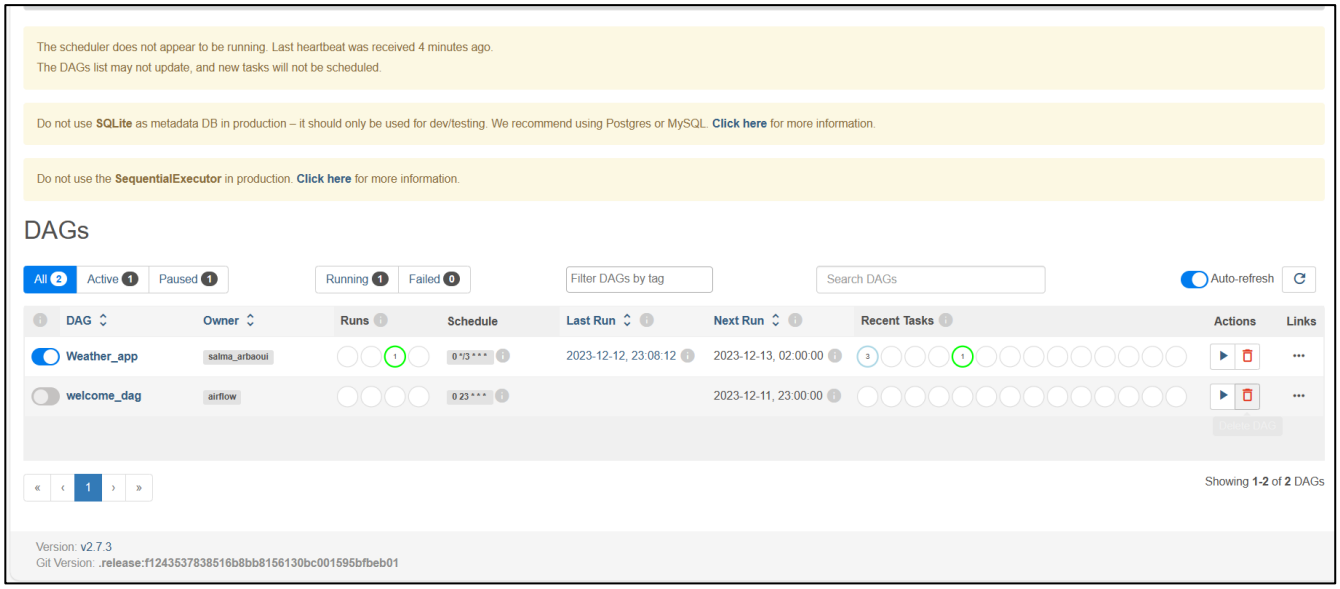


Figure 13:visualisation Dag

Maintenant vous pourrez visualiser votre DAG nommé "Weather_App". Lancer son exécution pour initier le flux de travail défini dans le DAG.

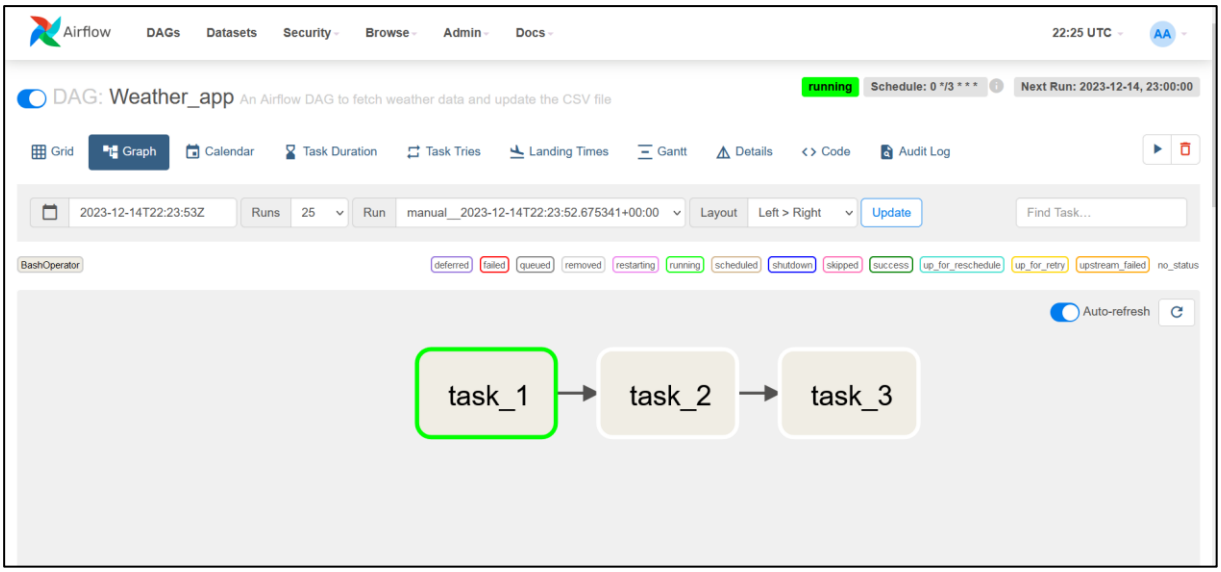
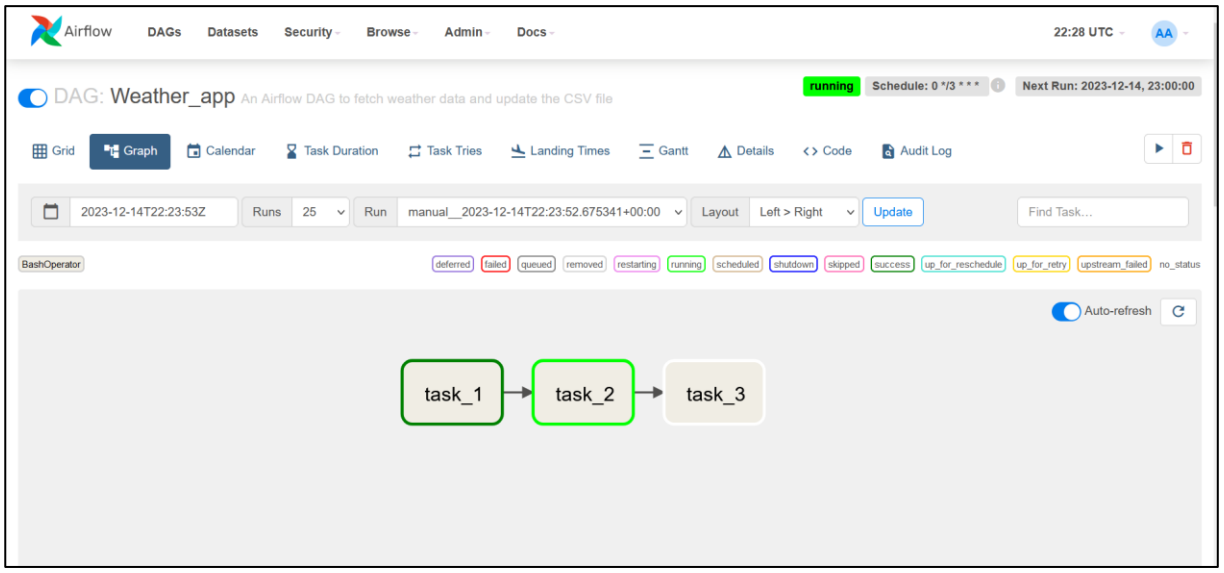


Figure 14:le premier task en execution

execute_script_task_1: Tâche Bash pour mettre à jour votre fichier CSV contenant des données météorologiques.

execute_script_task_2: Tâche Bash pour exécuter le script qui effectue un flux de travail complet, de la préparation des données jusqu'à la sauvegarde du meilleur modèle pour une utilisation ultérieure .

execute_script_task_3: Tâche Bash pour exécuter un script Streamlit (streamlit_app.py).



La première tâche a été exécutée avec succès ! Maintenant, passons à la deuxième tâche qui consiste à entrainer les modèles de prédiction et de générer le fichier pickle qui contient le model celui avec la meilleure performance.

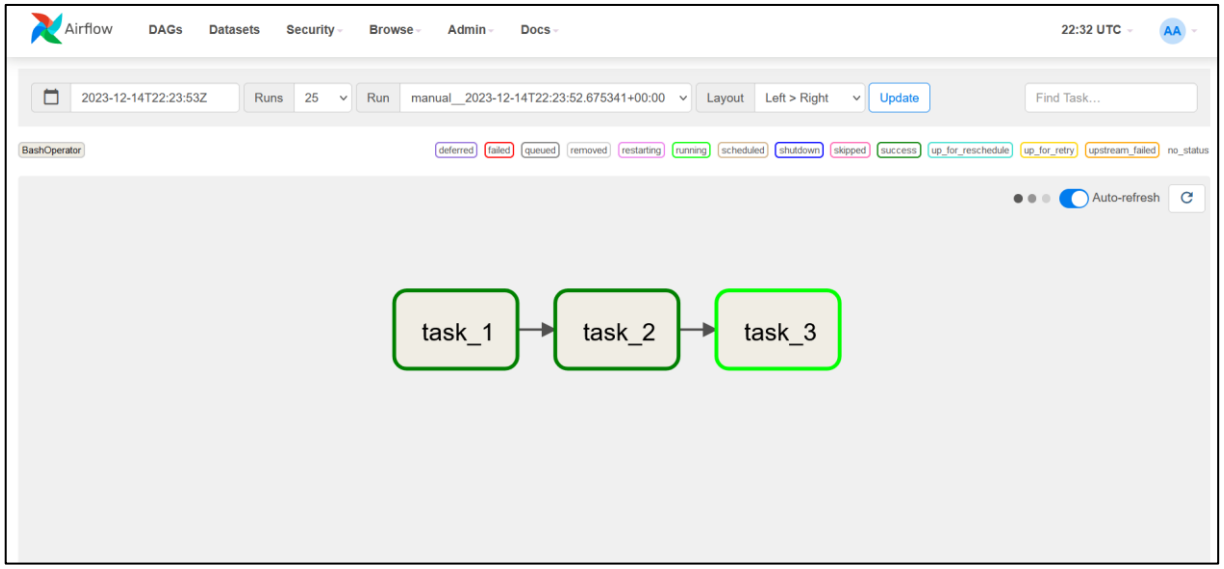


Figure 15:execution task3

Toutes les tâches ont été exécutées avec succès figure 16 , et la dernière tâche est toujours en cours d'exécution car elle est responsable du lancement de l'application. Après l'exécution de cette tâche, le processus recommencera, effectuant le même traitement de manière itérative.

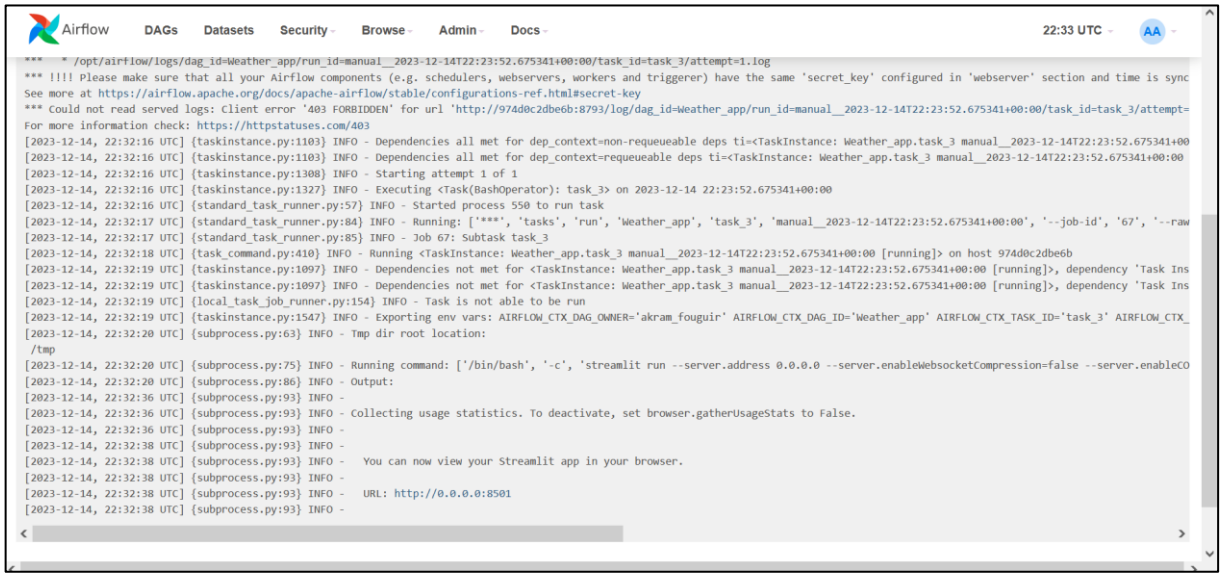


Figure 16:Execution Dag avec success

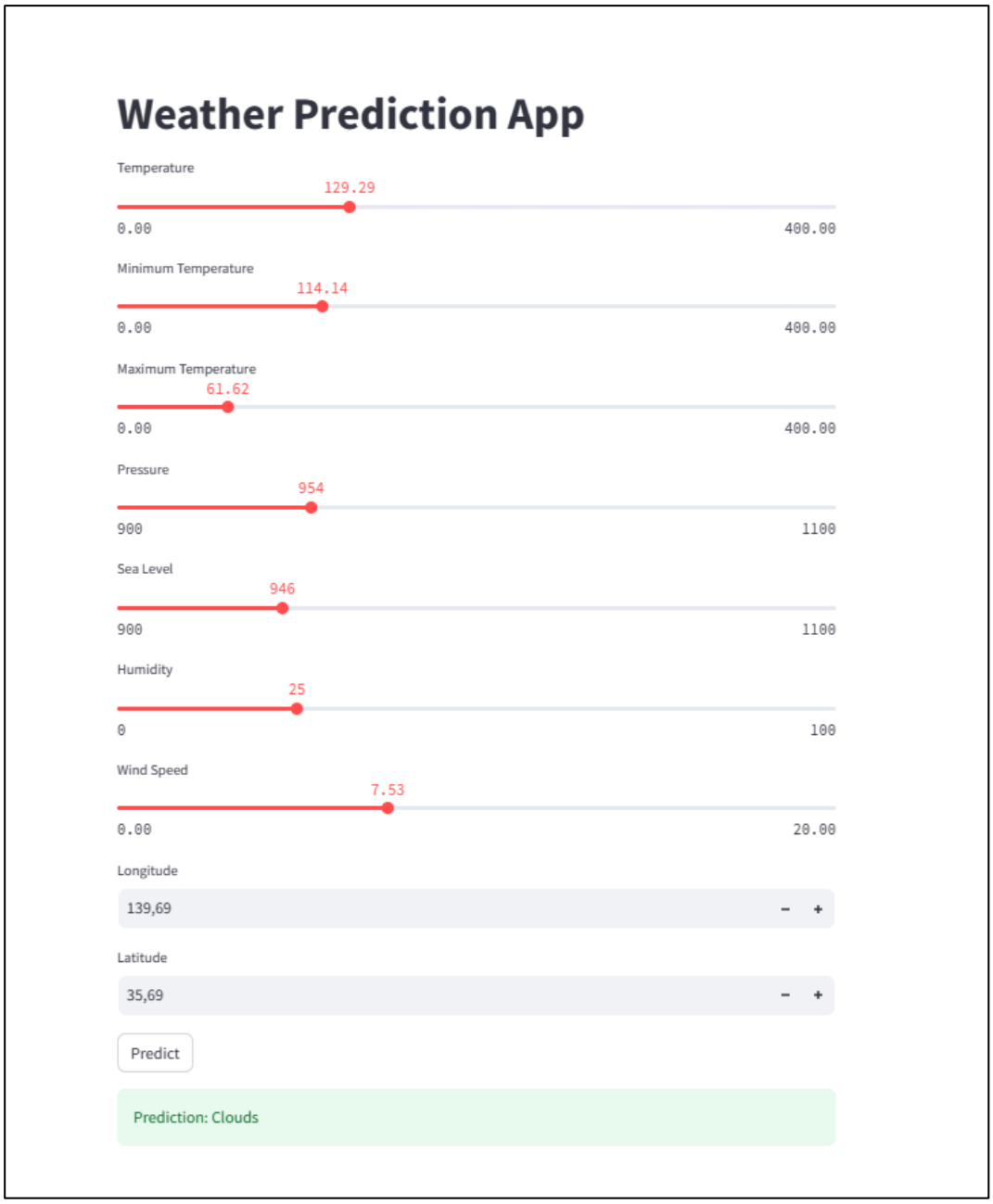


Figure 17:interface finale avec prediction


Weather Prediction App
Temperature
62.12
0.00
400.00
Minimum Temperature
53.79
0.00
400.00
Maximum Temperature
70.45
0.00
400.00
Pressure
953
900
1100
Sea Level
994
900
1100
Humidity
44
0
100
Wind Speed
12.20
0.00
20.00
Longitude
139,69
Latitude
35,69
Predict
Prediction: Rain

Sur la page de visualisation, vous pouvez spécifier la ville dont vous souhaitez afficher les données.

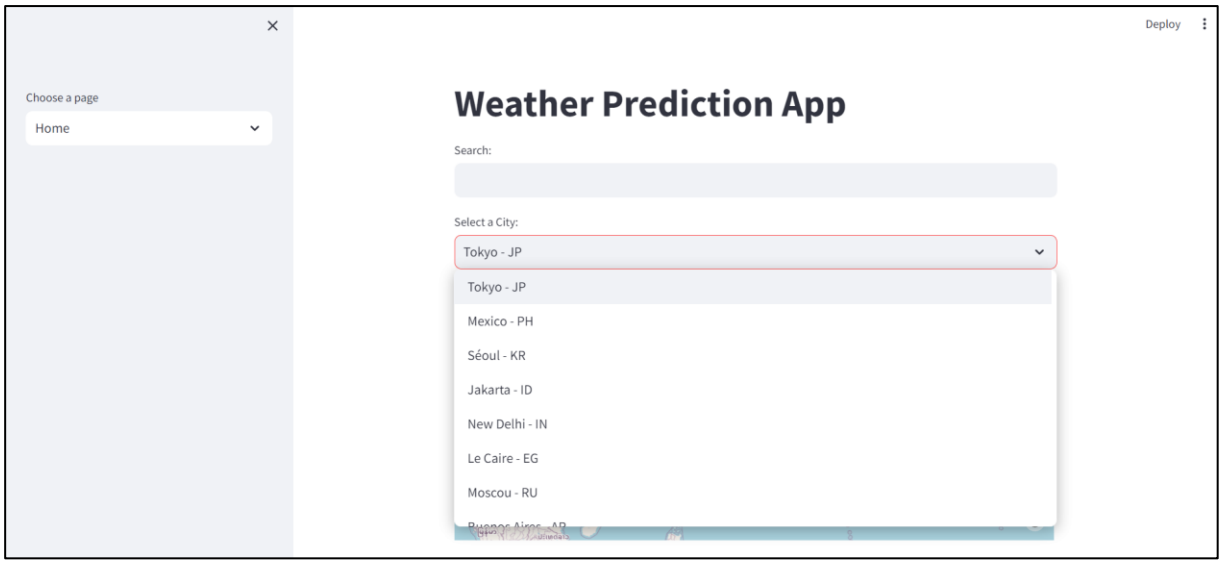


Figure 18:interface choix de la capitale pour la visualisation:

Celles-ci seront alors affichées sur la carte.

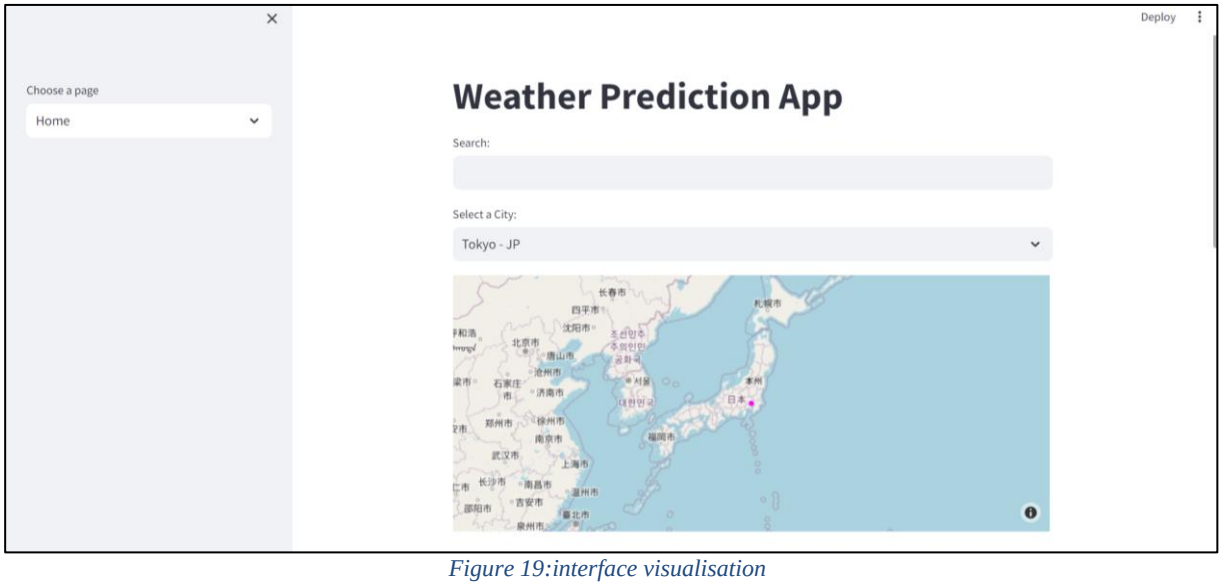


Figure 19:interface visualisation

En parcourant la page, vous rencontrerez un menu déroulant qui propose une liste complète des visualisations disponibles. Par exemple, pour chaque type de visualisation, comme un scatter plot, un histogramme ou un graphique en barres, vous avez la possibilité de spécifier les caractéristiques pertinentes à afficher. Dans notre cas, nous avons choisi un graphique en courbes (line chart) pour visualiser l'évolution de la température au fil du temps, mais cette sélection peut être répétée pour tous les autres types de visualisations disponibles.

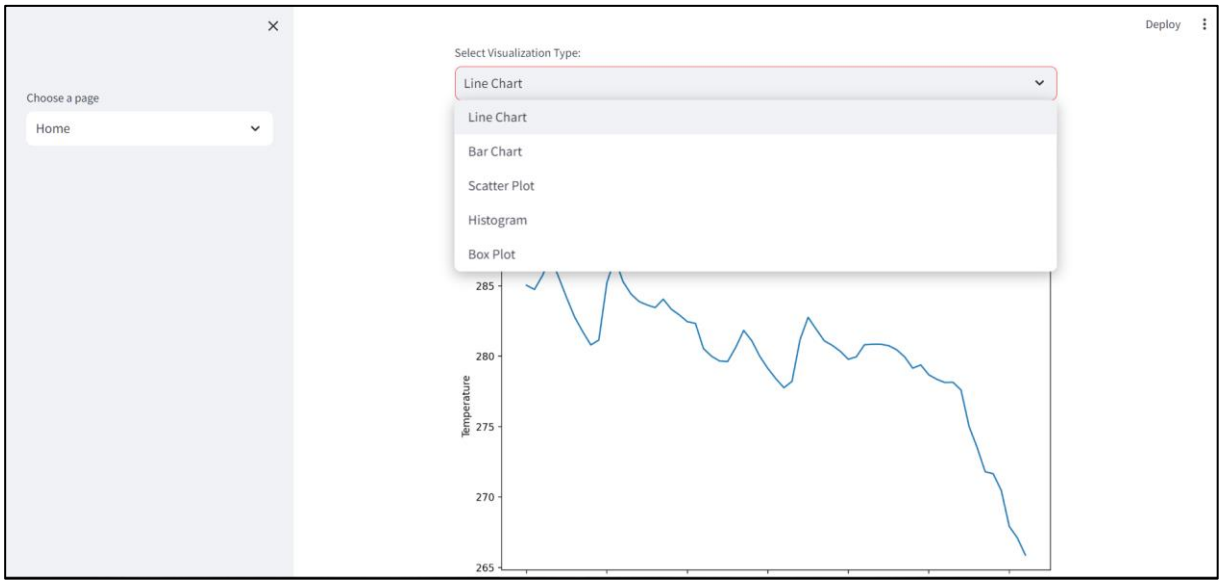


Figure 20:choisir type visualisation

Ainsi, après avoir choisi le type de visualisation (scatter plot), vous spécifiez ensuite les deux caractéristiques que vous souhaitez afficher.