

Guide du Projet

Filière : **Ingénierie Informatique et Technologies Emergentes (2ITE)**
3 ème année Cycle Ingénieur

**Mise en place d'un pipeline
d'entraînement continu en utilisant
Apache Airflow, FastAPI, Flask et**

Réalisé par :

LAYOUNE Ghita

HANIM Hanae

YOUSFI Meryem

Encadré par :

Mr. KALLOUBI

I. TABLE DES MATIERES

I.	Objectif	4
II.	Prérequis - Installation & Configuration	4
1.	Environnement de travail	4
2.	Configuration des conteneurs.....	4
III.	Création de l'application Fast API	7
1.	Choix du model	7
2.	Créations du modèle sérialisé.....	23
3.	Création de l'API	26
IV.	Automatisation du pipeline avec Airflow	28
1.	Création des DAG	28
2.	Résultats d'exécution	33
V.	Création de l'interface avec Flask	33
VI.	Conteneurisation de l'API.....	37
	Annexe	43

LISTE DES FIGURE

Figure I-1: Diabetes Dataset.....	4
Figure II-1: Configuration du namenode	5
Figure II-2: Datanode & Ressource Manager	5
Figure II-3: NodeManager & HistoryServer	6
Figure II-4: Configuration de Zeppelin	6
Figure II-5 : Création et lancement du Cluster.....	7
Figure III-1: Vue du dataset	8
Figure III-2: API diabète	26
Figure III-3: API Predict	27
Figure III-4: Modèle diabetes.....	27
Figure III-5: Test de l'API	28
Figure IV-1: final_diabetes.py	29
Figure IV-2: Configuration et Importation des Modules	30
Figure IV-3: Arguments par Défaut et Initialisation du DAG	30
Figure IV-4: Tâche de Détection de Fichier.....	31
Figure IV-5: Tâche de Vérification de Modification de Fichier	31
Figure IV-6: Tâche d'Exécution de Script Python	31
Figure IV-7: Tâche de Saut d'Exécution de Script Python.....	32
Figure IV-8: Tâche d'Exécution de l'Application FastAPI.....	32
Figure IV-9: Dépendances entre les Tâches.....	32
Figure IV-10: Interface airflow	33
Figure V-1: API Fast du projet Diabetes.....	33
Figure V-2: Main du flask	34
Figure V-3 : Suite du Main	34
Figure V-4: fichier html Index	35
Figure V-5:Suite du fichier html Index	35
Figure V-6: Fichier de style css.....	36
Figure V-7: Interface de l'application.....	36
Figure V-8: Interface de l'application.....	37
Figure V-9: Le résultat du test.....	37
Figure VI-1:docker file.....	38
Figure VI-2: requirements.txt.....	38
Figure VI-3: Lancement du container	39
Figure VI-4: Conteneurisation de l'API	39
Figure VI-5: Api conteneurisée.....	40
Figure VI-6: Lien vers l'API.....	40
Figure VI-7: Application Flask API.....	41

I. Objectif

L'objectif de ce projet est d'automatiser l'ensemble du processus, de l'entraînement du modèle à la mise à disposition d'une API de prédiction, en passant par la création d'une application web pour l'interaction avec le modèle. L'utilisation d'outils tels que FastAPI, Apache Airflow, Flask, et Docker vise à rendre le processus efficace, automatisé et évolutif. Tous cela en opérant sur le dataset issue de kaggle :

<https://www.kaggle.com/datasets/mathchi/diabetes-data-set>

The screenshot shows the Kaggle dataset page for the Diabetes Dataset. At the top, it displays the title "Diabetes Dataset" and a small thumbnail image of white medical capsules. Below the title, it says "This dataset is originally from the N. Inst. of Diabetes & Diges. & Kidney Dis." Under the title, there are three buttons: "Data Card", "Code (294)", and "Discussion (4)". The "Data Card" button is underlined, indicating it is the active tab. On the right side, there are several metadata fields: "Usability" (10.00), "License" (CC0: Public Domain), "Expected update frequency" (Never), and a "Tags" section containing the word "Diabetes".

Figure I-1: Diabetes Dataset

II. Prérequis - Installation & Configuration

Pour mettre en place le pipeline d'entraînement continu avec Apache Airflow, FastAPI, Flask, et Docker, plusieurs prérequis doivent être satisfaits.

1. Environnement de travail

L'environnement de travail :

- Docker : Suivre les instructions d'installation sur le site officiel de Docker (<https://docs.docker.com/get-docker/>).
- Docker-compose : Installer Docker-compose pour faciliter la gestion des conteneurs

2. Configuration des conteneurs

Nous allons tous d'abord créer notre docker compose :

```

version: "3"

services:
# Setting Up HDFS & YARN #
namenode:
  image: mrugankray/namenode-spark-airflow-flume-zepplin:1.1
  container_name: namenode
  restart: always
  ports:
    - 9872:9872
    - 9001:9001
    - 8080:8080 # spark master web ui
    - 8081:8081 # spark slave web ui
    - 4040:4040 # spark driver web ui
    - 3000:3000 # airflow ui
  volumes:
    - hadoop_namenode:/hadoop/dfs/name
    - hadoop_namenode_conda:/root/anaconda
    - hadoop_namenode_spark:/opt/spark
    - ./configs/namenode_bashrc.txt:/root/.bashrc
    - ./configs/namenode_airflow.cfg:/root/airflow/airflow.cfg
    - ./dags:/root/airflow/dags
    - airflow_namenode:/root/airflow
    - ./configs/namenode_flume-env.sh:/opt/flume/conf/flume-env.sh
    - ./flume_config/flume.conf:/opt/flume/conf/flume.conf
    - hadoop_namenode_flume:/opt/flume
  environment:
    - CLUSTER_NAME=hadoop-learning
    - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
  env_file:
    - ./hadoop.env

```

Figure II-1: Configuration du namenode

La figure 1, présente la configuration du namenode, nous trouvions comme volume celui de namenode, flume et d'autre, ce qui nous concerne à présent est l'utilité de cette image pour le stockage du fichier du dataset.

```

datanode:
  image: mrugankray/datanode-python:1.0
  container_name: datanode
  restart: always
  volumes:
    - hadoop_datanode:/hadoop/dfs/data
    - hadoop_datanode_conda:/root/anaconda
    - ./configs/datanode_bashrc.txt:/root/.bashrc
  environment:
    SERVICE_PRECONDITION: "namenode:9870"
    CORE_CONF_fs_defaultFS: "hdfs://namenode:9000"
  ports:
    - "9864:9864"
  env_file:
    - ./hadoop.env

resourcemanager:
  image: mrugankray/resourcemanager-python:1.0
  container_name: resourcemanager
  restart: always
  volumes:
    - hadoop_resourcemanager_conda:/root/anaconda
    - ./configs/resourcemanager_bashrc.txt:/root/.bashrc
  environment:
    SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864"
  ports:
    - "8088:8088"
  env_file:
    - ./hadoop.env

```

Figure II-2: Datanode & Ressource Manager

```

nodemanager:
  image: mrugankray/nodemanager-python:1.0
  container_name: nodemanager
  restart: always
  volumes:
    - hadoop_nodemanager_conda:/root/anaconda
    - ./configs/nodemanager_bashrc.txt:/root/.bashrc
  environment:
    SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864 resourcemanager:8088"
  ports:
    - "8042:8042"
    - "19888:19888" # to access job history
  env_file:
    - ./hadoop.env

historyserver:
  image: bde2020/hadoop-historyserver:2.0.0-hadoop3.2.1-java8
  container_name: historyserver
  restart: always
  environment:
    SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode:9864 resourcemanager:8088"
  ports:
    - "8188:8188"
  volumes:
    - hadoop_historyserver:/hadoop/yarn/timeline
  env_file:
    - ./hadoop.env

```

Figure II-3: NodeManager & HistoryServer

Les figures 2 & 3 présentent les configurations des démons nécessaires de HDFS. Ils sont utiles pour lancement du namenode.

```

# Zeppelin #
zeppelin:
  image: apache/zeppelin:0.10.1
  environment:
    ZEPPELIN_PORT: 8082
    ZEPPELIN_JAVA_OPTS: >-
      -Dspark.driver.memory=1g
      -Dspark.executor.memory=1g
      -Dspark.rdd.compress=true
      -Dspark.serializer=org.apache.spark.serializer.KryoSerializer
      -Dspark.kryo.registrator=org.zouzias.spark.lucenerdd.LuceneRDDKryoRegistrar
      -Dspark.driver.extraJavaOptions=-Dlucenerdd.index.store.mode=disk
      -Dspark.executor.extraJavaOptions=-Dlucenerdd.index.store.mode=disk
  master: local[*]
  ports:
    - 8082:8082
  volumes:
    - ./data:/zeppelin/data
    - ./notebooks:/zeppelin/notebook
    - ./zeppelin-logs:/zeppelin/logs
    - ./zeppelin-run:/zeppelin/run

```

Figure II-4: Configuration de Zeppelin

Apache Zeppelin est un notebook open source basé sur le web qui permet l'analyse interactive des données et la science des données collaborative. Il offre un environnement interactif et multilingue avec une prise en charge intégrée de divers backends de traitement des données, notamment Apache Spark, Apache Flink, Apache Hadoop, et d'autres. Zeppelin permet aux

utilisateurs de créer et de partager des documents (appelés notebooks) contenant du code en direct, des équations, des visualisations et du texte narratif.

Cette configuration (illustrer dans la figure 4) nous permettra de créer l'image propre à Zeppelin.

Et bien d'autres service qu'on peut ajouter à notre fichier *docker.yml* pour élargir notre cluster contenant les services Big Data.

Pour créer le cluster des containers nous devons lancer la commande *docker compose up -d*

```
C:\Users\rifad\Desktop\BIG DATA\Atelier Sqoop, Hive, Spark\Big-Data-Cluster-main>docker-compose up -d
[+] Building 0.0s (0/0)
[+] Running 22/22
  ✓ Container kadmin                Started      0.0s
  ✓ Container mysql                 Started      0.0s
  ✓ Container cassandra             Running     0.0s
  ✓ Container big-data-cluster-main-spark-1 Started      0.0s
  ✓ Container hive-metastore-postgresql Started      0.0s
  ✓ Container huedb                 Started      0.0s
  ✓ Container nodemanager           Running     0.0s
  ✓ Container datanode              Running     0.0s
  ✓ Container resourcemanager       Running     0.0s
  ✓ Container namenode              Started      0.0s
  ✓ Container historyserver         Running     0.0s
  ✓ Container external_postgres_db  Started      0.0s
  ✓ Container big-data-cluster-main-spark-worker-1 Started      0.0s
  ✓ Container big-data-cluster-main-zeppelin-1        Started      0.0s
  ✓ Container hive-metastore         Started      0.0s
  ✓ Container zookeeper              Started      0.0s
  ✓ Container hue                   Started      0.0s
  ✓ Container external_pgadmin      Started      0.0s
  ✓ Container hive-server            Started      0.0s
  ✓ Container kafka-broker          Started      0.0s
  ✓ Container schema-registry       Running     0.0s
  ✓ Container control-center        Running     0.0s
```

Figure II-5 : Création et lancement du Cluster

Comme démontré dans la figure 5, tous les services ont été créer avec succès et sont en un état de Running.

Pour accéder à :

- Zepelin : <http://localhost:8082>
- Namenode : <http://localhost:9000>
- Airflow: <http://localhost:3000>

Remarque:

Assurez-vous que les ports alloués aux conteneurs dans le fichier docker-compose.yml sont vides et accessibles, et ne sont pas utiliser par d'autres applications système.

III. Création de l'application Fast API

1. Choix du model

Avant de choisir le model, il faut tous d'abord comprendre la thématique d'après le dataset :

A	B	C	D	E	F	G	H	I	
1	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
2	6	148	72	35	0	33.6	0.627	50	1
3	1	85	66	29	0	26.6	0.351	31	0
4	8	183	64	0	0	23.3	0.672	32	1
5	1	89	66	23	94	28.1	0.167	21	0
6	0	137	40	35	168	43.1	2.288	33	1
7	5	116	74	0	0	25.6	0.201	30	0
8	3	78	50	32	88	31	0.248	26	1
9	10	115	0	0	0	35.3	0.134	29	0
10	2	197	70	45	543	30.5	0.158	53	1
11	8	125	96	0	0	0	0.232	54	1
12	4	110	92	0	0	37.6	0.191	30	0
13	10	168	74	0	0	38	0.537	34	1
14	10	139	80	0	0	27.1	1.441	57	0
15	1	189	60	23	846	30.1	0.398	59	1
16	5	166	72	19	175	25.8	0.587	51	1
17	7	100	0	0	0	30	0.484	32	1
18	0	118	84	47	230	45.8	0.551	31	1
19	7	107	74	0	0	29.6	0.254	31	1
20	1	103	30	38	83	43.3	0.183	33	0
21	1	115	70	30	96	34.6	0.529	32	1
22	0	126	99	41	225	29.2	0.704	27	0

Figure III-1: Vue du dataset

On peut remarquer que la colonne Outcome contient des valeurs (0,1), donc c'est un problème de classification.

Maintenant, commençant le prétraitement. Dans ce notebook Zeppelin, je souhaite organiser les quatre modèles d'ensemble suivants :

- Stacking Model
- Soft Voting Model
- Hard Voting Model
- XGBoost

Notez que le dataset doit être stockée sur le container du namenode :

```
hdfs_path = "hdfs://namenode:9000/input/diabetes.csv"
```

Le jeu de données sur le diabète se compose de :

- Pregnancies: Nombre de grossesses
- Glucose : Concentration plasmatique en glucose 2 heures après un test de tolérance au glucose oral
- BloodPressure : Pression artérielle diastolique (mm Hg)
- SkinThickness : Épaisseur du pli cutané tricipital (mm)
- Insulin : Insuline sérique à 2 heures (mu U/ml)
- BMI : Indice de masse corporelle (poids en kg/(taille en m)^2)
- DiabetesPedigreeFunction : Fonction de pedigree du diabète
- Age : Âge (années)
- Outcome : Variable de classe (0 ou 1) 268 sur 768 sont égales à 1, les autres sont à 0.

1. Chargement des packages :

```

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import rcParams

from sklearn import model_selection
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_auc_score
from sklearn.metrics import f1_score, confusion_matrix, precision_recall_curve, roc_curve
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.preprocessing import StandardScaler

import plotly.express as px
from plotly.subplots import make_subplots
import plotly.graph_objects as go

import warnings
warnings.filterwarnings(action='ignore')

```

Took 9 sec. Last updated by anonymous at December 09 2023, 5:48:26 PM.

Nous allons faire appel PyCaret.

PyCaret est une bibliothèque de machine learning en Python qui simplifie le processus de développement de modèles. Elle offre une interface simple et intuitive pour effectuer rapidement des tâches courantes telles que la préparation des données, la création de modèles, l'évaluation des performances et l'ajustement des hyperparamètres.

Pour plus d'information, vous allez trouver la documentation dans le lien Github suivant :

<https://github.com/pycaret/pycaret>

2. Définition des fonctions d'utilité :

```

def get_clf_eval(y_test, pred=None, pred_proba=None):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    f1 = f1_score(y_test, pred)

    roc_auc = roc_auc_score(y_test, pred_proba)

    # ROC-AUC print
    print('accuracy: {:.4f}, precision: {:.4f}, recall: {:.4f},\nF1: {:.4f}, AUC:{:.4f}'.format(accuracy, precision, recall, f1, roc_auc))
    return confusion

```

Took 0 sec. Last updated by anonymous at December 09 2023, 5:48:26 PM.

get_clf_eval est une fonction permettant d'évaluer les performances des modèles.

3. Lecture et vérification des données :

```

from pyspark.sql import SparkSession
# Créez une session Spark
spark = SparkSession.builder.appName("Diabetes").getOrCreate()

# Spécifiez le chemin HDFS de votre fichier CSV
hdfs_path = "hdfs://namenode:9000/input/diabetes.csv"

# Chargez le fichier CSV dans un DataFrame Spark
diabetes_df_spark = spark.read.csv(hdfs_path, header=True, inferSchema=True)

# Affichez les premières lignes du DataFrame
diabetes_df_spark.show(5)

Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/12/09 16:48:29 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
/opt/conda/envs/python_3_with_R/lib/python3.7/site-packages/pyspark/context.py:317: FutureWarning: Python 3.7 support is deprecated in Spark 3.4.
  warnings.warn("Python 3.7 support is deprecated in Spark 3.4.", FutureWarning)
+-----+-----+-----+-----+-----+
|Pregnancies|Glucose|BloodPressure|SkinThickness|Insulin| BMI|DiabetesPedigreeFunction|Age|Outcome|
+-----+-----+-----+-----+-----+
|      6|   148|        72|       35|  0|33.6|           0.627| 50|     1|
|      1|    85|        66|       29|  0|26.6|           0.351| 31|     0|
|      8|   183|        64|       0|  0|23.3|           0.672| 32|     1|
|      1|    89|        66|       23| 94|28.1|           0.167| 21|     0|
|      0|   137|        40|       35| 168|43.1|           2.288| 33|     1|
+-----+-----+-----+-----+-----+
only showing top 5 rows

```

Pour le visualiser de manière appropriée et pour pouvoir le manipuler, nous allons transformer le dataframe en un dataframe pandas.

```
import missingno as msno
import pandas as pd
import matplotlib.pyplot as plt

# Assuming diabetes_df_spark is your PySpark DataFrame
diabetes_df = diabetes_df_spark.toPandas()

Took 0 sec. Last updated by anonymous at December 09 2023, 5:48:39 PM.

diabetes_df.head().T.style.set_properties(**{'background-color': 'grey',
                                             'color': 'white',
                                             'border-color': 'white'})

0 1 2 3 4
Pregnancies 6.000000 1.000000 8.000000 1.000000 0.000000
Glucose 148.000000 85.000000 183.000000 89.000000 137.000000
BloodPressure 72.000000 66.000000 64.000000 66.000000 40.000000
SkinThickness 35.000000 29.000000 0.000000 23.000000 35.000000
Insulin 0.000000 0.000000 0.000000 94.000000 168.000000
BMI 33.600000 26.600000 23.300000 28.100000 43.100000
DiabetesPedigreeFunction 0.627000 0.351000 0.672000 0.167000 2.288000
Age 50.000000 31.000000 32.000000 21.000000 33.000000
Outcome 1.000000 0.000000 1.000000 0.000000 1.000000
```

Took 1 sec. Last updated by anonymous at December 09 2023, 5:48:40 PM.

```
diabetes_df.rename(columns={"DiabetesPedigreeFunction": "DPF"}, inplace=True)
```

Took 0 sec. Last updated by anonymous at December 09 2023, 5:48:40 PM.

4. Analyse exploratoire des données :

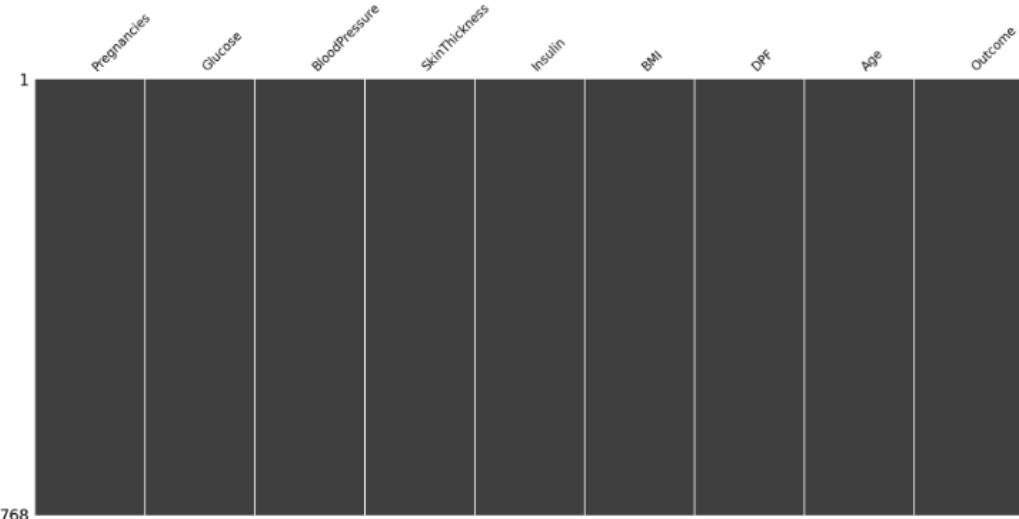
Vérification des valeurs manquantes et des types de données :

```
import missingno as msno
import matplotlib.pyplot as plt
import tempfile
from IPython.display import Image, display

# Assuming diabetes_df is your DataFrame
msno.matrix(diabetes_df)

# Save the plot as an image
temp_image_file = tempfile.NamedTemporaryFile(delete=False, suffix=".png")
plt.savefig(temp_image_file.name, bbox_inches="tight", pad_inches=0.1)
plt.close()

# Display the saved image
display(Image(filename=temp_image_file.name))
```



Parfait ! Il n'y a aucune valeur manquante, et tous les types de fonctionnalités sont numériques. Par conséquent, il n'est pas nécessaire de prétraiter les valeurs manquantes.

Vérifiez le déséquilibre de la cible.

```
from IPython.display import Image, display
import pandas as pd

# Assuming diabetes_df is your DataFrame
colors = ['gold', 'mediumturquoise']
labels = ['0', '1']
values = diabetes_df['Outcome'].value_counts()

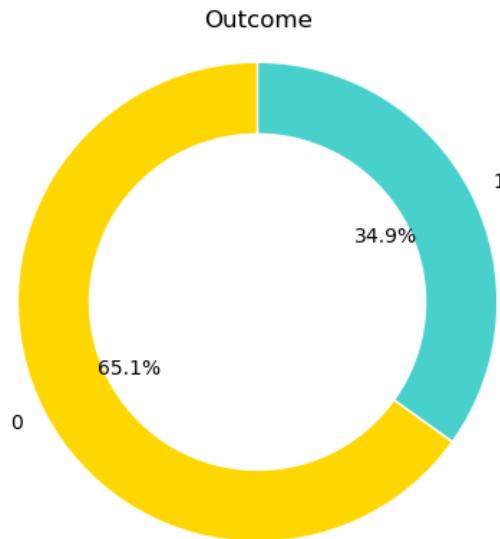
# Create a donut chart using Matplotlib
fig, ax = plt.subplots()
ax.pie(values, labels=labels, autopct='%.1f%%', colors=colors, startangle=90, wedgeprops=dict(width=0.3, edgecolor ='w'))

# Draw a circle in the center to create a donut chart
centre_circle = plt.Circle((0, 0), 0.70, fc='white')
fig.gca().add_artist(centre_circle)

# Set the aspect ratio equal so that pie is drawn as a circle.
ax.set_aspect('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
ax.set_title("Outcome")

# Save the figure
temp_image_file = "/tmp/diabetes_plot.png"
fig.savefig(temp_image_file, bbox_inches="tight", pad_inches=0.1)
plt.close()

# Display the saved image
display(Image(filename=temp_image_file))
```



Vérification des statistiques :

```
import numpy as np
import pandas as pd

def highlight_min(s, props=''):
    return np.where(s == np.nanmin(s.values), props, '')

# Assuming diabetes_df is your DataFrame
diabetes_df.describe().style.apply(highlight_min, props='color:Black;background-color:Grey', axis=0)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DPF	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Took 0 sec. Last updated by anonymous at December 09 2023, 5:48:44 PM.

Parmi les caractéristiques, il y a plusieurs pour lesquelles la valeur min() est 0. Examinons davantage ces caractéristiques.

Vérification et suppression des valeurs aberrantes :

```
feature_names = [cname for cname in diabetes_df.loc[:, : 'Age'].columns]

# Set figure size and other style parameters
plt.rcParams['figure.figsize'] = (40, 80)
sns.set(font_scale=3)
sns.set_style("white")
sns.set_palette("bright")
plt.subplots_adjust(hspace=0.5)

i = 1
for name in feature_names:
    plt.subplot(5, 2, i)
    sns.histplot(data=diabetes_df, x=name, hue="Outcome", kde=True, palette="YlGnBu")

    # Manually adjust the legend font size for each subplot
    plt.legend(title="Outcome", title_fontsize="20", fontsize="15")

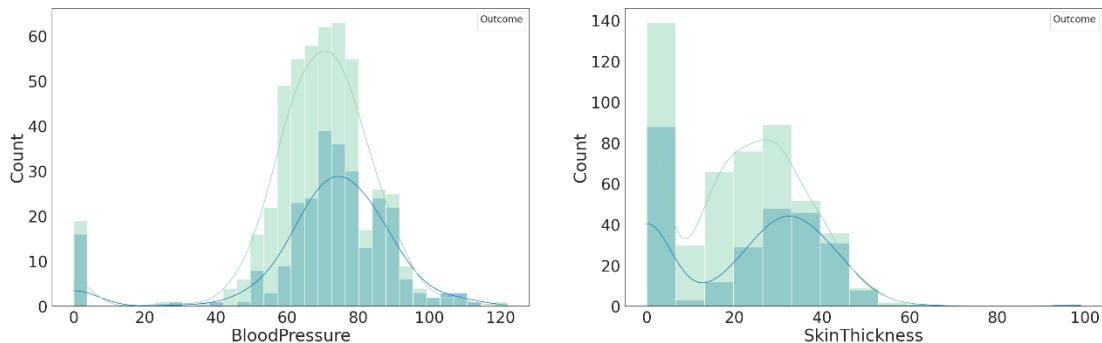
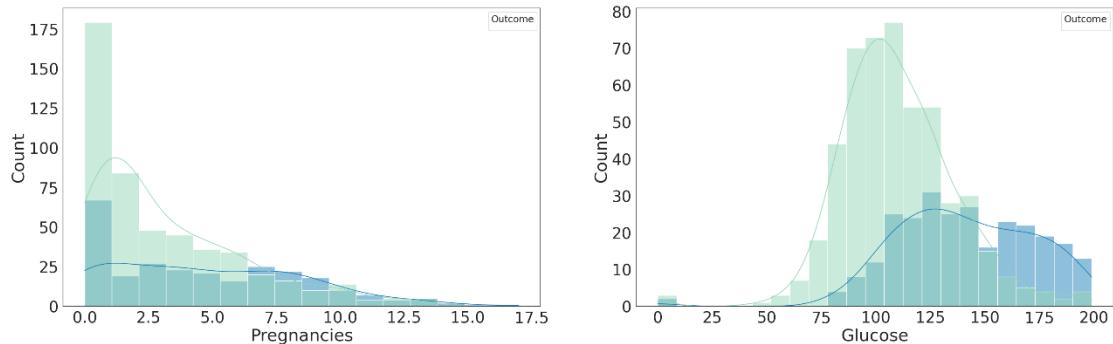
    i += 1

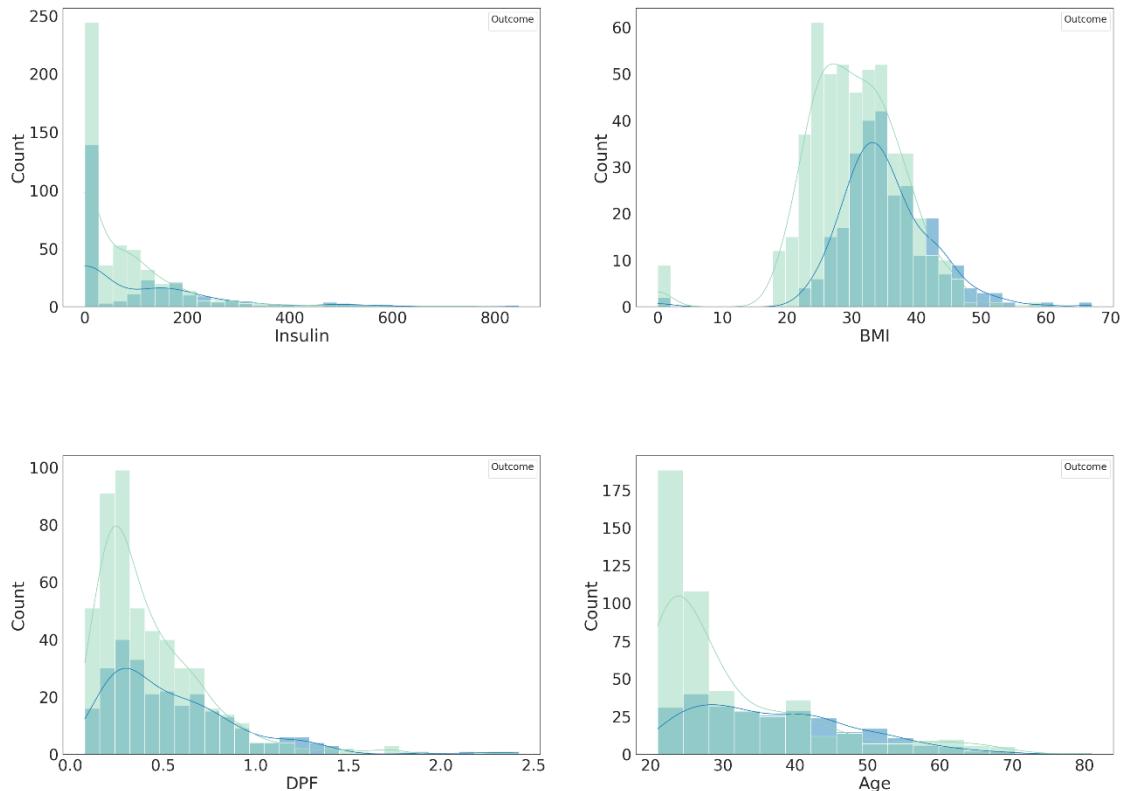
# Save the figure
temp_image_file = "/tmp/diabetes_subplot.png"
plt.savefig(temp_image_file, bbox_inches="tight", pad_inches=1.5)
plt.close()

# Display the saved image
display(Image(filename=temp_image_file))

No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when 1
```

Le Output est :





Comme nous pouvons le voir, il serait bon de changer la valeur zéro de chaque caractéristique par une autre valeur. Tout d'abord, calculons la proportion de valeurs zéro dans chaque caractéristique.

```

zero_features = ['Pregnancies','Glucose','BloodPressure','SkinThickness','Insulin','BMI']
total_count = diabetes_df['Glucose'].count()

for feature in zero_features:
    zero_count = diabetes_df[diabetes_df[feature]==0][feature].count()
    print('{0} 0 number of cases {1}, percent is {2:.2f} %'.format(feature, zero_count, 100*zero_count/total_count))

Pregnancies 0 number of cases 111, percent is 14.45 %
Glucose 0 number of cases 5, percent is 0.65 %
BloodPressure 0 number of cases 35, percent is 4.56 %
SkinThickness 0 number of cases 227, percent is 29.56 %
Insulin 0 number of cases 374, percent is 48.70 %
BMI 0 number of cases 11, percent is 1.43 %

Took 0 sec. Last updated by anonymous at December 09 2023, 8:00:29 PM.

```

Comme le montre les statistiques ci-dessus, le ratio de la valeur zéro dans les caractéristiques SkinThickness et Insulin semble être élevé. Modifions les valeurs correspondantes pour qu'elles correspondent à la valeur moyenne de chaque caractéristique.

Cependant, une valeur nulle peut avoir une signification particulière pour la caractéristique correspondante. Si vous disposez d'un expert ayant des connaissances en diabète, vous pourrez confirmer que votre décision est correcte. Cependant, étant donné qu'il n'y a pas de connaissances spécifiques dans ce domaine, nous allons d'abord remplacer les valeurs correspondantes par la valeur moyenne.

```
| diabetes_mean = diabetes_df[zero_features].mean()
| diabetes_df[zero_features]=diabetes_df[zero_features].replace(0, diabetes_mean)
```

Took 0 sec. Last updated by anonymous at December 09 2023, 8:00:29 PM.

```
| X = diabetes_df.iloc[:, :-1]
| y = diabetes_df.iloc[:, -1]
```

Took 0 sec. Last updated by anonymous at December 09 2023, 8:00:29 PM.

Si nous supprimons la valeur zéro de chaque caractéristique, nous obtenons une distribution similaire à la distribution normale. Par conséquent, effectuons une mise à l'échelle linéaire et une mise à l'échelle standard.

5. Mise à l'échelle (Scaling) :

Bien que les valeurs zéro de chaque caractéristique soient converties en valeurs moyennes, certaines caractéristiques ont une forme unilatérale. Par conséquent, nous avons décidé d'effectuer une mise à l'échelle non linéaire et avons opté pour l'utilisation de QuantileTransformer, qui modifie la distribution la plus proche de la distribution normale en se référant au cahier de notes référencé ci-dessous.

La fonction de quantile classe ou lisse la relation entre les observations et peut être mise en correspondance avec d'autres distributions, telles que la distribution uniforme ou normale.

```
| from sklearn.preprocessing import QuantileTransformer
| scaler = QuantileTransformer(n_quantiles=100, random_state=0, output_distribution='normal')
| X_scaled = scaler.fit_transform(X)
```

Took 0 sec. Last updated by anonymous at December 09 2023, 8:00:29 PM.

```
| X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

Took 0 sec. Last updated by anonymous at December 09 2023, 8:00:29 PM.

```
| def highlight_min(s, props=''):
|     return np.where(s == np.nanmin(s.values), props, '')
```

```
| X_train.describe().style.apply(highlight_min, props='color:Black;background-color:Grey', axis=0)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DPF	Age
count	576.000000	576.000000	576.000000	576.000000	576.000000	576.000000	576.000000	576.000000
mean	4.315407	121.709154	72.144653	26.341978	118.600098	32.315027	0.468618	32.921875
std	2.916151	30.080570	12.113756	9.158041	94.570101	6.878494	0.339325	11.507539
min	1.000000	44.000000	24.000000	8.000000	14.000000	18.200000	0.078000	21.000000
25%	2.000000	100.000000	64.000000	20.536458	79.799479	27.275000	0.240000	24.000000
50%	3.845052	118.000000	72.000000	23.000000	79.799479	32.000000	0.372500	29.000000
75%	6.000000	138.250000	80.000000	32.000000	127.500000	36.325000	0.612250	40.000000
max	17.000000	199.000000	122.000000	63.000000	846.000000	67.100000	2.420000	81.000000

Took 0 sec. Last updated by anonymous at December 09 2023, 8:00:29 PM.

```

import seaborn as sns
import matplotlib.pyplot as plt
from IPython.display import Image, display

# Set figure size and other style parameters
plt.rcParams['figure.figsize'] = (40, 80)
sns.set(font_scale=3)
sns.set_style("white")
sns.set_palette("bright")
plt.subplots_adjust(hspace=0.5)

# Assuming X is the features table
feature_names = X.columns

# Create subplots
i = 1
for name in feature_names:
    plt.subplot(5, 2, i)
    sns.histplot(data=diabetes_df, x=name, hue="Outcome", kde=True, palette="YlGnBu")

# Manually adjust the legend font size for each subplot
plt.legend(title="Outcome", title_fontsize="20", fontsize="15")

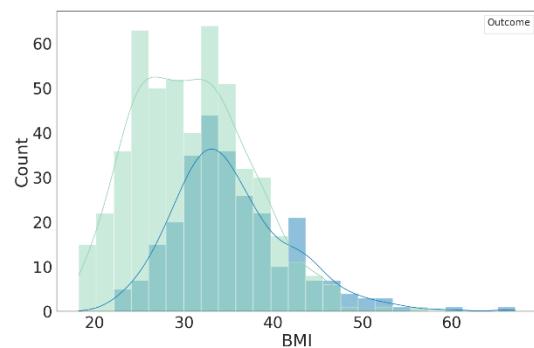
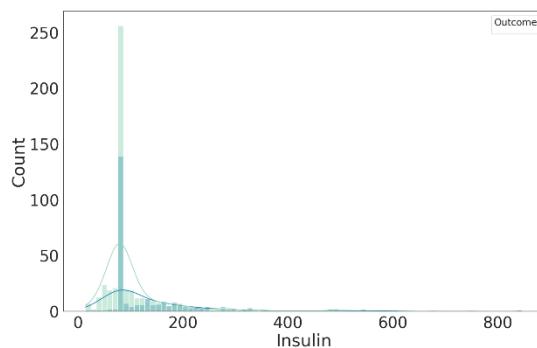
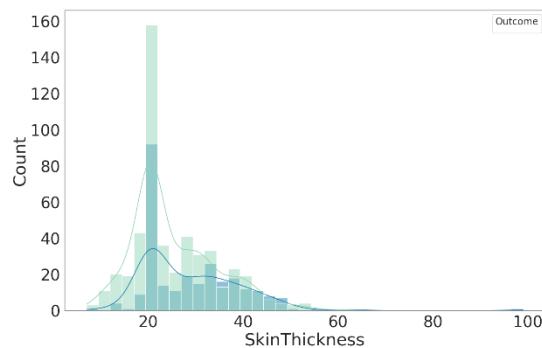
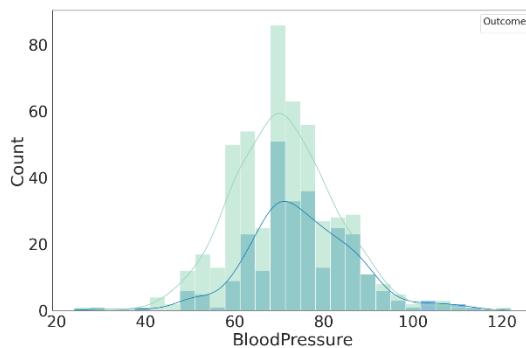
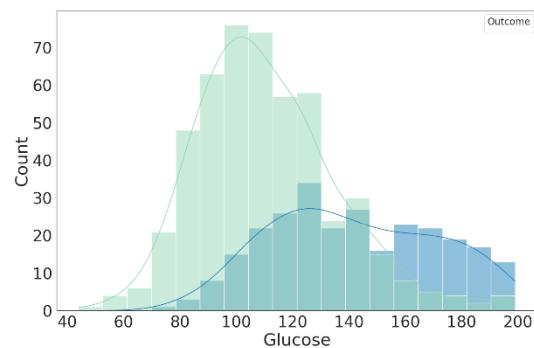
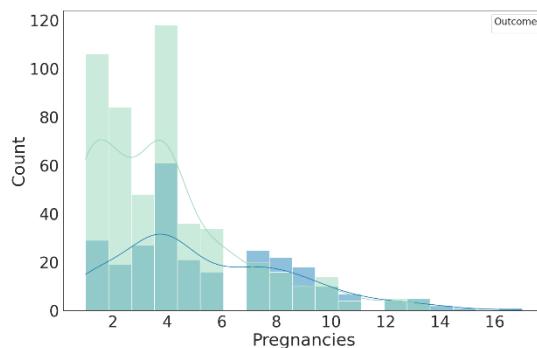
i += 1

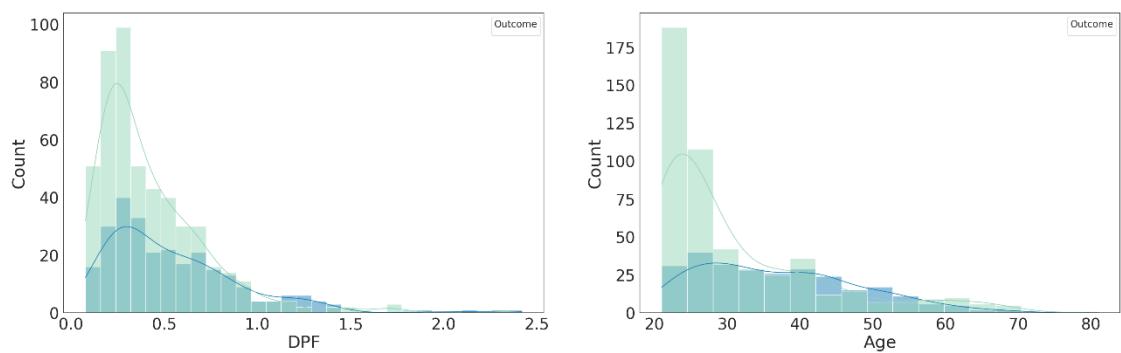
# Save the figure
temp_image_file = "/tmp/diabetes_subplot.png"
plt.savefig(temp_image_file, bbox_inches="tight", pad_inches=1.5)
plt.close()

# Display the saved image
display(Image(filename=temp_image_file))

```

Nous pouvons visualiser les graphiques comme suit.





6. Vérification de la corrélation entre les caractéristiques :

```

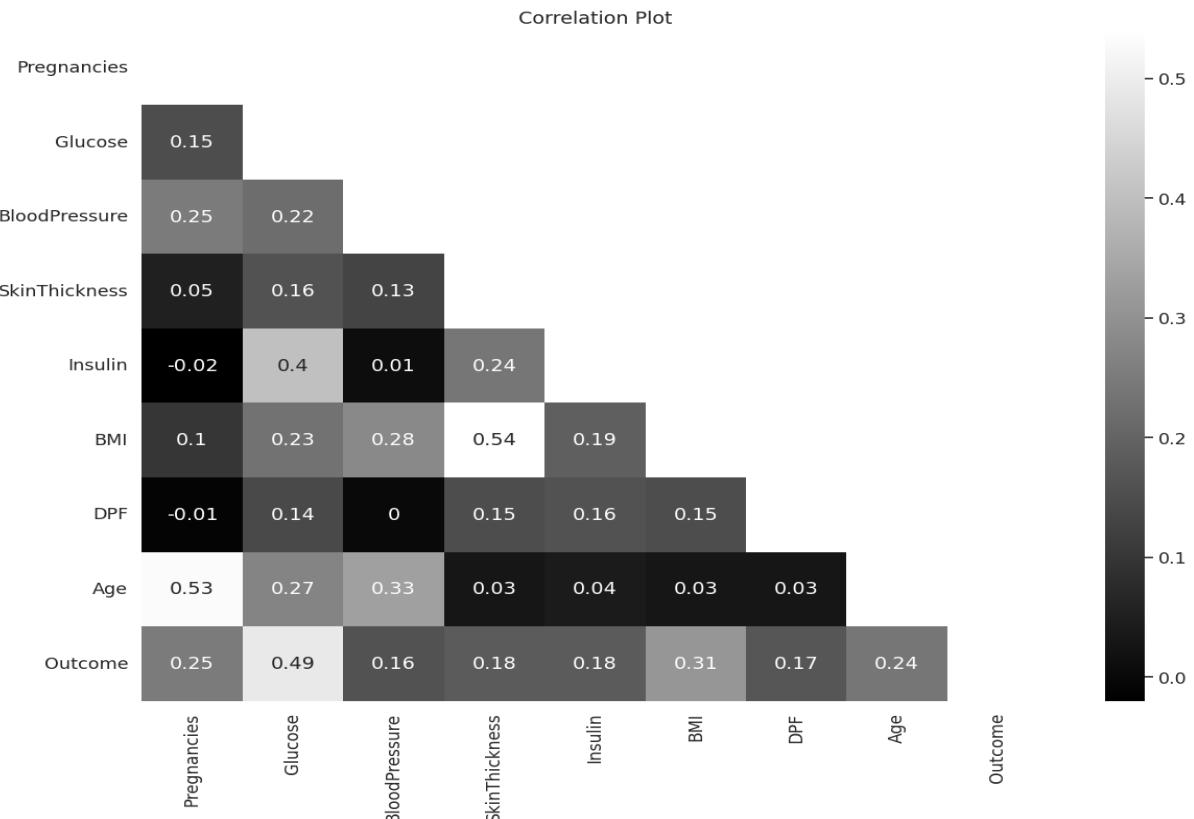
corr = diabetes_df.corr().round(2)

sns.set(font_scale=1.15)
plt.figure(figsize=(14, 10))
sns.set_palette("bright")
sns.set_style("white")
mask = np.zeros_like(corr)
mask[np.triu_indices_from(mask)] = True
sns.heatmap(corr, annot=True, cmap='gist_yang_r', mask=mask, cbar=True)
plt.title('Correlation Plot')

# Save the figure
temp_image_file = "/tmp/diabetes_plot.png"
plt.savefig(temp_image_file, bbox_inches="tight", pad_inches=0.1)
plt.close()

# Display the saved image
display(Image(filename=temp_image_file))

```



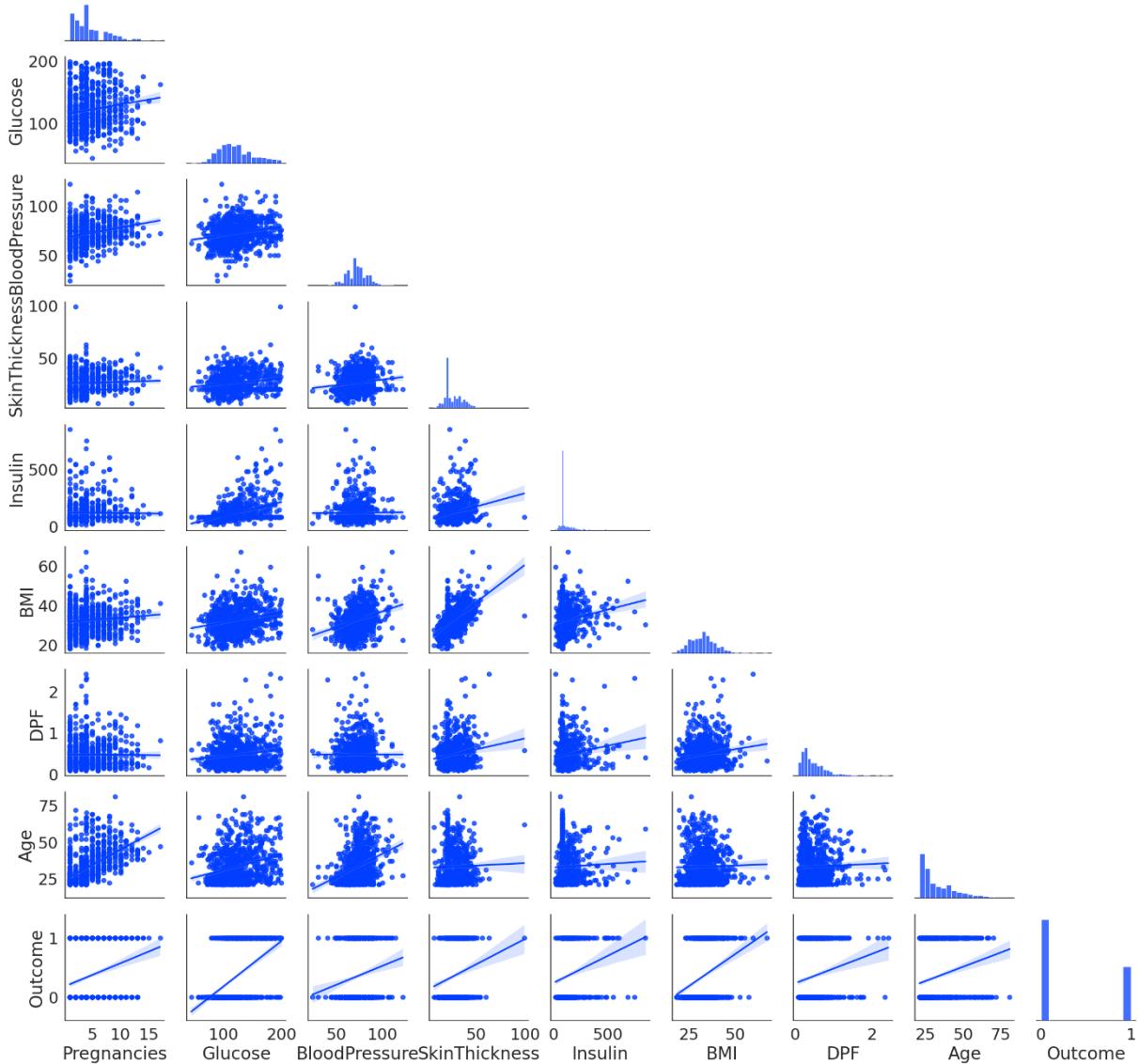
```

sns.set(font_scale=2)
plt.figure(figsize=(10, 8))
sns.set_style("white")
sns.set_palette("bright")
sns.pairplot(diabetes_df, kind='reg', corner=True, palette='YlGnBu')

# Save the figure
temp_image_file = "/tmp/diabetes_plot.png"
plt.savefig(temp_image_file, bbox_inches="tight", pad_inches=0.1)
plt.close()

# Display the saved image
display(Image(filename=temp_image_file))

```



7. Division des ensembles d'entraînement et de test :

```

X_train = diabetes_df.drop('Outcome', axis=1)
y_train = diabetes_df['Outcome']

```

Took 0 sec. Last updated by anonymous at December 09 2023, 8:00:49 PM.

8. Ensemble :

```
| from pycaret.classification import *
Task Over. Last updated by anonymous at December 09 2023, 8:28:19 PM.

clf1 = setup(data = diabetes_df,
             target = 'Outcome',
             preprocess = False)
Task 1 sec. Last updated by anonymous at December 09 2023, 8:28:17 PM.

top5 = compare_models(sort='AIC',
                      n_select = 5,
                      exclude=['lightgbm','xgboost','dummy','svm','ridge','knn','dt','nb','ada'])

FINISHED ⏺ ✎ 📈 ⏹
```

```
Initiated ..... 19:28:17
Status ..... Loading Dependencies
Estimator ..... Compiling Library
Processing: 0% | 0/3 [00:00]
Task 30 sec. Last updated by anonymous at December 09 2023, 8:29:07 PM.
```

9. Création des modèles :

```
from pycaret.classification import create_model
# Rename the variable to avoid conflict with the catboost library
catboost_model = create_model('catboost')
rf = create_model('rf')
lr = create_model('lr')
lda = create_model('lda')
gbc = create_model('gbc')

# Interpret the CatBoost model
interpret_model(catboost_model)
```



```
| interpret_model(rf)

Glucose
BMI
Age
DPF
Pregnancies
SkinThickness
Insulin
BloodPressure
```

SHAP summary plot for the Random Forest model. The x-axis represents the SHAP value (impact on model output) ranging from -0.2 to 0.3. The y-axis lists the features: Glucose, BMI, Age, DPF, Pregnancies, SkinThickness, Insulin, and BloodPressure. A color scale on the right indicates the Feature value, ranging from Low (blue) to High (red). Most features show a positive impact on the output, with Glucose having the highest impact.

Took 1 sec. Last updated by anonymous at December 09 2023, 8:29:23 PM.

Observation :

- Comme prévu, le glucose est utilisé comme la caractéristique la plus importante.
- SkinThickness et BloodPressure ont une faible importance.

10. Ajustement des hyperparamètres :

```
# Import necessary libraries
from pycaret.classification import create_model, tune_model, interpret_model

# Rename the variable to avoid conflict with the catboost library
catboost_model = create_model('catboost')

# Other models
rf = create_model('rf')
lr = create_model('lr')
lda = create_model('lda')
gbc = create_model('gbc')

# Tune the models
tuned_catboost = tune_model(catboost_model, optimize='AUC')
tuned_rf = tune_model(rf, optimize='AUC')
tuned_lr = tune_model(lr, optimize='AUC')
tuned_lda = tune_model(lda, optimize='AUC')
tuned_gbc = tune_model(gbc, optimize='AUC')
```

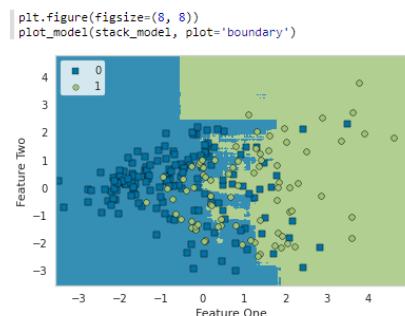
11. Stacking

```
stack_model = stack_models(estimator_list = top5, meta_model = top5[0],optimize = 'AUC')
```

```
Initiated ..... 19:32:13
Status ..... Loading Dependencies
Estimator ..... Compiling Library
```

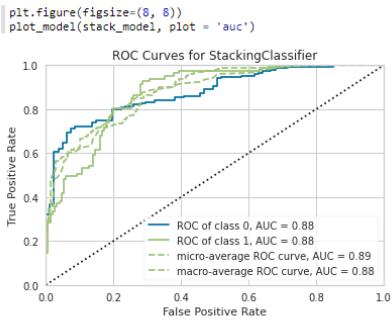
Processing: 0% | 0/6 [00:00]

Took 1 min 48 sec. Last updated by anonymous at December 09 2023, 8:34:01 PM.



Took 5 sec. Last updated by anonymous at December 09 2023, 8:34:07 PM.

Dans le cas d'un modèle de stacking, il peut y avoir du surajustement dans certains cas et du sous-ajustement dans d'autres cas.



Took 1 sec. Last updated by anonymous at December 09 2023, 8:34:08 PM.

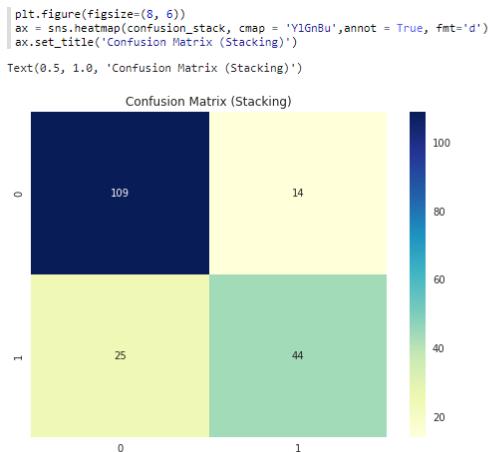
```

#prediction
pred = stack_model.predict(X_test)
pred_proba = stack_model.predict_proba(X_test)[:,1]
#Accuracy
confusion_stack = get_clf_eval(y_test,pred,pred_proba)

accuracy: 0.7969, precision: 0.7586, recall: 0.6377,      F1: 0.6929, AUC:0.8711

```

Took 0 sec. Last updated by anonymous at December 09 2023, 8:34:08 PM.



Took 1 sec. Last updated by anonymous at December 09 2023, 8:34:09 PM.

12. Soft voting :

```

#prediction
pred = blend_soft.predict(X_test)
pred_proba = blend_soft.predict_proba(X_test)[:,1]
#accuracy
confusion_soft = get_clf_eval(y_test,pred,pred_proba)

accuracy: 0.9115, precision: 0.9062, recall: 0.8406,      F1: 0.8722, AUC:0.9591

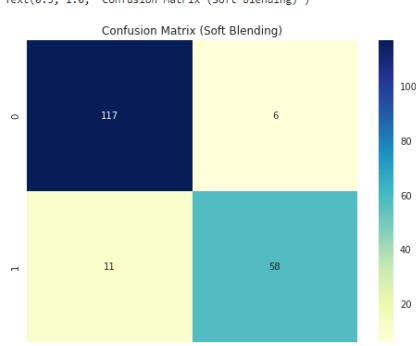
```

Took 1 sec. Last updated by anonymous at December 09 2023, 8:34:36 PM.

```

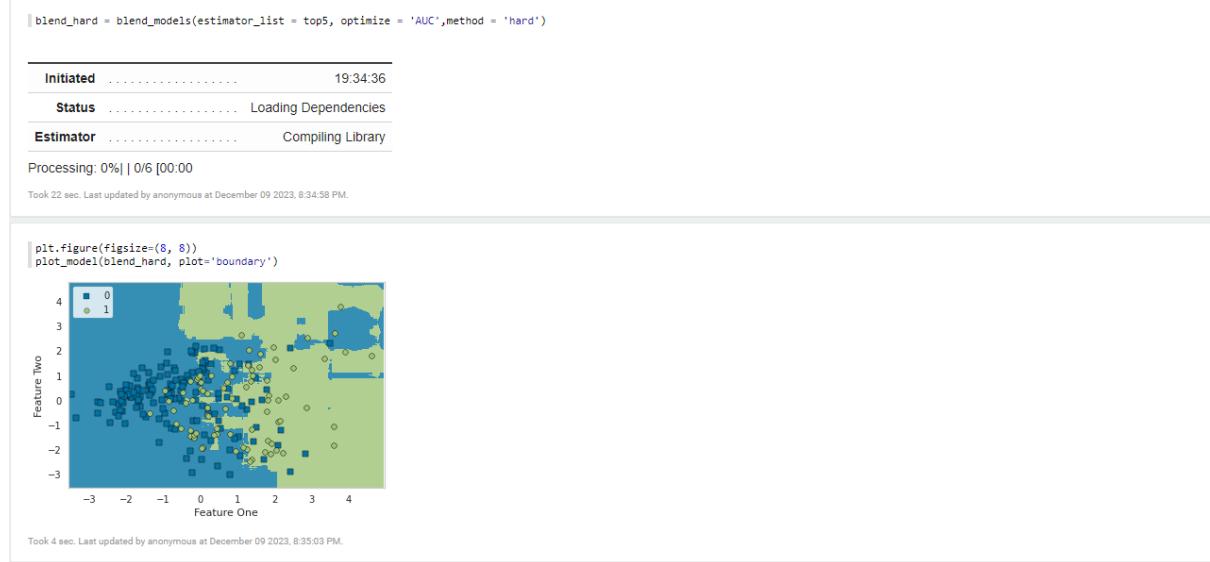
plt.figure(figsize=(8, 6))
ax = sns.heatmap(confusion_soft, cmap = 'YlGnBu', annot = True, fmt='d')
ax.set_title('Confusion Matrix (Soft Blending)')

```

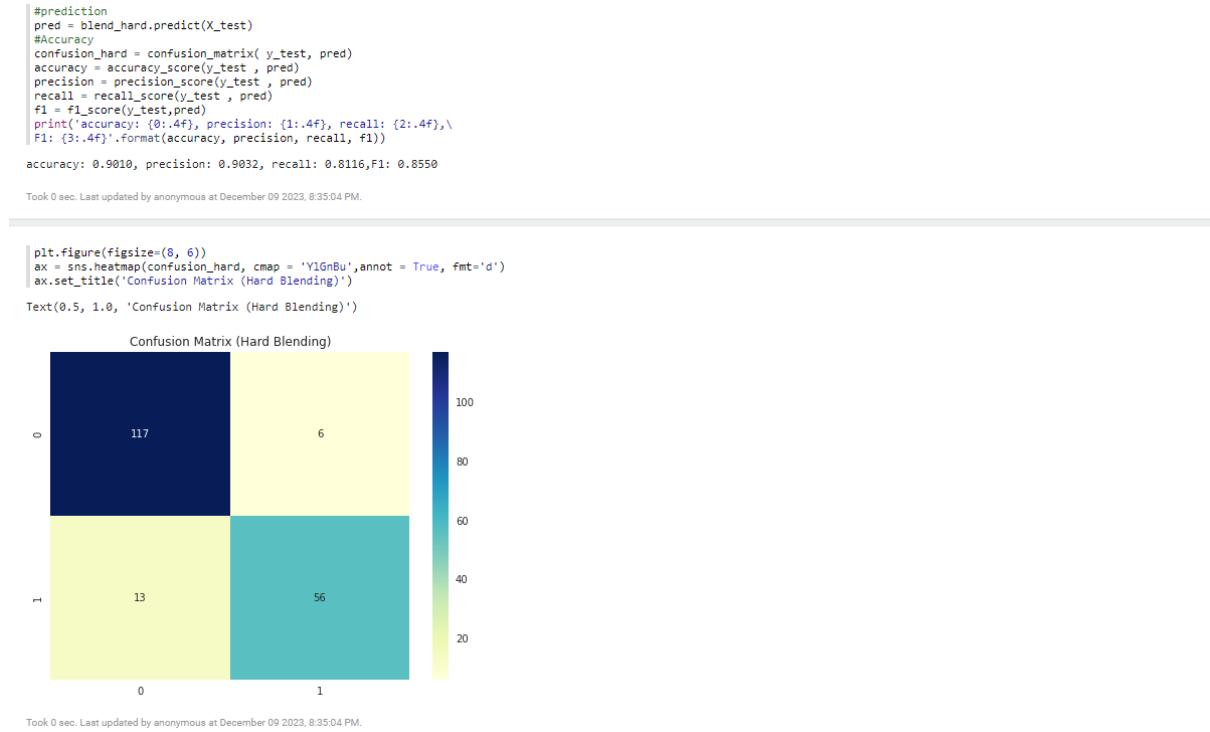


Il semble que le modèle ait été correctement entraîné. La matrice de confusion est bien équilibrée et les résultats semblent bons.

13. Hard voting :



Il semble avoir bien entrainné.



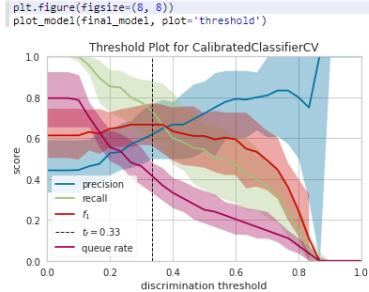
Le résultat n'est pas mauvais. Cependant, il est moins bon que le résultat du soft voting.

14. Calibrating & finalizing the final model :

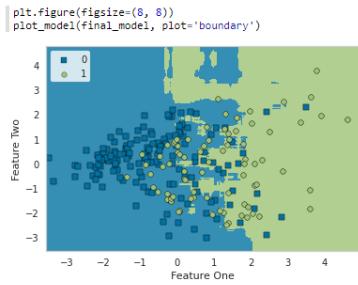
```
| cali_model = calibrate_model(blend_soft)

Initiated ..... 19:35:04
Status ..... Loading Dependencies
Estimator ..... Compiling Library
Processing: 0% | 0/6 [00:00]
Took 2 min 0 sec. Last updated by anonymous at December 09 2023, 8:37:04 PM.
```

```
| final_model = finalize_model(cali_model)
Took 9 sec. Last updated by anonymous at December 09 2023, 8:37:13 PM.
```



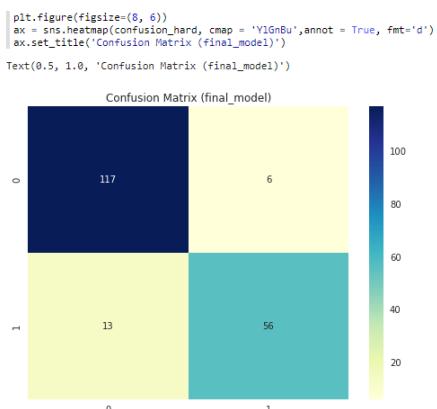
Took 7 min 19 sec. Last updated by anonymous at December 09 2023, 8:44:32 PM.



Took 10 sec. Last updated by anonymous at December 09 2023, 8:44:42 PM.

```
#prediction
pred = final_model.predict(X_test)
accuracy = accuracy_score(y_test, pred)
precision = precision_score(y_test, pred)
recall = recall_score(y_test, pred)
f1 = f1_score(y_test, pred)
print('accuracy: {0:.4f}, precision: {1:.4f}, recall: {2:.4f},\nF1: {3:.4f}'.format(accuracy, precision, recall, f1))
accuracy: 0.9323, precision: 0.9516, recall: 0.8551, F1: 0.9008
```

Took 2 sec. Last updated by anonymous at December 09 2023, 8:44:44 PM.



Took 0 sec. Last updated by anonymous at December 09 2023, 8:44:44 PM.

Après l'EDA et le prétraitement, trois modèles d'ensemble ont été exécutés et les performances ont été vérifiées avec l'ensemble de validation. L'ensemble utilisant le vote mou (soft voting) et le vote dur (hard voting) a donné le meilleur résultat pour résoudre ce problème, mais des résultats différents peuvent être obtenus en fonction du prétraitement, de la sélection des modèles de base et des paramètres d'hyperparamètres.

On ajoute maintenant le xgboost :

```
import seaborn as sns
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc

y = diabetes_df['Outcome']
x = diabetes_df.drop(['Outcome'], axis=1)
X_train, X_val, y_train, y_val = train_test_split(
    x, y, test_size=0.2, random_state=42)

model = XGBClassifier(
    n_estimators=100, max_depth=20, learning_rate=0.1,
    use_label_encoder=False, verbosity=0,
    scale_pos_weight=1.0
)
model.fit(
    X_train, y_train,
    eval_set=[(X_train, y_train), (X_val, y_val)],
    early_stopping_rounds=8,
    verbose=True
)
model.save_model('xgb.pkl')

y_val_pred = model.predict_proba(X_val)[:, 1]

print('XGB classifier saved')

# Additional metrics
pred = model.predict(X_val)
accuracy = accuracy_score(y_val, pred)
precision = precision_score(y_val, pred)
recall = recall_score(y_val, pred)
f1 = f1_score(y_val, pred)

print('Accuracy: {:.4f}'.format(accuracy))
print('Precision: {:.4f}'.format(precision))
print('Recall: {:.4f}'.format(recall))
print('F1 Score: {:.4f}'.format(f1))

Accuracy: 0.9993
Precision: 0.9534
Recall: 0.8796
F1 Score: 0.8757
```

Le xgboost a donné une meilleure performance que les autres algorithmes ensemblistes.

2. Créations du modèle sérialisé

Maintenant nous allons générer le modèle sérialisé, pour se faire, nous créeront un nouveau notebook là où nous diviserons les étapes en fonction :

1. Les importations :

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
from typing import List
```

- **pandas, numpy, matplotlib, seaborn** : pour la manipulation des données et la visualisation.
- **XGBClassifier de XGBoost** : pour créer et entraîner un modèle XGBoost.
- **train_test_split de sklearn.model_selection** : pour diviser les données en ensembles d'entraînement et de validation.
- **roc_curve et auc de sklearn.metrics** : pour évaluer les performances du modèle.
- **List de typing** : pour spécifier le type d'annotations.

2. Fonction train_model() :

```

def train_model() -> None:
    """Trains an XGBClassifier to predict diabetes.
    """
    data = get_samples()
    y = data['Outcome']
    x = data.drop(['Outcome'], axis=1)
    x_train, x_val, y_train, y_val = train_test_split(
        x, y, test_size=0.2, random_state=101
    )

    model = XGBClassifier(
        n_estimators=100, max_depth=20, learning_rate=0.1,
        use_label_encoder=False, verbosity=0,
        scale_pos_weight=1.8
    )
    model.fit(
        x_train, y_train,
        eval_set=[(x_train, y_train), (x_val, y_val)],
        eval_metric='auc',
        early_stopping_rounds=8,
        verbose=True
    )
    # Save the trained XGBoost model
    model.save_model('xgb.pkl')

    y_val_pred = model.predict_proba(x_val)[:, 1]
    plot_roc(y_true=y_val, y_pred=y_val_pred)
    plot_features(x_train.columns, model.feature_importances_)
    print('XGB classifier saved')

```

- Charge les données en appelant la fonction `get_samples()`.
- Divise les données en ensembles d'entraînement et de validation.
- Crée et entraîne un modèle **XGBoost** avec des hyperparamètres spécifiés.
- Sauvegarde le modèle entraîné au format pickle (`xgb.pkl`).
- Utilise le modèle pour prédire les probabilités sur l'ensemble de validation.

3. Fonction `predict_diabetes_probability(data: List[float])` :

```

def predict_diabetes_probability(data: List[float], verbose=True) -> float:
    """Returns the model's predicted probability of diabetes from the given
    physiological data.

    Args:
        data (List[float]): A list of the physiological data.

    Returns:
        float: Predicted probability of diabetes,
    """
    print('Predicting...\\n' if verbose else "", end="")
    xgb_model = XGBClassifier()
    print('Loading model...\\n' if verbose else "", end="")
    xgb_model.load_model('xgb.pkl')
    print('Making Prediction...\\n' if verbose else "", end="")
    prediction = xgb_model.predict_proba(np.array(data).reshape(1, -1))[0][1]
    print(f'Predicted probability {round(prediction*100)} %\\n' if verbose else "", end="")
    return prediction

```

- Charge le modèle XGBoost entraîné à partir du fichier pickle.
- Utilise le modèle pour prédire la probabilité de diabète à partir des données fournies.

4. Fonction `get_samples()` :

```

def get_samples() -> pd.DataFrame:
    """Get samples from CSV.

    Returns:
        pd.DataFrame: samples from the dataset.
    """
    from pyspark.sql import SparkSession

    # Créez une session Spark
    # Assurez-vous que l'adresse du serveur Spark est correcte
    spark = SparkSession.builder.appName("Diabetes").getOrCreate()

    # Spécifiez le chemin HDFS de votre fichier CSV
    hdfc_path = "hdfs://namenode:9000/input/diabetes.csv"

    try:
        # Chargez le fichier CSV dans un DataFrame Spark
        diabetes_df_spark = spark.read.csv(hdfc_path, header=True, inferSchema=True)

        # Affichez les premières lignes du DataFrame
        diabetes_df_spark.show(5)
    except Exception as e:
        print(f"Une erreur s'est produite : {e}")

    data = diabetes_df_spark.toPandas()
    return data

```

- Charge les données depuis un fichier CSV à l'aide de Spark.
- Affiche les premières lignes du DataFrame Spark.
- Convertit le DataFrame Spark en un DataFrame pandas.

5. Bloc principal (`__main__`) :

```

if __name__ == "__main__":
    train_model()

```

Appelle la fonction `train_model()` lorsqu'il est exécuté en tant que script.

Le output après l'execussion :

```

+-----+-----+-----+-----+-----+-----+-----+
|Pregnancies|Glucose|BloodPressure|SkinThickness|Insulin| BMI|DiabetesPedigreeFunction|Age|Outcome|
+-----+-----+-----+-----+-----+-----+-----+
|      6|   148|       72|       35|    0|33.6|           0.627| 50|     1|
|      1|    85|       66|       29|    0|26.6|           0.351| 31|     0|
|      8|   183|       64|       0|    0|23.3|           0.672| 32|     1|
|      1|    89|       66|       23| 94|28.1|           0.167| 21|     0|
|      0|   137|       40|       35| 168|43.1|           2.288| 33|     1|
+-----+-----+-----+-----+-----+-----+-----+
only showing top 5 rows

[0] validation_0-auc:0.95190      validation_1-auc:0.78469
[1] validation_0-auc:0.96381      validation_1-auc:0.79164
[2] validation_0-auc:0.97265      validation_1-auc:0.80440
[3] validation_0-auc:0.97528      validation_1-auc:0.80145
[4] validation_0-auc:0.97949      validation_1-auc:0.80497
[5] validation_0-auc:0.98299      validation_1-auc:0.81020
...  ...  ...  ...  ...  ...
[17] validation_0-auc:0.99907      validation_1-auc:0.81401
[18] validation_0-auc:0.99920      validation_1-auc:0.81572
[19] validation_0-auc:0.99930      validation_1-auc:0.81363
[20] validation_0-auc:0.99945      validation_1-auc:0.81649

XGB classifier saved

```

Vérifions le répertoire Docker de zeppelin :

roc_curve.png	ADDED	21.2 kB	38 minutes ago	-rw-r--r--
> run			2 years ago	drwxrwxr-x
> scripts			2 years ago	drwxr-xr-x
> webapps			1 hour ago	drwxr-xr-x
xgb.hdf5	MODIFIED	103.5 kB	17 hours ago	-rw-r--r--
xgb.pkl	ADDED	103.5 kB	32 minutes ago	-rw-r--r--
zeppelin-web-0.10.1.war		28.3 MB	2 years ago	-rw-r--r--
zeppelin-web-angular-0.10.1.war		28.5 MB	2 years ago	-rw-r--r--

Ceci ne sont que les fonctions principales, vous trouverez les notebooks complets et les codes fourni dans les figures dans le lien github (voir annexe)

3. Création de l'API

Une fois le modèle sérialisé est récupéré, on va créer notre première API en utilisant PyCharm.

Nous allons, en premier lieu créer un projet python de type FastAPI, et nous copierons le pickle dans la racine du projet :

The screenshot shows the PyCharm interface with the following details:

- Project Structure:** The project is named "ProjectDiabetes". It contains a "venv" folder (marked as "library root"), "api.py", "Dockerfile", "logs.log", "main.py", "models.py", "requirements.txt", and "xgb_model.pkl".
- Code Editor:** The "api.py" file is open. The code uses FastAPI to create a simple API endpoint. It imports pickle, uvicorn, and FastAPI, along with numpy and joblib. It loads a model from "xgb_model.pkl" and defines two endpoints: a GET endpoint that returns "bonjour" and a POST endpoint for predictions.

```
Project ~
  ProjectDiabetes C:\User...
    > static
    > templates
    > venv library root
      api.py
      Dockerfile
      logs.log
      main.py
      models.py
      requirements.txt
      xgb_model.pkl
    > External Libraries
    > Scratches and Consoles

  main.py
  models.py
  them.css
  index.html
  requirements.txt
  api.py
```

```
1 import pickle
2 import uvicorn
3 from fastapi import FastAPI
4 import numpy as np # Add this import
5
6 from models import Diabetes
7
8 app = FastAPI()
9 import joblib
10
11 # ...
12
13 # Chargement du modèle
14 model_path = "xgb_model.pkl" # Mettez le chemin correct s'il est différent
15 model = joblib.load(open(model_path, "rb"))
16
17 @app.get("/")
18 def great():
19     return {"message": "bonjour"}
20
21 2 usages (2 dynamic)
22 @app.post("/predict")
23 def predict(req: Diabetes):
24     preg = req.Pregnancies
25     glucose = req.Glucose
```

Figure III-2: API diabète

Dans ce code nous avons fait appel au model.pkl, ainsi que nous avons créer api predict comme le montre la figure 3 :

The screenshot shows a code editor with a dark theme. The tab bar at the top includes files: main.py, models.py, them.css, index.html, requirements.txt, and api.py (which is currently selected). The code in api.py is as follows:

```
21 @app.post("/predict")
22 def predict(req: Diabetes):
23     preg = req.Pregnancies
24     glucose = req.Glucose
25     bp = req.BloodPressure
26     skinthickness = req.SkinThickness
27     insulin = req.Insulin
28     bmi = req.BMI
29     dpf = req.DPF
30     age = req.Age
31     features = np.array([preg, glucose, bp, skinthickness, insulin, bmi, dpf, age]).reshape(1, -1) # Convert to N
32     prediction = model.predict(features)[0]
33     probab = model.predict_proba(features)
34
35     if prediction == 1:
36         return {"ans": f"You have been tested positive with {probab[0][1]} probability"}
37     else:
38         return {"ans": f"You have been tested negative with {probab[0][0]} probability"}
39
40 if __name__ == "__main__":
41     uvicorn.run(app)
42
```

Figure III-3: API Predict

Vous allez constater un objet `model`, ce n'est que l'objet contenant les paramètres de diabète.

The screenshot shows a code editor with a dark theme. The tab bar at the top includes files: main.py, models.py (which is currently selected), them.css, index.html, Dockerfile, requirements.txt, and api.py. The code in models.py is as follows:

```
1 from pydantic import BaseModel
2
3 class Diabetes(BaseModel):
4     Pregnancies:int
5     Glucose:int
6     BloodPressure:int
7     SkinThickness:int
8     Insulin:int
9     BMI:float
10    DPF:float
11    Age:int
```

Figure III-4: Modèle diabetes

On va lancer maintenant l'API avec la commande :

```
uvicorn api:app --reload
```

Pregnancies : 2,
 "Glucose": 23,
 "BloodPressure": 12,
 "SkinThickness": 13,
 "Insulin": 14,
 "BMI": 12,
 "DPF": 0,
 "Age": 50
}`

Request URL
<http://127.0.0.1:8000/predict>

Server response

Code	Details
200	Response body <pre>{ "ans": "You have been tested negative with 0.686874270439148 probability" }</pre>

[Copy](#) [Download](#)

Figure III-5: Test de l'API

Le modèle fonctionne correctement. Ce qui reste est d'automatiser le pipeline.

IV. Automatisation du pipeline avec Airflow

Apache Airflow est une plateforme open source de gestion de flux de travail (workflow) développée principalement par Airbnb. Elle offre une approche programmable pour orchestrer et automatiser des tâches complexes dans le domaine du traitement des données

C'est une solution puissante pour automatiser et orchestrer des flux de travail complexes, offrant une flexibilité, une extensibilité et une visibilité accrues dans le traitement des données et les opérations par lots.



1. Création des DAG

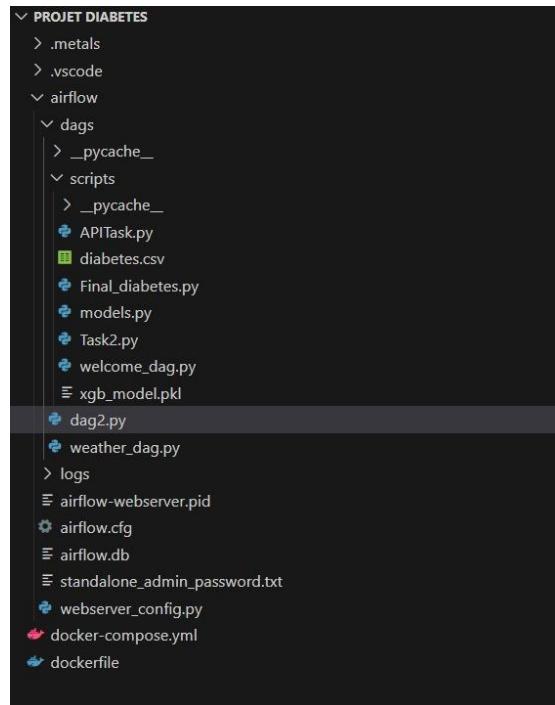
Un DAG, ou Directed Acyclic Graph (en français, graphe orienté acyclique), est un concept central dans Apache Airflow. Dans le contexte d'Airflow, un DAG représente un ensemble de tâches interdépendantes organisées dans un graphe orienté.

Ce code définit un DAG (Directed Acyclic Graph) dans Apache Airflow nommé 'test_download1000' avec deux tâches : une tâche FileSensor et une tâche PythonOperator. Le DAG est programmé pour s'exécuter toutes les 5 minutes :

1. La Structure du projet :

Le projet doit être sous la structure suivante, le dossier dags contient tous les code qu'on devra utiliser, cela facilitera l'accès de airflow aux fichiers.

- APITask.py contient le code de notre API.
- Final_diabetes contient le code du training.
- Models.py et la class contenant les features.
- dag2.py est celui contenant les taches automatisées.



Les changements apportés aux fichiers par rapport à ce qu'on avait précédemant et comme démontrer dans la figure qui suit :

```
'           '
print('XGB classifier saved')
# Enregistrez le modèle avec joblib
joblib.dump(model, '/opt/airflow/dags/scripts/xgb_model.pkl')
print(['XGB classifier saved as xgb_model.pkl'])
y_val_pred = model.predict_proba(x_val)[:, 1]

print('XGB classifier saved')
```

Figure IV-1: final_diabetes.py

Comme vous pouvez remarquer on a ajouté le path là où on va enregistrer notre modèle sérialisé.

2. Le code du DAG :

Tous d'abord il faut installer le package airflow avec la commande suivante :

```
pip install apache-airflow
```

```

1  from datetime import datetime, timedelta
2  from airflow.utils.dates import days_ago
3  from airflow import DAG
4  from airflow.operators.python import PythonOperator
5  from airflow.sensors.filesystem import FileSensor
6  import requests
7  from pytz import timezone
8  from airflow.operators.bash import BashOperator
9  from airflow.operators.python import BranchPythonOperator
10 import os
11 from datetime import datetime
12 from subprocess import run

```

Figure IV-2: Configuration et Importation des Modules

- Configuration des dates et du fuseau horaire : Importation des modules nécessaires pour manipuler les dates, les durées et les fuseaux horaires.
- Importation des modules Airflow : Importation des modules et opérateurs essentiels d'Apache Airflow.
- Requests : Importation du module requests (même s'il n'est pas utilisé dans le code fourni).
- OS et Subprocess : Importation des modules pour interagir avec le système d'exploitation et exécuter des processus en sous-shell.

```

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

dag = DAG(
    dag_id: 'Diabetes_App3',
    default_args=default_args,
    description='An Airflow DAG to automate the pipeline',
    schedule_interval='0 */1 * * *', # Run every 5 minutes
    max_active_runs=1, # Ensure only one run at a time
    catchup=False, # Do not run backfill for the intervals between start_date and the current date
)

```

Figure IV-3: Arguments par Défaut et Initialisation du DAG

- Configuration du fuseau horaire : Création d'un objet de fuseau horaire pour Casablanca (casablanca_tz).
- Arguments par Défaut : Définition des paramètres par défaut pour le DAG, y compris le propriétaire, la date de début et les configurations de réessai.
- Initialisation du DAG : Création d'un DAG nommé 'Diabetes_App3' avec les paramètres spécifiés, y compris une description, un intervalle de planification, le nombre maximal d'exécutions actives et les réglages de rattrapage.

```

# Define the path to the file to be monitored
file_path = '/opt/airflow/dags/scripts/diabetes.csv' # Replace with the actual path

# Create a FileSensor task to check for changes
file_sensor_task = FileSensor(
    task_id='file_sensor',
    filepath=file_path,
    poke_interval=60, # Check every 60 seconds
    timeout=600, # Timeout after 600 seconds (10 minutes)
    mode='poke',
    soft_fail=True,
    dag=dag,
)

```

Figure IV-4: Tâche de Détection de Fichier

- Chemin du Fichier : Définition du chemin vers le fichier à surveiller (file_path).
- Tâche de Détection de Fichier : Création d'une tâche FileSensor nommée 'file_sensor' qui vérifie les modifications du fichier spécifié toutes les 60 secondes (poke_interval). Il a un délai d'attente de 10 minutes (timeout) et fonctionne en mode "poke". Il est défini pour échouer doucement, ce qui signifie qu'il continuera avec les tâches en aval même si le fichier n'est pas trouvé.

```

check_file_task = BranchPythonOperator(
    task_id='check_file_modification',
    python_callable=check_file_modification,
    provide_context=True,
    dag=dag,
)

```

Figure IV-5: Tâche de Vérification de Modification de Fichier

- BranchPythonOperator : Création d'un BranchPythonOperator nommé 'check_file_modification'. Cet opérateur exécutera la fonction Python check_file_modification et décidera quelle branche suivre en fonction du résultat retourné.

```

python_script_path = '/opt/airflow/dags/scripts/Final_diabetes.py'
python_task = BashOperator(
    task_id='execute_python_script',
    bash_command=f'python {python_script_path}',
    dag=dag,
)

```

Figure IV-6: Tâche d'Exécution de Script Python

- Chemin du Script Python : Définition du chemin vers le script Python à exécuter (python_script_path).
- BashOperator : Création d'un BashOperator nommé 'execute_python_script' pour exécuter le script Python à l'aide de l'attribut bash_command. Le chemin du script Python est inclus dans la commande.

```
skip_python_task = BashOperator(
    task_id='skip_python_script',
    bash_command='echo "No modification in the last 5 minutes. Skipping script execution"',
    dag=dag,
)
```

Figure IV-7: Tâche de Saut d'Exécution de Script Python

- BashOperator pour le Saut : Création d'un BashOperator nommé 'skip_python_script' pour afficher un message s'il n'y a pas eu de modification dans le fichier au cours des 5 dernières minutes, indiquant que l'exécution du script Python est ignorée.

```
api_task = PythonOperator(
    task_id='execute_api_task',
    python_callable=execute_api_task,
    provide_context=True,
    dag=dag,
)
```

Figure IV-8: Tâche d'Exécution de l'Application FastAPI

- PythonOperator pour FastAPI : Création d'un PythonOperator nommé 'execute_api_task' pour exécuter la fonction Python execute_api_task. Cette fonction est supposée exécuter un script d'application FastAPI (APITask.py) en utilisant uvicorn sur l'hôte 0.0.0.0 et le port 8000.

```
file_sensor_task >> api_task >> check_file_task
check_file_task >> [python_task, skip_python_task]...
```

Figure IV-9: Dépendances entre les Tâches

- Dépendances entre les Tâches : Spécification des dépendances entre les tâches en utilisant l'opérateur >>.
- file_sensor_task déclenche api_task, qui, à son tour, déclenche check_file_task.
- check_file_task déclenche soit python_task ou skip_python_task en fonction du résultat de la vérification de la modification du fichier.

Pour lancer le dag, il suffit de lancer ce fichier python, et par la suite vérifier dans l'interface airflow sa présence sous le nom qui lui a été affecté dans le script.

2. Résultats d'exécution

On n'a rien changé donc, l'api n'a pas été exécutée :

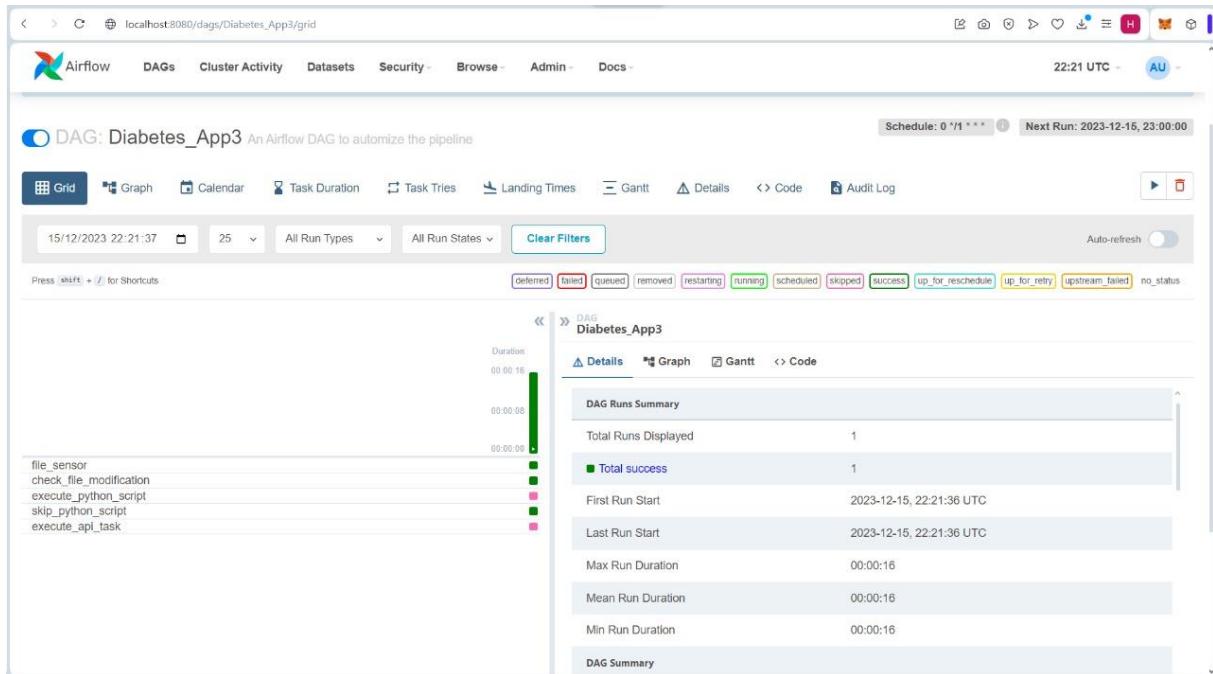


Figure IV-10: Interface airflow

V. Création de l'interface avec Flask

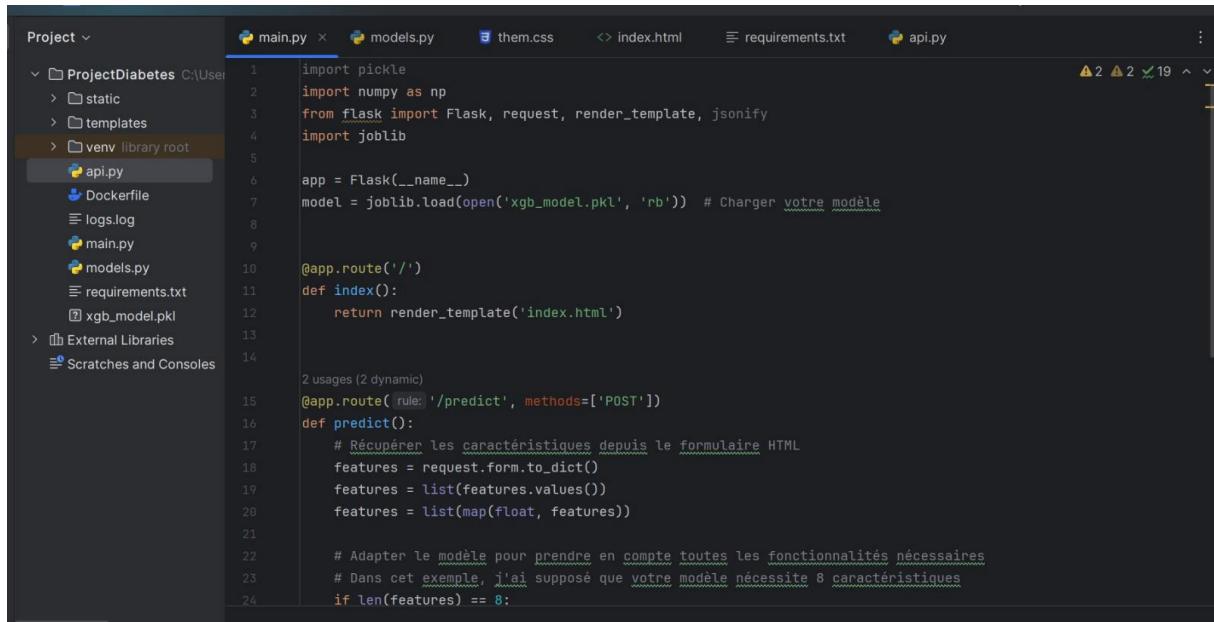
L'API est fonctionnelle, ce qui reste est de créer l'interface avec laquelle on interrogera l'API :

```
main.py models.py them.css index.html requirements.txt api.py

21 @app.post("/predict")
22 def predict(req: Diabetes):
23     preg = req.Pregnancies
24     glucose = req.Glucose
25     bp = req.BloodPressure
26     skinthickness = req.SkinThickness
27     insulin = req.Insulin
28     bmi = req.BMI
29     dpf = req.DPF
30     age = req.Age
31     features = np.array([preg, glucose, bp, skinthickness, insulin, bmi, dpf, age]).reshape(1, -1) # Convert to N
32     prediction = model.predict(features)[0]
33     probab = model.predict_proba(features)
34
35     if prediction == 1:
36         return {"ans": f"You have been tested positive with {probab[0][1]} probability"}
37     else:
38         return {"ans": f"You have been tested negative with {probab[0][0]} probability"}
39
40     if __name__ == "__main__":
41         uvicorn.run(app)
42
```

Figure V-1: API Fast du projet Diabetes

Le main :

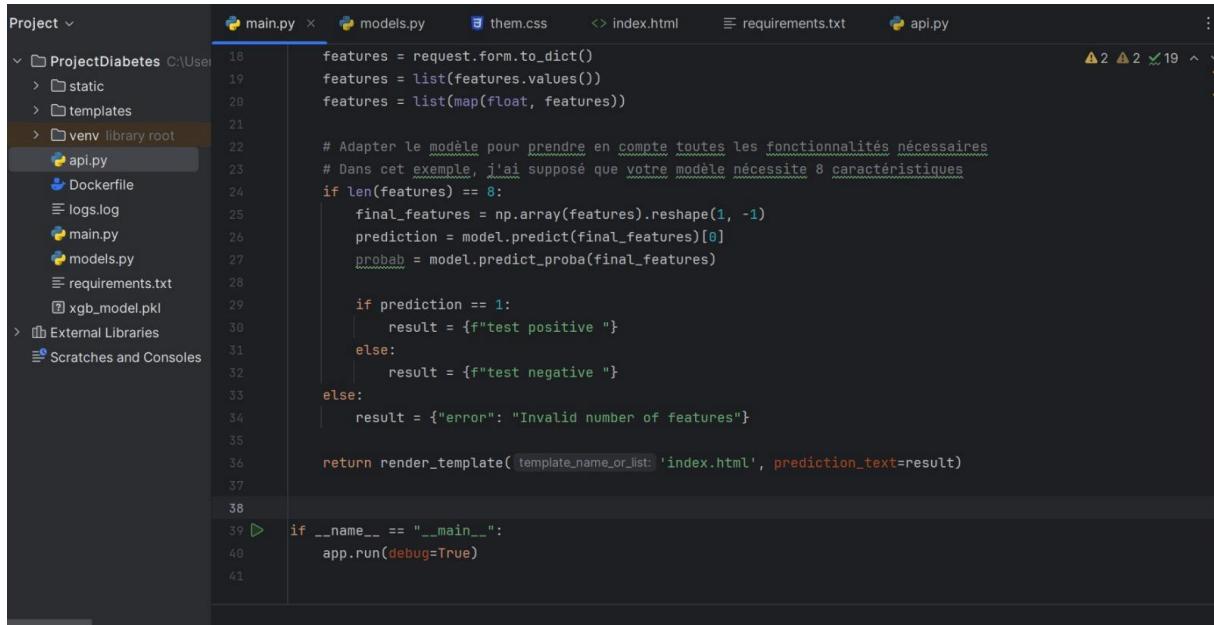


```
Project v main.py x models.py them.css index.html requirements.txt api.py
ProjectDiabetes C:\User
> static
> templates
> venv library root
api.py Dockerfile logs.log main.py models.py requirements.txt xgb_model.pkl
External Libraries Scratches and Consoles

1 import pickle
2 import numpy as np
3 from flask import Flask, request, render_template, jsonify
4 import joblib
5
6 app = Flask(__name__)
7 model = joblib.load(open('xgb_model.pkl', 'rb')) # Charger votre modèle
8
9
10 @app.route('/')
11 def index():
12     return render_template('index.html')
13
14
15 # 2 usages (2 dynamic)
16 @app.route('/predict', methods=['POST'])
17 def predict():
18     # Récupérer les caractéristiques depuis le formulaire HTML
19     features = request.form.to_dict()
20     features = list(features.values())
21     features = list(map(float, features))
22
23     # Adapter le modèle pour prendre en compte toutes les fonctionnalités nécessaires
24     # Dans cet exemple, j'ai supposé que votre modèle nécessite 8 caractéristiques
25     if len(features) == 8:
```

Figure V-2: Main du flask

Définition des routes dans le main :



```
Project v main.py x models.py them.css index.html requirements.txt api.py
ProjectDiabetes C:\User
> static
> templates
> venv library root
api.py Dockerfile logs.log main.py models.py requirements.txt xgb_model.pkl
External Libraries Scratches and Consoles

18 features = request.form.to_dict()
19 features = list(features.values())
20 features = list(map(float, features))
21
22 # Adapter le modèle pour prendre en compte toutes les fonctionnalités nécessaires
23 # Dans cet exemple, j'ai supposé que votre modèle nécessite 8 caractéristiques
24 if len(features) == 8:
25     final_features = np.array(features).reshape(1, -1)
26     prediction = model.predict(final_features)[0]
27     probab = model.predict_proba(final_features)
28
29     if prediction == 1:
30         result = {"test positive"}
31     else:
32         result = {"test negative"}
33     else:
34         result = {"error": "Invalid number of features"}
35
36     return render_template(template_name_or_list='index.html', prediction_text=result)
37
38
39 if __name__ == "__main__":
40     app.run(debug=True)
```

Figure V-3 : Suite du Main

Cette fois nous allons créer l'index.html dans le répertoire template :

```
4     <meta charset="UTF-8">
5     <meta http-equiv="X-UA-Compatible" content="IE=edge">
6     <meta name="viewport" content="width=device-width, initial-scale=1.0">
7     <title>Diabetes Prediction</title>
8     <link rel="stylesheet" type="text/css" href="/static/them.css">
9 </head>
10 <body>
11     <h1>Diabetes Prediction</h1>
12     <form action="/predict" method="post">
13         <label>Pregnancies:
14             <input type="number" name="Pregnancies" required>
15         </label><br>
16
17         <label>Glucose:
18             <input type="number" name="Glucose" required>
19         </label><br>
20
21         <label>Blood Pressure:
22             <input type="number" name="BloodPressure" required>
23         </label><br>
24
25         <label>Skin Thickness:
26             <input type="number" name="SkinThickness" required>
27         </label><br>
```

Figure V-4: fichier html Index

La suite est la suivante :

```
31     </label><br>
32
33     <label>BMI:
34         <input type="number" name="BMI" required>
35     </label><br>
36
37     <label>Diabetes Pedigree Function:
38         <input type="number" name="DPF" required>
39     </label><br>
40
41     <label>Age:
42         <input type="number" name="Age" required>
43     </label><br>
44
45         <input type="submit" value="Predict">
46     </form>
47     <div class="prediction-result">
48         <p>{{ prediction_text }}</p>
49     </div>
50
51 </body>
52 </html>
```

Figure V-5: Suite du fichier html Index

Pour plus de dynamité et de créativité, nous allons ajouter le fichier CSS (voir figure 6)

```
/* style.css */

body {
    font-family: 'Arial', sans-serif;
    background-color: #f0f8ff; /* AliceBlue */
    margin: 0;
    padding: 0;
    text-align: center;
}

h1 {
    color: #333;
}

form {
    width: 50%; /* Adjusted width to 50% */
    margin: 20px auto;
    padding: 20px;
    background-color: #ffffff; /* White */
    border-radius: 10px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

label {
    display: block;
```

Figure V-6: Fichier de style css

En lançant l'application, vous devriez pouvoir voir l'interface suivante dans le lien :

127.0.0.1:5000

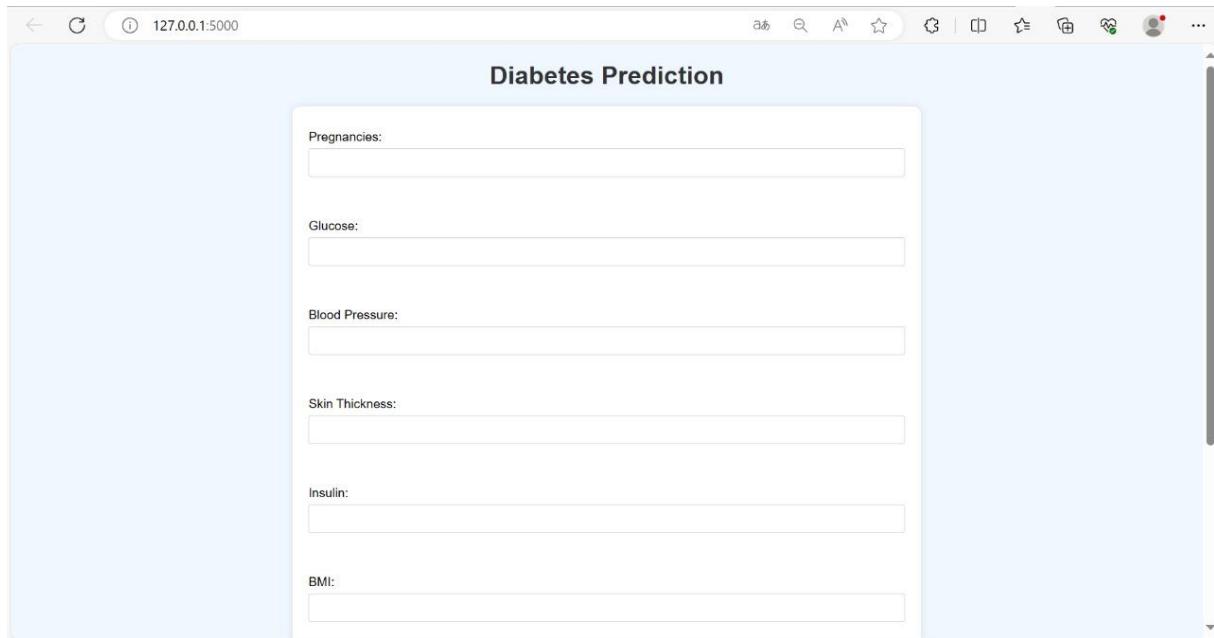


Figure V-7: Interface de l'application

The screenshot shows a web-based application interface. At the top, there is a header bar with icons for back, forward, search, and other browser functions. Below the header, the URL '127.0.0.1:5000' is visible. The main content area contains a form with six input fields. Each field has a label above it and a corresponding input box below. The labels are: 'Blood Pressure.', 'Skin Thickness.', 'Insulin.', 'BMI.', 'Diabetes Pedigree Function.', and 'Age.'. Below the input fields is a large green rectangular button with the word 'Predict' in white text.

Figure V-8: Interface de l'application

On test part des valeurs aléatoires, on devra voir le résultat qui s'affiche comme suivant :

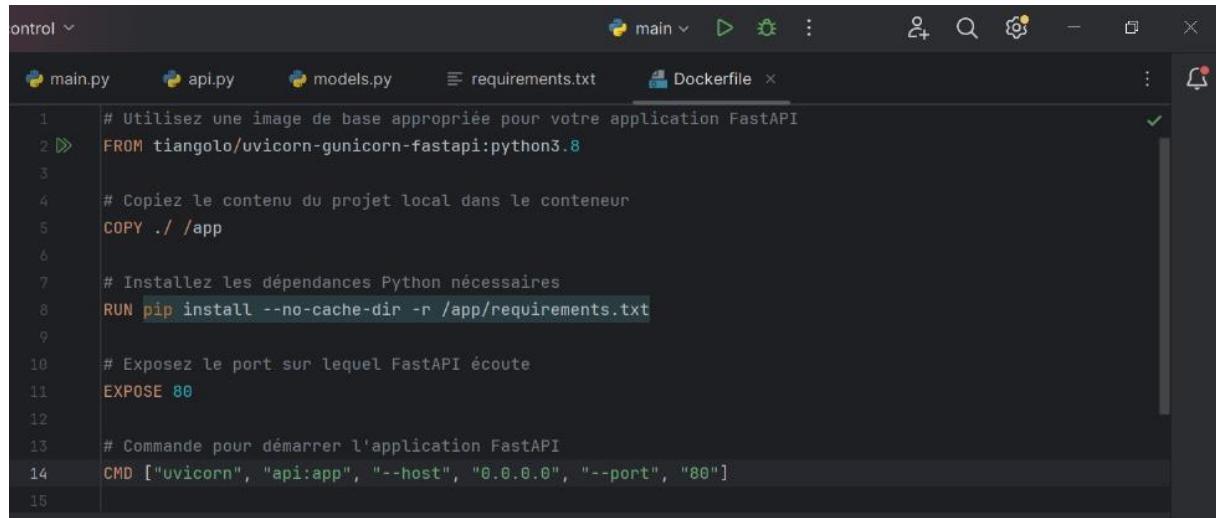
This screenshot shows the same application interface as Figure V-8, but with different input values. The 'Diabetes Pedigree Function' field now contains '0' and the 'Age' field contains '50'. The green 'Predict' button is still present. At the bottom of the page, there is some text in a light blue box that reads '{'test negative '}'.

Figure V-9: Le résultat du test

VI. Conteneurisation de l'API

Pour conteneuriser l'API, nous devrons suivre les étapes suivantes (vous trouvez le lien de documentation *FastAPI in Containers – Docker* dans l'annexe)

1. Créer le fichier docker file dans la racine du projet :



```
1 # Utilisez une image de base appropriée pour votre application FastAPI
2 > FROM tiangolo/uvicorn-gunicorn-fastapi:python3.8
3
4 # Copiez le contenu du projet local dans le conteneur
5 COPY ./ /app
6
7 # Installez les dépendances Python nécessaires
8 RUN pip install --no-cache-dir -r /app/requirements.txt
9
10 # Exposez le port sur lequel FastAPI écoute
11 EXPOSE 80
12
13 # Commande pour démarrer l'application FastAPI
14 CMD ["uvicorn", "api:app", "--host", "0.0.0.0", "--port", "80"]
15
```

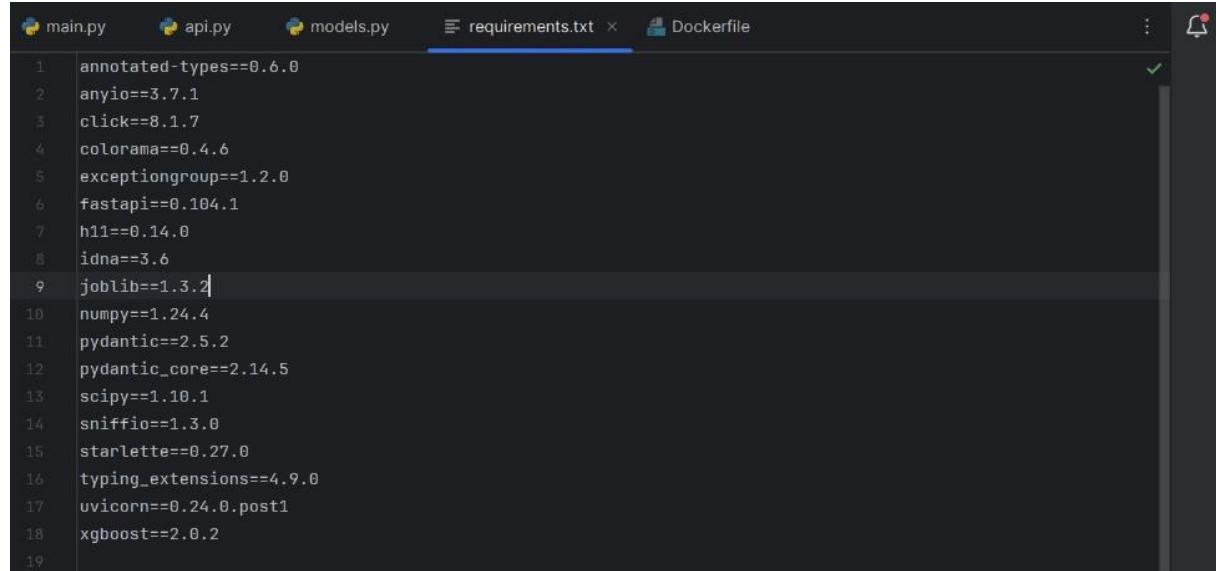
Figure VI-1: docker file

2. Créer le fichier requirements.txt dans le même emplacement :

Pour se faire, nous devrons lancer la commande depuis le terminal du projet

```
pip freeze > requirements.txt
```

Cette commande va remplir automatiquement le fichier de requirements.txt avec les packages utiles du projet :



```
1 annotated-types==0.6.0
2 anyio==3.7.1
3 click==8.1.7
4 colorama==0.4.6
5 exceptiongroup==1.2.0
6 fastapi==0.104.1
7 h11==0.14.0
8 idna==3.6
9 joblib==1.3.2
10 numpy==1.24.4
11 pydantic==2.5.2
12 pydantic_core==2.14.5
13 scipy==1.10.1
14 sniffio==1.3.0
15 starlette==0.27.0
16 typing_extensions==4.9.0
17 uvicorn==0.24.0.post1
18 xgboost==2.0.2
19
```

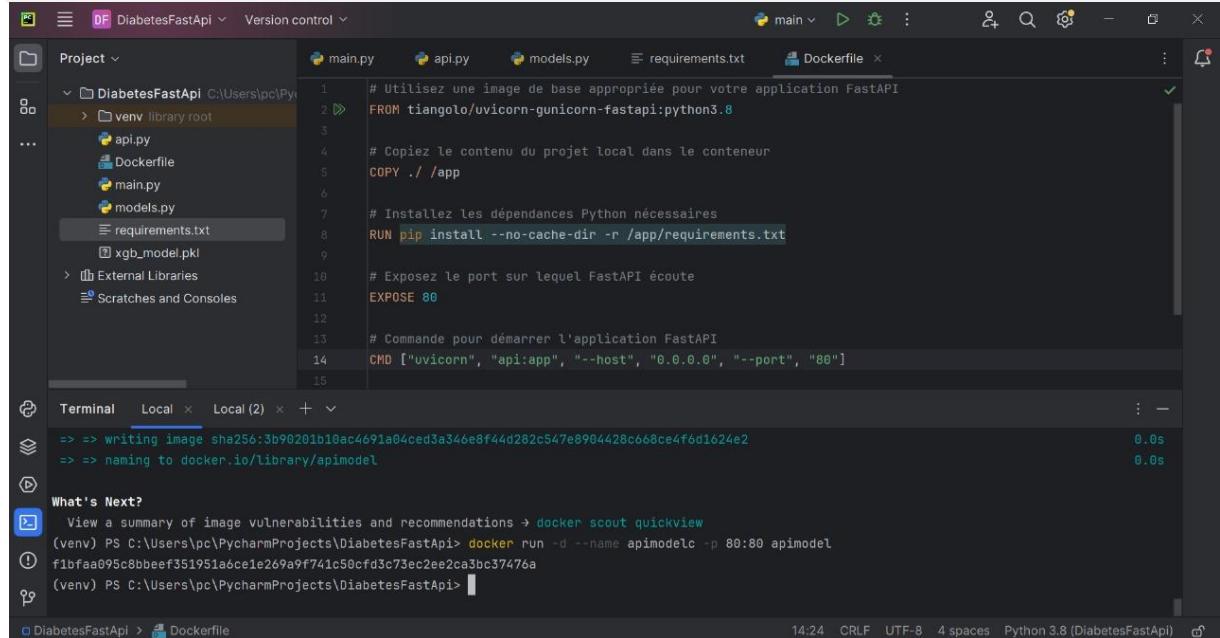
Figure VI-2: requirements.txt

Pour lancer le chargement de l'image, tapez la commande suivante dans le terminal du projet :

```
docker build -t apimodel .
```

Quand le chargement prend fin, on lance le conteneur de l'image avec la commande suivante :

```
docker run -d --name apimodelc -p 80:80 apimodel
```



The screenshot shows the PyCharm IDE interface. The top navigation bar includes tabs for 'main', 'Dockerfile', and other project files. The left sidebar shows a 'Project' view with files like 'main.py', 'api.py', 'models.py', 'requirements.txt', and 'Dockerfile'. The 'Dockerfile' tab is selected, displaying the following content:

```
1 # Utilisez une image de base appropriée pour votre application FastAPI
2 FROM tiangolo/unicorn-gunicorn-fastapi:python3.8
3
4 # Copiez le contenu du projet local dans le conteneur
5 COPY ./app
6
7 # Installez les dépendances Python nécessaires
8 RUN pip install --no-cache-dir -r /app/requirements.txt
9
10 # Exposez le port sur lequel FastAPI écoute
11 EXPOSE 80
12
13 # Commande pour démarrer l'application FastAPI
14 CMD ["uvicorn", "api:app", "--host", "0.0.0.0", "--port", "80"]
```

The terminal below shows the command being run and its output:

```
=> => writing image sha256:3b90201b10ac4691a04ced3a346e8f44d282c547e8904428c668ce4f6d1624e2
=> => naming to docker.io/library/apimodel
```

The status bar at the bottom indicates the file is Python 3.8 (DiabetesFastApi).

Figure VI-3: Lancement du container

Le conteneur doit s'afficher dans cet état comme le montre la figure 3 :

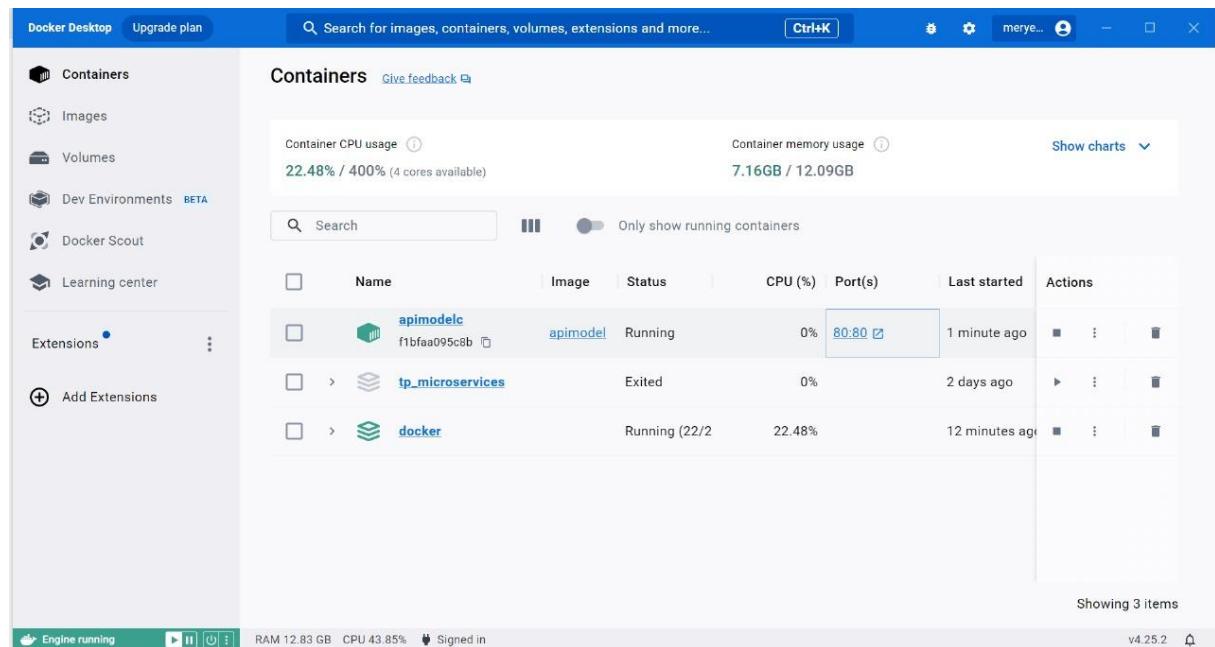
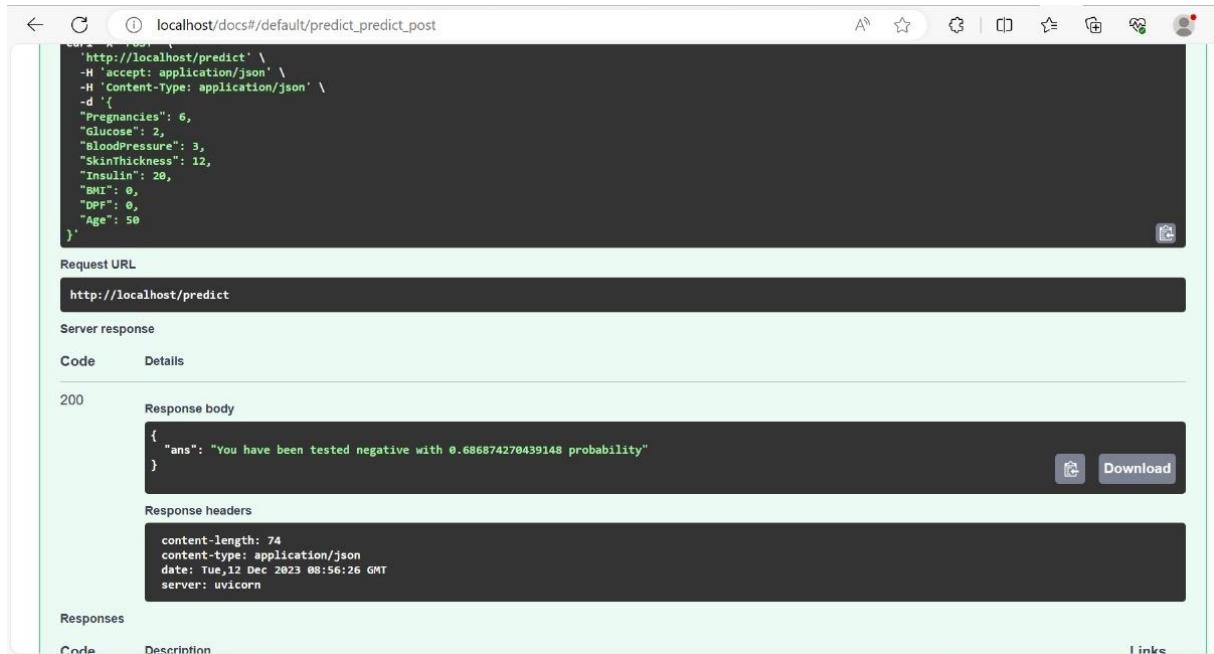


Figure VI-4: Conteneurisation de l'API

Le résultat final après l'accès au localhost :



The screenshot shows a browser window with the URL `localhost/docs#/default/predict_predict_post`. The request body is a JSON object:

```
'http://localhost/predict' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "Pregnancies": 6,
    "Glucose": 2,
    "BloodPressure": 3,
    "SkinThickness": 12,
    "Insulin": 20,
    "BMI": 0,
    "DPF": 0,
    "Age": 50
}'
```

The server response is 200 OK. The response body is:

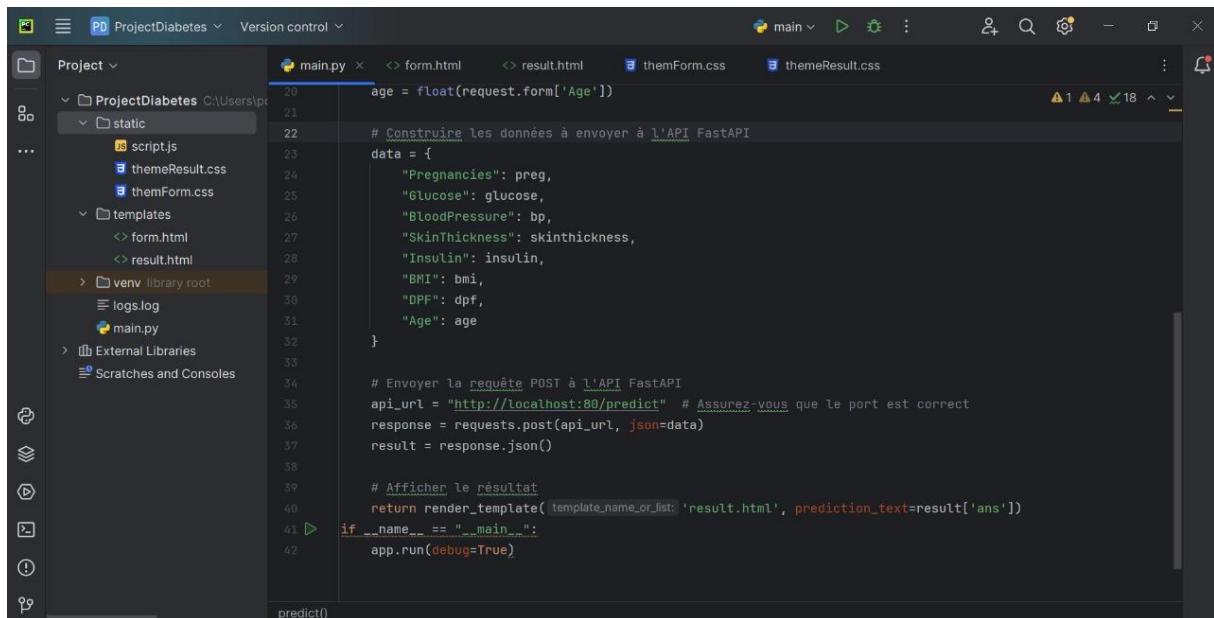
```
{
    "ans": "You have been tested negative with 0.686874270439148 probability"
}
```

The response headers are:

```
content-length: 74
content-type: application/json
date: Tue, 12 Dec 2023 08:56:26 GMT
server: uvicorn
```

Figure VI-5: Api conteneurisée

Testons notre api conteneurisée, pour faire cela, on va l'intégrer dans Flask comme ceci :



```
age = float(request.form['Age'])

# Construire les données à envoyer à l'API FastAPI
data = {
    "Pregnancies": preg,
    "Glucose": glucose,
    "BloodPressure": bp,
    "SkinThickness": skintickness,
    "Insulin": insulin,
    "BMI": bmi,
    "DPF": dpf,
    "Age": age
}

# Envoyer la requête POST à l'API FastAPI
api_url = "http://localhost:80/predict" # Assurez-vous que le port est correct
response = requests.post(api_url, json=data)
result = response.json()

# Afficher le résultat
return render_template('result.html', prediction_text=result['ans'])

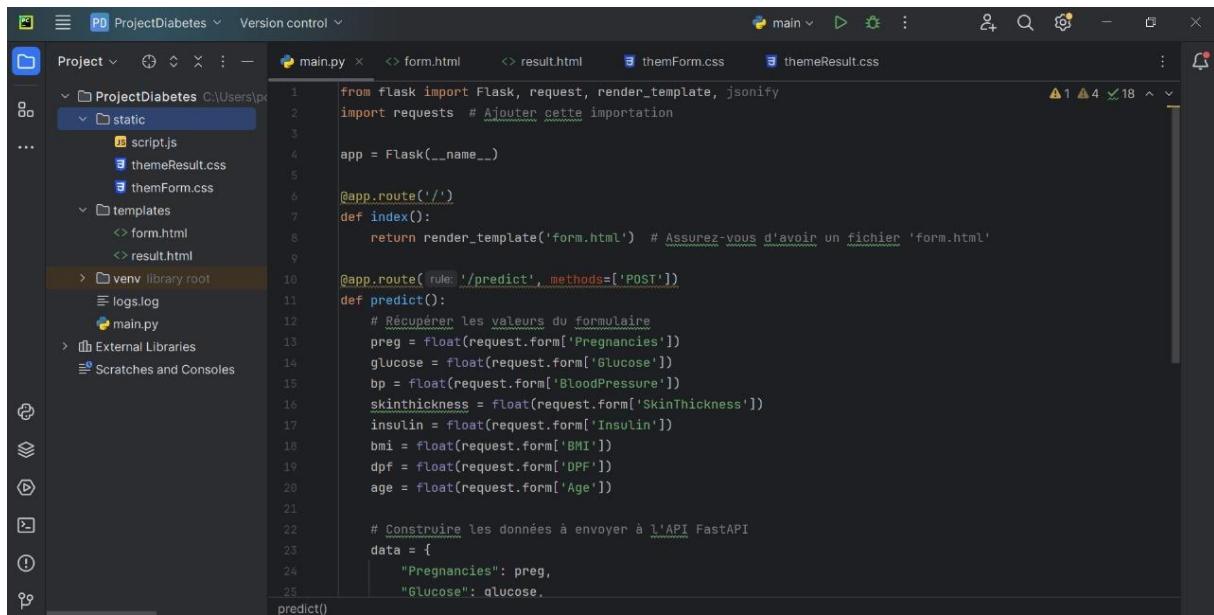
if __name__ == "__main__":
    app.run(debug=True)
```

Figure VI-6: Lien vers l'API

Vous pouvez remarquer que le lien change, on a mis le lien vers l'api conteneurisée qui est :

<http://localhost:80/predict>

Le reste du code reste presque le même, vous allez trouver cela dans le lien Github du projet :



The screenshot shows the PyCharm IDE interface. The left sidebar displays the project structure for 'ProjectDiabetes' located at 'C:\Users\pxx'. The 'main.py' file is open in the editor, showing Python code for a Flask application. The code imports Flask and requests, defines routes for index and predict, and constructs data to send to a FastAPI endpoint. The right side of the screen shows the code editor with syntax highlighting and various status indicators.

```
from flask import Flask, request, render_template, jsonify
import requests # Ajouter cette importation

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('form.html') # Assurez-vous d'avoir un fichier 'form.html'

@app.route('/predict', methods=['POST'])
def predict():
    # Récupérer les valeurs du formulaire
    preg = float(request.form['Pregnancies'])
    glucose = float(request.form['Glucose'])
    bp = float(request.form['BloodPressure'])
    skinthickness = float(request.form['SkinThickness'])
    insulin = float(request.form['Insulin'])
    bmi = float(request.form['BMI'])
    dpf = float(request.form['DPF'])
    age = float(request.form['Age'])

    # Construire les données à envoyer à l'API FastAPI
    data = {
        "Pregnancies": preg,
        "Glucose": glucose,
```

Maintenant lançons notre application Flask :

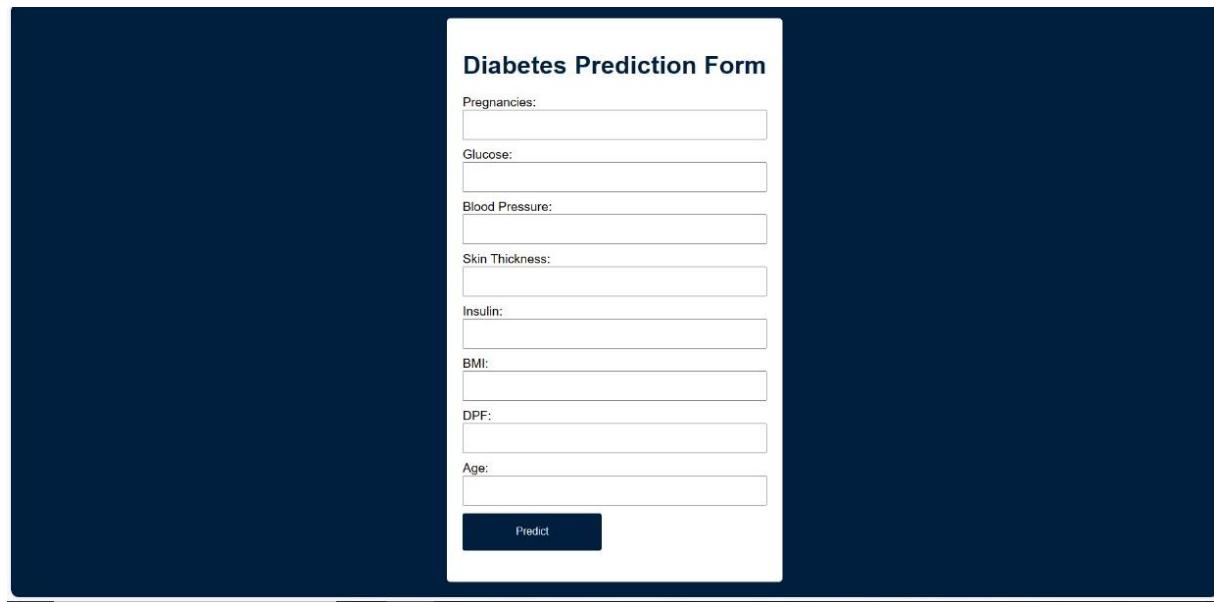


Figure VI-7: Application Flask API

On va remplir par des valeurs pour tester le fonctionnement, le comportement doit être le même que celui avec l'API intégré localement.

Diabetes Prediction Form

Pregnancies:

Glucose:

Blood Pressure:

Skin Thickness:

Insulin:

BMI:

DPF:

Age:

On clique sur submit :

Prediction Result:

You have been tested positive with 0.798926591873169 probability

[Retour à l'accueil](#)

Annexe

[1] <https://github.com/ghm-group/continuous-training-pipeline-airflow/tree/main>

[2] <https://github.com/pycaret/pycaret>

[3] <https://fastapi.tiangolo.com/deployment/docker/>

