from fastapi import FastAPI, File, UploadFile, HTTPException

import firebase\_admin

from firebase\_admin import credentials, storage

import os

import requests

from scipy.signal import find\_peaks, savgol\_filter

from skimage.io import imread

from skimage.color import rgb2gray

from skimage.filters import threshold\_otsu

import joblib

import tempfile

import cv2

import numpy as np

app = FastAPI()

# Initialize Firebase

cred = credentials.Certificate('./missioncapstone-21b12-firebase-adminsdk-9p748-0a02dc3abd.json') # Adjust path

firebase\_admin.initialize\_app(cred, {'storageBucket': 'missioncapstone-21b12.appspot.com'})

# Function to fetch the most recent image from Firebase Storage

async def fetch\_most\_recent\_image():

bucket = storage.bucket()

blobs = bucket.list\_blobs(prefix='images/')

images = [{'name': blob.name, 'updated': blob.updated} for blob in blobs if blob.name.endswith('.jpg')]

if not images:

raise HTTPException(status\_code=404, detail="No images found in Firebase storage.")

most\_recent\_image = sorted(images, key=lambda x: x['updated'], reverse=True)[0]['name']

temp\_image\_path = tempfile.mktemp(suffix='.jpg')

blob = bucket.blob(most\_recent\_image)

blob.download\_to\_filename(temp\_image\_path)

return temp\_image\_path

# Preprocess the ECG image

def process\_image\_for\_ecg(image\_path):

# Read the image using OpenCV

image = cv2.imread(image\_path)

if image.shape[2] == 4: # Convert RGBA to RGB

image = cv2.cvtColor(image, cv2.COLOR\_RGBA2RGB)

gray\_image = rgb2gray(image)

threshold = threshold\_otsu(gray\_image)

binary\_image = gray\_image > threshold

smoothed\_signal = savgol\_filter(binary\_image.flatten(), 41, 2)

return smoothed\_signal

# Extract ECG metrics

def extract\_ecg\_metrics(smoothed\_signal):

r\_peaks, \_ = find\_peaks(smoothed\_signal, height=0.5, distance=50)

rr\_intervals = np.diff(r\_peaks)

avg\_rr\_interval = np.mean(rr\_intervals) if len(rr\_intervals) > 0 else 0

rr\_interval\_std = np.std(rr\_intervals) if len(rr\_intervals) > 0 else 0

heart\_rate = 60 / avg\_rr\_interval if avg\_rr\_interval else 0

qrs\_durations = [min(smoothed\_signal[:r]) - min(smoothed\_signal[r:]) for r in r\_peaks]

avg\_qrs\_duration = np.mean(qrs\_durations) if len(qrs\_durations) > 0 else 0

return {

'avgRRInterval': avg\_rr\_interval,

'rrIntervalStd': rr\_interval\_std,

'heartRate': heart\_rate,

'avgQRSDuration': avg\_qrs\_duration,

}

# Predict ECG label

async def predict\_ecg\_label(features):

label\_mapping = {0: 'Abnormal', 1: 'HMI', 2: 'MI', 3: 'Normal'}

model\_url = "https://firebasestorage.googleapis.com/v0/b/missioncapstone-21b12.appspot.com/o/76\_voting\_classifier\_model.pkl?alt=media&token=5d72ddc5-0b3f-45fa-85e1-81632ee0e882"

response = requests.get(model\_url)

model = joblib.load(response.content)

predicted\_class = model.predict([features])[0]

return label\_mapping.get(predicted\_class, 'Unknown')

# Get detailed ECG results based on classification

def get\_ecg\_result\_details(predicted\_label):

result\_mapping = {

'Normal': {

'ECGResult': 'Normal ECG Image',

'Notes': 'A normal ECG shows a regular rhythm with normal P waves, QRS complexes, and T waves.'

},

'Abnormal': {

'ECGResult': 'Abnormal ECG Image',

'Notes': 'An abnormal ECG shows irregular rhythms, absent P waves, or prolonged QT intervals.'

},

'MI': {

'ECGResult': 'Myocardial Infarction (MI) ECG Image',

'Notes': 'An ECG showing ST elevation, T wave inversion, and pathological Q waves.'

},

'History of MI': {

'ECGResult': 'History of Myocardial Infarction (MI) ECG Image',

'Notes': 'An ECG with persistent Q waves and T wave inversion, indicating past heart damage.'

}

}

return result\_mapping.get(predicted\_label, {

'ECGResult': 'Unknown',

'Notes': 'The classification result could not be determined.'

})

@app.post("/predict")

async def predict\_ecg():

try:

image\_path = await fetch\_most\_recent\_image()

smoothed\_signal = process\_image\_for\_ecg(image\_path)

metrics = extract\_ecg\_metrics(smoothed\_signal)

features = [metrics['avgRRInterval'], metrics['rrIntervalStd'], metrics['heartRate'], metrics['avgQRSDuration']]

predicted\_label = await predict\_ecg\_label(features)

result\_details = get\_ecg\_result\_details(predicted\_label)

return {

"predictedLabel": predicted\_label,

"ECGResults": result\_details['ECGResult'],

"Notes": result\_details['Notes']

}

except Exception as e:

raise HTTPException(status\_code=500, detail=str(e))

from fastapi import FastAPI, HTTPException

from pydantic import BaseModel

import aiohttp

import json

app = FastAPI()

# Define possible ECG classifications

ecg\_classes = ['Myocardial Infarction', 'Normal Heartbeat', 'History of Myocardial Infarction', 'Abnormal Heartbeat']

# Pydantic models to handle incoming requests

class HeartDiseaseData(BaseModel):

Smoking: str

AlcoholDrinking: str

Stroke: str

DiffWalking: str

Sex: str

Diabetic: str

PhysicalActivity: str

Asthma: str

KidneyDisease: str

SkinCancer: str

AgeCategory: str

Race: str

GenHealth: str

BMI: float

PhysicalHealth: float

MentalHealth: float

SleepTime: float

class ECGRequest(BaseModel):

ecgClass: str

heartDiseaseData: HeartDiseaseData

ecgImage: str

# Combined Diagnosis, Recommendation, and Notes Function

def assign\_diagnosis\_recommendation\_and\_notes(ecg\_class, heart\_disease):

diagnosis, notes, recommendation, follow\_up, critical\_alert, referral = "", "", "", "", "", ""

# Your existing logic here for assigning diagnosis based on ECG class and heart disease

if ecg\_class == 'Myocardial Infarction':

if heart\_disease == 'Positive':

diagnosis = 'Acute Coronary Syndrome with High Risk of Recurrence'

notes = 'This is a critical scenario with high urgency, requiring immediate intervention.'

recommendation = 'Immediate referral to a cardiologist; prescribe anticoagulants and lifestyle changes.'

follow\_up = 'Weekly follow-up until condition stabilizes.'

critical\_alert = 'High'

referral = 'Yes'

else:

diagnosis = 'Previous Myocardial Infarction; Stable Condition'

notes = 'The patient’s condition is stable post-MI, but monitoring is required for long-term health.'

recommendation = 'Regular cardiovascular assessment every 3 months; monitor symptoms.'

follow\_up = '3-month follow-up with primary care.'

critical\_alert = 'Medium'

referral = 'Consider cardiology if symptoms worsen'

elif ecg\_class == 'History of Myocardial Infarction':

if heart\_disease == 'Positive':

diagnosis = 'History of Myocardial Infarction; Risk of Complications'

notes = 'This patient has a history of MI, and further evaluation is necessary to prevent complications.'

recommendation = 'Lifestyle counseling and medication adherence; low-dose aspirin advised.'

follow\_up = 'Monthly check-ins to assess risk.'

critical\_alert = 'Medium'

referral = 'Yes'

else:

diagnosis = 'Stable Post-MI; No Current Risk'

notes = 'The patient has recovered well from a previous MI, with no current issues.'

recommendation = 'Annual cardiovascular exam; maintain healthy lifestyle.'

follow\_up = 'Annual check-up.'

critical\_alert = 'Low'

referral = 'No'

elif ecg\_class == 'Abnormal Heartbeat':

if heart\_disease == 'Positive':

diagnosis = 'Chronic Arrhythmia'

notes = 'Chronic arrhythmia requires ongoing management and may involve medication or procedures.'

recommendation = 'Anti-arrhythmic drugs and possible ablation therapy; avoid strenuous activities.'

follow\_up = 'Bi-weekly follow-up with ECG monitoring.'

critical\_alert = 'High'

referral = 'Yes'

else:

diagnosis = 'Minor Arrhythmia; No Immediate Concern'

notes = 'Minor arrhythmias are typically benign but should be monitored if symptoms develop.'

recommendation = 'Lifestyle modifications; reduce caffeine and stress.'

follow\_up = '3-month follow-up if symptoms develop.'

critical\_alert = 'Low'

referral = 'No'

else: # Normal Heartbeat

if heart\_disease == 'Negative':

diagnosis = 'Healthy Cardiac Status'

notes = 'The patient has a normal heart rhythm, indicating healthy cardiac function.'

recommendation = 'Maintain a balanced diet, regular exercise, and healthy sleep habits.'

follow\_up = 'Annual wellness check-up.'

critical\_alert = 'None'

referral = 'No'

else:

diagnosis = 'Early Signs of Cardiac Risk'

notes = 'Early signs of potential heart disease, requiring preventive measures and monitoring.'

recommendation = 'Begin preventive medication and cardiovascular fitness plan.'

follow\_up = '6-month follow-up for early intervention.'

critical\_alert = 'Medium'

referral = 'Consider preventive cardiology'

return {

'Diagnosis': diagnosis,

'Notes': notes,

'Recommendation': recommendation,

'FollowUp': follow\_up,

'CriticalAlert': critical\_alert,

'Referral': referral

}

# API Route to fetch diagnosis from both the Heart Disease model and the ECG model

@app.post("/get-diagnosis")

async def get\_diagnosis(ecg\_request: ECGRequest):

ecg\_class = ecg\_request.ecgClass

heart\_disease\_data = ecg\_request.heartDiseaseData

ecg\_image = ecg\_request.ecgImage

if not ecg\_class or not heart\_disease\_data:

raise HTTPException(status\_code=400, detail="ECG Class and Heart Disease data are required.")

try:

# Step 1: Call the ECG classification API (assuming it's hosted at http://ecg-api.com)

async with aiohttp.ClientSession() as session:

async with session.post('http://ecg-api.com/classify', json={'image': ecg\_image}) as ecg\_response:

ecg\_data = await ecg\_response.json()

ecg\_prediction = ecg\_data['prediction']

# Step 2: Call the Heart Disease prediction API (assuming it's hosted at http://heartdisease-api.com)

async with session.post('http://heartdisease-api.com/predict', json=heart\_disease\_data.dict()) as heart\_disease\_response:

heart\_disease\_data = await heart\_disease\_response.json()

heart\_disease\_prediction = heart\_disease\_data['prediction']

# Step 3: Combine the predictions from both models

result = assign\_diagnosis\_recommendation\_and\_notes(ecg\_prediction, heart\_disease\_prediction)

# Step 4: Send the combined result as response

return {

'Heart Disease Prediction': 'Positive' if heart\_disease\_prediction == 'Positive' else 'Negative',

'ECG Class': ecg\_prediction,

'Diagnosis': result['Diagnosis'],

'Notes': result['Notes'],

'Recommendation': result['Recommendation'],

'FollowUp': result['FollowUp'],

'CriticalAlert': result['CriticalAlert'],

'Referral': result['Referral']

}

except Exception as error:

raise HTTPException(status\_code=500, detail=f"An error occurred while fetching predictions: {error}")

# Run the server using: uvicorn filename:app --reload

from fastapi import FastAPI, HTTPException

from pydantic import BaseModel

from firebase\_admin import credentials, firestore, storage, initialize\_app

import tensorflow as tf

import pickle

import json

import os

# Initialize Firebase Admin SDK

cred = credentials.Certificate("path-to-firebase-service-account.json") # Update this path

initialize\_app(cred, {"storageBucket": "your-bucket-name.appspot.com"}) # Replace with your Firebase Storage bucket name

# Initialize FastAPI

app = FastAPI()

# Define the patient data model

class PatientData(BaseModel):

Smoking: str

AlcoholDrinking: str

Stroke: str

DiffWalking: str

Sex: str

Diabetic: str

PhysicalActivity: str

Asthma: str

KidneyDisease: str

SkinCancer: str

AgeCategory: str

Race: str

GenHealth: str

BMI: float

PhysicalHealth: float

MentalHealth: float

SleepTime: float

# Download file from Firebase Storage

def download\_file(firebase\_path, local\_path):

bucket = storage.bucket()

blob = bucket.blob(firebase\_path)

blob.download\_to\_filename(local\_path)

print(f"Downloaded {firebase\_path} to {local\_path}")

# Store patient data in Firebase Firestore

def store\_patient\_data(patient\_data):

db = firestore.client()

patient\_ref = db.collection("patients").add(patient\_data.dict())

print("Patient data stored successfully in Firestore.")

# Load encoders and scaler from Firebase Storage

def load\_encoders\_and\_scaler():

encoders\_path = './label\_encoders'

scaler\_path = './scaler'

# Download files

download\_file('Label\_encoder/continuous\_scaler.pkl', f'{scaler\_path}/continuous\_scaler.pkl')

with open(f'{scaler\_path}/continuous\_scaler.pkl', 'rb') as file:

scaler = pickle.load(file)

label\_encoders = {}

categorical\_features = [

'Smoking', 'AlcoholDrinking', 'Stroke', 'DiffWalking', 'Sex',

'Diabetic', 'PhysicalActivity', 'Asthma', 'KidneyDisease', 'SkinCancer',

'AgeCategory', 'Race', 'GenHealth',

]

for feature in categorical\_features:

download\_file(f'Label\_encoder/{feature}\_encoder.pkl', f'{encoders\_path}/{feature}\_encoder.pkl')

with open(f'{encoders\_path}/{feature}\_encoder.pkl', 'rb') as file:

label\_encoders[feature] = pickle.load(file)

return label\_encoders, scaler

# Load the trained model

def load\_model(model\_path):

download\_file('https://firebasestorage.googleapis.com/v0/b/missioncapstone-21b12.appspot.com/o/Models%2FHeartDiseaseModel\_best.keras?alt=media&token=7139da60-4042-4eb8-9bea-fd5ef29891b5', model\_path)

model = tf.keras.models.load\_model(model\_path)

return model

# Prediction logic

def predict\_heart\_disease(model, label\_encoders, scaler, input\_data):

categorical\_features = [

'Smoking', 'AlcoholDrinking', 'Stroke', 'DiffWalking', 'Sex',

'Diabetic', 'PhysicalActivity', 'Asthma', 'KidneyDisease', 'SkinCancer',

'AgeCategory', 'Race', 'GenHealth',

]

continuous\_features = ['BMI', 'PhysicalHealth', 'MentalHealth', 'SleepTime']

# Encode categorical features

for feature in categorical\_features:

if input\_data.get(feature):

input\_data[feature] = label\_encoders[feature].get(input\_data[feature], -1) # Adjust for encoding

# Scale continuous features

for feature in continuous\_features:

if input\_data.get(feature):

input\_data[feature] = (input\_data[feature] - scaler['mean'][feature]) / scaler['std'][feature]

# Prepare input data for prediction

input\_tensor = tf.convert\_to\_tensor([list(input\_data.values())], dtype=tf.float32)

prediction = model.predict(input\_tensor).flatten()[0]

return prediction

# API Routes

# Sync patient data and store it in Firestore

@app.post("/store-patient-data")

async def store\_patient\_data\_route(patient\_data: PatientData):

try:

store\_patient\_data(patient\_data)

return {"message": "Patient data synced successfully!"}

except Exception as e:

print(f"Error storing patient data: {e}")

raise HTTPException(status\_code=500, detail="Error syncing patient data")

# Prediction route

@app.post("/predict-heart-disease")

async def predict\_heart\_disease\_route(patient\_data: PatientData):

try:

label\_encoders, scaler = load\_encoders\_and\_scaler()

model = load\_model("./Models/HeartDiseaseModel\_best.keras")

# Predict heart disease

input\_data = patient\_data.dict()

prediction = predict\_heart\_disease(model, label\_encoders, scaler, input\_data)

prediction\_label = "Positive" if prediction > 0.5 else "Negative"

result\_message = "This indicates that this patient is likely to have heart disease." if prediction > 0.5 else "This indicates that this patient is unlikely to have heart disease."

return {

"prediction": f"Heart Disease Prediction: {prediction\_label} (Score: {prediction:.2f})",

"resultMessage": result\_message,

}

except Exception as e:

print(f"Error during prediction: {e}")

raise HTTPException(status\_code=500, detail="Error making prediction")

# Start the server (FastAPI auto-starts on the specified port)

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000)