"""

Combined Prototype File for Multimodal Deep Learning for Cardiovascular Prediction

This file includes functions to load and process data, build models, evaluate them, and

an API to serve predictions. Each section is commented with explanations in simple terms.

Usage:

To run training/evaluation: python combined\_prototype.py train

To run the API server: python combined\_prototype.py api

"""

#######################################

# IMPORTS #

#######################################

import os # For operating system interactions

import sys # For reading command-line arguments

import requests # To talk to other web services

import pandas as pd # For working with tables (data frames)

import numpy as np # For math and numbers

import missingno as msno # To visualize missing data

from typing import Optional # To say that something may be optional

from sklearn.impute import SimpleImputer # To fill in missing numbers

from pyspark.sql import SparkSession # To use Spark (for big data)

from pyspark.sql.types import StructType # To define the structure of data

from pyspark.sql import DataFrame # Spark DataFrame type

from tsfresh import extract\_features # To get extra details from time series data

# TensorFlow and Keras for building deep learning models

import tensorflow as tf

from tensorflow import keras

Model = keras.models.Model

Sequential = keras.models.Sequential

load\_model = keras.models.load\_model

layers = tf.keras.layers

Input = layers.Input

Dense = layers.Dense

Dropout = layers.Dropout

BatchNormalization = layers.BatchNormalization

Conv2D = layers.Conv2D

LSTM = layers.LSTM

Flatten = layers.Flatten

Concatenate = layers.Concatenate

Adam = tf.keras.optimizers.Adam

EarlyStopping = tf.keras.callbacks.EarlyStopping

to\_categorical = tf.keras.utils.to\_categorical

import joblib # For saving/loading objects (like our scaler)

import matplotlib.pyplot as plt # For plotting graphs

import shap # For explaining model predictions

# Scikit-learn tools for evaluation and data splitting

from sklearn.metrics import (accuracy\_score, precision\_score, recall\_score, f1\_score,

roc\_auc\_score, mean\_absolute\_error, mean\_squared\_error, r2\_score)

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

# Pydantic for validating incoming API data

from pydantic import BaseModel, ValidationError

# Flask for the API

from flask import Flask, request, jsonify

#######################################

# DATA PROCESSING FUNCTIONS #

#######################################

def get\_spark\_session(app\_name: str = "MultimodalDataPipeline") -> SparkSession:

"""

Start a Spark session (like turning on a big helper computer).

"""

return SparkSession.builder.appName(app\_name).getOrCreate()

def load\_data\_spark(file\_path: str, schema: Optional[StructType] = None) -> DataFrame:

"""

Load a CSV file using Spark and cache it (store it for fast access).

Args:

file\_path (str): The path to the CSV file.

schema (Optional[StructType]): The structure of the data (if provided).

Returns:

DataFrame: A Spark DataFrame with the data.

"""

spark = get\_spark\_session()

# Read the CSV file with header, guess the schema, and cache the result.

return spark.read.csv(r'/Users/manny/Desktop/Desktop Folder/MultimodalDataset021525/ehr\_data.csv', header=True, schema=schema, inferSchema=True).cache()

def load\_data\_spark(file\_path: str, schema: Optional[StructType] = None) -> DataFrame:

"""

Load a CSV file using Spark and cache it (store it for fast access).

Args:

file\_path (str): The path to the CSV file.

schema (Optional[StructType]): The structure of the data (if provided).

Returns:

DataFrame: A Spark DataFrame with the data.

"""

spark = get\_spark\_session()

# Read the CSV file with header, guess the schema, and cache the result.

return spark.read.csv(r'/Users/manny/Desktop/Desktop Folder/MultimodalDataset021525/imaging\_data.csv', header=True, schema=schema, inferSchema=True).cache()

def load\_data\_spark(file\_path: str, schema: Optional[StructType] = None) -> DataFrame:

"""

Load a CSV file using Spark and cache it (store it for fast access).

Args:

file\_path (str): The path to the CSV file.

schema (Optional[StructType]): The structure of the data (if provided).

Returns:

DataFrame: A Spark DataFrame with the data.

"""

spark = get\_spark\_session()

# Read the CSV file with header, guess the schema, and cache the result.

return spark.read.csv(r'/Users/manny/Desktop/Desktop Folder/MultimodalDataset021525/wearables\_data.csv', header=True, schema=schema, inferSchema=True).cache()

def load\_data\_spark(file\_path: str, schema: Optional[StructType] = None) -> DataFrame:

"""

Load a CSV file using Spark and cache it (store it for fast access).

Args:

file\_path (str): The path to the CSV file.

schema (Optional[StructType]): The structure of the data (if provided).

Returns:

DataFrame: A Spark DataFrame with the data.

"""

spark = get\_spark\_session()

# Read the CSV file with header, guess the schema, and cache the result.

return spark.read.csv(r'/Users/manny/Desktop/Desktop Folder/MultimodalDataset021525/genomic\_data.csv', header=True, schema=schema, inferSchema=True).cache()

def impute\_and\_visualize(df: pd.DataFrame) -> pd.DataFrame:

"""

Show missing data with a plot and fill in missing numbers with the average.

Args:

df (pd.DataFrame): The table of data.

Returns:

pd.DataFrame: The same table with missing numbers filled.

"""

msno.matrix(df) # Show a picture of missing data

imputer = SimpleImputer(strategy='mean') # Create a tool to fill missing numbers with the average

return pd.DataFrame(imputer.fit\_transform(df), columns=df.columns)

def merge\_data(ehr\_data: pd.DataFrame, imaging\_data: pd.DataFrame,

wearables\_data: pd.DataFrame, genomic\_data: pd.DataFrame) -> pd.DataFrame:

"""

Combine different tables into one big table using the 'patient\_id' column.

Args:

ehr\_data, imaging\_data, wearables\_data, genomic\_data (pd.DataFrame): The data tables.

Returns:

pd.DataFrame: The merged table.

"""

merged = ehr\_data.merge(imaging\_data, on='patient\_id')\

.merge(wearables\_data, on='patient\_id')\

.merge(genomic\_data, on='patient\_id')

return merged

def extract\_time\_series\_features(wearables\_data: pd.DataFrame) -> pd.DataFrame:

"""

Extract extra details from time-series data (like steps over time) using tsfresh.

Args:

wearables\_data (pd.DataFrame): Wearable device data with 'patient\_id' and 'time'.

Returns:

pd.DataFrame: A table with extra features.

"""

return extract\_features(wearables\_data, column\_id="patient\_id", column\_sort="time")

#######################################

# MODEL BUILDING FUNCTIONS #

#######################################

def build\_deep\_learning\_model(input\_dim: int) -> tf.keras.Model:

"""

Build a model that predicts both a classification (yes/no disease) and regression (time).

Args:

input\_dim (int): The number of input features.

Returns:

Model: A compiled Keras model.

"""

# Create the input layer expecting input\_dim numbers.

inputs = Input(shape=(input\_dim,))

# Add a dense (fully connected) layer with 128 neurons.

x = Dense(128, activation='relu')(inputs)

# Dropout randomly turns off 30% of neurons to prevent overfitting.

x = Dropout(0.3)(x)

# Batch normalization keeps numbers balanced.

x = BatchNormalization()(x)

# Classification head: predicts yes/no using two outputs.

classification = Dense(2, activation='softmax', name='classification')(x)

# Regression head: predicts a number (like time to event).

regression = Dense(1, activation='linear', name='regression')(x)

# Create the model with the two outputs.

model = Model(inputs=inputs, outputs=[classification, regression])

# Compile the model with Adam optimizer and proper loss functions.

model.compile(optimizer=Adam(),

loss={'classification': 'categorical\_crossentropy', 'regression': 'mse'},

metrics={'classification': 'accuracy', 'regression': 'mae'})

return model

def build\_multimodal\_model(input\_dim: int) -> tf.keras.Model:

"""

Build a multimodal model that handles numerical, time-series, image, and genomic data.

Args:

input\_dim (int): The number of features for numerical data.

Returns:

Model: A compiled Keras model.

"""

# Numerical branch (for regular numbers)

input\_num = Input(shape=(input\_dim,))

x = Dense(128, activation='relu')(input\_num)

x = Dropout(0.3)(x)

x = BatchNormalization()(x)

# Time-series branch (for data over time)

input\_ts = Input(shape=(None, input\_dim))

ts = LSTM(64, return\_sequences=False)(input\_ts)

# Image branch (for picture data)

input\_img = Input(shape=(64, 64, 3))

img = Conv2D(32, (3, 3), activation='relu')(input\_img)

img = Flatten()(img)

# Genomic branch (for gene data, here treated like numbers)

input\_genomic = Input(shape=(input\_dim,))

genomic = Dense(64, activation='relu')(input\_genomic)

# Merge (or "fuse") all branches together.

merged = Concatenate()([x, ts, img, genomic])

merged = Dense(128, activation='relu')(merged)

merged = Dropout(0.4)(merged)

# Two prediction heads: one for classification and one for regression.

classification = Dense(2, activation='softmax', name='classification')(merged)

regression = Dense(1, activation='linear', name='regression')(merged)

model = Model(inputs=[input\_num, input\_ts, input\_img, input\_genomic],

outputs=[classification, regression])

model.compile(optimizer=Adam(),

loss={'classification': 'categorical\_crossentropy', 'regression': 'mse'},

metrics={'classification': 'accuracy', 'regression': 'mae'})

return model

def build\_baseline\_model(input\_dim: int) -> tf.keras.Model:

"""

Build a simple model for binary classification to serve as a baseline.

Args:

input\_dim (int): The number of input features.

Returns:

Model: A compiled Keras Sequential model.

"""

model = Sequential([

Dense(128, activation='relu', input\_shape=(input\_dim,)),

Dropout(0.3),

Dense(64, activation='relu'),

Dense(1, activation='sigmoid') # Sigmoid outputs a value between 0 and 1.

])

model.compile(optimizer=Adam(), loss='binary\_crossentropy', metrics=['accuracy'])

return model

def get\_model\_and\_scaler(model\_path: str = "multimodal\_model.h5", scaler\_path: str = "scaler.pkl"):

"""

Load a saved model and scaler from disk so we can use them later.

Args:

model\_path (str): Where the saved model is.

scaler\_path (str): Where the saved scaler is.

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

tuple: The model and scaler objects.

"""

model = load\_model(model\_path)

scaler = joblib.load(scaler\_path)

return model, scaler

#######################################

# EVALUATION FUNCTIONS #

#######################################

def evaluate\_model(model, X\_test, y\_test):

"""

Evaluate a model by comparing its predictions to the true answers.

Args:

model: The Keras model to evaluate.

X\_test (np.ndarray): The test data.

y\_test (np.ndarray): The true labels.

Returns:

dict: A set of evaluation scores.

"""

# Predict using the model and turn probabilities into 0 or 1.

y\_pred = (model.predict(X\_test) > 0.5).astype(int)

return {

"accuracy": accuracy\_score(y\_test, y\_pred),

"precision": precision\_score(y\_test, y\_pred),

"recall": recall\_score(y\_test, y\_pred),

"f1": f1\_score(y\_test, y\_pred),

"roc\_auc": roc\_auc\_score(y\_test, y\_pred)

}

def ablation\_study(multimodal\_model, data\_dict, target\_class, split\_func):

"""

Test the importance of each data type by removing one at a time and evaluating the model.

Args:

multimodal\_model: The model to test.

data\_dict (dict): A dictionary with keys like 'ehr', 'imaging', etc.

target\_class (pd.Series): The true labels.

split\_func: A function to split data into training and test sets.

Returns:

dict: The evaluation scores for each ablation.

"""

ablation\_results = {}

# Combine all modalities first.

combined\_data = pd.concat([data\_dict['ehr'], data\_dict['imaging'],

data\_dict['wearables'], data\_dict['genomic']], axis=1)

X\_train, X\_test, y\_train, y\_test = split\_func(combined\_data, target\_class)

multimodal\_model.fit(X\_train, y\_train, epochs=10, batch\_size=32, verbose=0)

ablation\_results['All'] = evaluate\_model(multimodal\_model, X\_test, y\_test)

# Now remove each modality one by one.

for modality in ['ehr', 'imaging', 'wearables', 'genomic']:

remaining = [data for key, data in data\_dict.items() if key != modality]

ablation\_data = pd.concat(remaining, axis=1)

X\_train, X\_test, y\_train, y\_test = split\_func(ablation\_data, target\_class)

multimodal\_model.fit(X\_train, y\_train, epochs=10, batch\_size=32, verbose=0)

ablation\_results[f'No {modality.capitalize()}'] = evaluate\_model(multimodal\_model, X\_test, y\_test)

return ablation\_results

def split\_data(features, target, test\_size=0.2, random\_state=42):

"""

Split the data into training and testing parts.

Args:

features (np.ndarray or pd.DataFrame): The feature data.

target (np.ndarray or pd.Series): The target labels.

Returns:

tuple: The split data (X\_train, X\_test, y\_train, y\_test).

"""

return train\_test\_split(features, target, test\_size=test\_size, random\_state=random\_state)

#######################################

# API SECTION #

#######################################

# Define a Pydantic model to validate incoming API data.

class PatientInput(BaseModel):

patient\_id: str

ehr\_features: dict

imaging\_features: list

wearables\_data: dict

genomic\_features: list

# Initialize Flask app for our API.

app = Flask(\_\_name\_\_)

def authorize\_request(token: str) -> bool:

"""

Check if the request has a valid token.

Args:

token (str): The token provided by the requester.

Returns:

bool: True if authorized, False otherwise.

"""

# In a real application, talk to an authorization server.

auth\_server = os.environ.get('AUTH\_SERVER', 'https://hospital-auth-server.com/validate')

response = requests.post(auth\_server, json={'token': token})

return response.status\_code == 200

# Middleware: Check authorization before every request.

@app.before\_request

def middleware():

token = request.headers.get('Authorization')

if not token or not authorize\_request(token):

return jsonify({'error': 'Unauthorized'}), 401

@app.route('/predict', methods=['POST'])

def predict\_cvd():

"""

API endpoint that takes in patient data and returns a heart disease prediction.

"""

try:

# Validate incoming data using our checklist (PatientInput).

validated\_data = PatientInput.parse\_obj(request.json)

except ValidationError as e:

return jsonify({'error': e.errors()}), 400

data = validated\_data.dict()

try:

# Convert incoming data into proper formats (tables and arrays).

ehr\_features = pd.DataFrame([data['ehr\_features']])

imaging\_features = np.array(data['imaging\_features']).reshape(1, -1)

wearables\_df = pd.DataFrame([data['wearables\_data']])

genomic\_features = np.array(data['genomic\_features']).reshape(1, -1)

# Extract extra features from the wearables (time series).

time\_series\_features = extract\_time\_series\_features(wearables\_df)

# Combine all features together.

combined\_features = pd.concat([ehr\_features, time\_series\_features], axis=1)

combined\_features = pd.concat([

combined\_features,

pd.DataFrame(imaging\_features),

pd.DataFrame(genomic\_features)

], axis=1)

# Scale the combined features using our pre-saved scaler.

combined\_features\_scaled = scaler.transform(combined\_features)

# Make predictions with the model. (Here we use placeholder arrays for branches like images.)

predictions = model.predict([

combined\_features\_scaled,

combined\_features\_scaled,

np.random.rand(len(combined\_features\_scaled), 64, 64, 3),

combined\_features\_scaled

])

classification = predictions[0]

regression = predictions[1]

# Interpret the predictions.

disease\_onset\_prob = classification[0][1]

disease\_onset\_pred = "Positive" if disease\_onset\_prob > 0.5 else "Negative"

time\_to\_event = regression[0][0]

# Prepare the response.

response = {

"patient\_id": data['patient\_id'],

"disease\_onset": disease\_onset\_pred,

"disease\_onset\_probability": disease\_onset\_prob,

"time\_to\_event": time\_to\_event

}

return jsonify(response)

except Exception as e:

return jsonify({'error': str(e)}), 500

@app.route('/test', methods=['GET'])

def test():

"""

A simple test endpoint to show an example of patient data.

"""

sample\_data = {

"patient\_id": "12345",

"ehr\_features": {

"age": 55,

"sex": 1,

"cp": 3,

"trestbps": 130,

"chol": 250,

"fbs": 0,

"restecg": 0,

"thalach": 150,

"exang": 0,

"oldpeak": 1.5,

"slope": 2,

"ca": 1,

"thal": 3

},

"imaging\_features": [0.1, 0.3, 0.5, 0.7, 0.9],

"wearables\_data": {

"patient\_id": "12345",

"steps\_per\_day": 8000,

"avg\_heart\_rate": 75,

"sleep\_hours": 7,

"activity\_level": "Moderate",

"time": 1 # a placeholder value needed for tsfresh extraction

},

"genomic\_features": [1, 0, 2, 1, 1, 0, 1, 2, 2, 0]

}

return jsonify(sample\_data)

#######################################

# MAIN TRAINING CODE #

#######################################

def run\_training():

"""

Load data, train the multimodal model, evaluate it, and save the model and scaler.

"""

# File names for our data (adjust paths as needed).

EHR\_FILE = r'/Users/manny/Desktop/Desktop Folder/MultimodalDataset021525/ehr\_data.csv'

IMAGING\_FILE = r'/Users/manny/Desktop/Desktop Folder/MultimodalDataset021525/imaging\_data.csv'

WEARABLES\_FILE = r'/Users/manny/Desktop/Desktop Folder/MultimodalDataset021525/wearables\_data.csv'

GENOMIC\_FILE = r'/Users/manny/Desktop/Desktop Folder/MultimodalDataset021525/genomic\_data.csv'

# Load data using Spark and then convert to Pandas.

ehr\_data = load\_data\_spark(EHR\_FILE).toPandas()

imaging\_data = load\_data\_spark(IMAGING\_FILE).toPandas()

wearables\_data = load\_data\_spark(WEARABLES\_FILE).toPandas()

genomic\_data = load\_data\_spark(GENOMIC\_FILE).toPandas()

# Fill in missing numbers and show missing data plots.

ehr\_data = impute\_and\_visualize(ehr\_data)

imaging\_data = impute\_and\_visualize(imaging\_data)

wearables\_data = impute\_and\_visualize(wearables\_data)

genomic\_data = impute\_and\_visualize(genomic\_data)

# Combine the different data tables into one big table.

merged\_data = merge\_data(ehr\_data, imaging\_data, wearables\_data, genomic\_data)

# Extract extra features from the wearables time series and merge them.

time\_series\_features = extract\_time\_series\_features(wearables\_data)

merged\_data = merged\_data.merge(time\_series\_features, on='patient\_id')

# Prepare our features by removing unwanted columns.

features = merged\_data.drop(columns=['patient\_id', 'disease\_onset', 'time\_to\_event'])

scaler\_local = StandardScaler()

features\_scaled = scaler\_local.fit\_transform(features) # Scale the features once.

# Get the labels we want to predict.

target\_class = merged\_data['disease\_onset']

target\_reg = merged\_data['time\_to\_event']

target\_class\_encoded = to\_categorical(target\_class)

# Split the data into training and testing sets.

X\_train, X\_test, y\_train\_class, y\_test\_class, y\_train\_reg, y\_test\_reg = train\_test\_split(

features\_scaled, target\_class\_encoded, target\_reg, test\_size=0.2, random\_state=42

)

# Build the multimodal model using the number of features.

input\_dim = features.shape[1]

multimodal\_model = build\_multimodal\_model(input\_dim)

early\_stopping = EarlyStopping(patience=10, restore\_best\_weights=True)

# Train the model. (For the time-series and image parts, we use placeholder data.)

history = multimodal\_model.fit(

[features\_scaled, features\_scaled, np.random.rand(len(features\_scaled), 64, 64, 3), features\_scaled],

{'classification': target\_class\_encoded, 'regression': target\_reg},

validation\_split=0.2,

epochs=100,

batch\_size=32,

callbacks=[early\_stopping]

)

# Evaluate the model on the test set.

test\_loss, test\_accuracy, test\_mae = multimodal\_model.evaluate(

[X\_test, X\_test, np.random.rand(len(X\_test), 64, 64, 3), X\_test],

{'classification': y\_test\_class, 'regression': y\_test\_reg}

)

print(f"Test Loss: {test\_loss}, Test Accuracy: {test\_accuracy}, Test MAE: {test\_mae}")

# Calculate classification metrics.

y\_class\_pred = np.argmax(multimodal\_model.predict(X\_test)[0], axis=1)

y\_class\_true = np.argmax(y\_test\_class, axis=1)

accuracy = accuracy\_score(y\_class\_true, y\_class\_pred)

precision = precision\_score(y\_class\_true, y\_class\_pred)

recall = recall\_score(y\_class\_true, y\_class\_pred)

f1 = f1\_score(y\_class\_true, y\_class\_pred)

roc\_auc = roc\_auc\_score(y\_test\_class, multimodal\_model.predict(X\_test)[0])

print(f"Classification Metrics:\nAccuracy: {accuracy}\nPrecision: {precision}\nRecall: {recall}\nF1-score: {f1}\nAUC-ROC: {roc\_auc}")

# Calculate regression metrics.

y\_reg\_pred = multimodal\_model.predict(X\_test)[1]

mae = mean\_absolute\_error(y\_test\_reg, y\_reg\_pred)

mse = mean\_squared\_error(y\_test\_reg, y\_reg\_pred)

rmse = np.sqrt(mse)

r2 = r2\_score(y\_test\_reg, y\_reg\_pred)

mape = np.mean(np.abs((y\_test\_reg - y\_reg\_pred) / y\_test\_reg)) \* 100

print(f"Regression Metrics:\nMAE: {mae}\nMSE: {mse}\nRMSE: {rmse}\nR-squared: {r2}\nMAPE: {mape}")

# Build and evaluate a simple baseline model (using EHR data only as an example).

X\_train\_baseline, X\_test\_baseline, y\_train\_baseline, y\_test\_baseline = train\_test\_split(

ehr\_data.drop(columns=['patient\_id']), target\_class, test\_size=0.2, random\_state=42

)

baseline\_model = build\_baseline\_model(X\_train\_baseline.shape[1])

baseline\_model.fit(X\_train\_baseline, y\_train\_baseline, epochs=10, batch\_size=32, verbose=0)

baseline\_results = evaluate\_model(baseline\_model, X\_test\_baseline, y\_test\_baseline)

print("Baseline EHR Model Results:", baseline\_results)

# Perform ablation studies to see which data parts are most important.

data\_dict = {

'ehr': ehr\_data,

'imaging': imaging\_data,

'wearables': wearables\_data,

'genomic': genomic\_data

}

ablation\_results = ablation\_study(multimodal\_model, data\_dict, target\_class, split\_data)

print("Ablation Study Results:", ablation\_results)

# Use SHAP to explain model predictions (make a summary plot).

explainer = shap.KernelExplainer(multimodal\_model.predict, features\_scaled)

shap\_values = explainer.shap\_values(features\_scaled)

shap.summary\_plot(shap\_values, features\_scaled)

# Save the trained model and the scaler for later use.

multimodal\_model.save("multimodal\_model.h5")

joblib.dump(scaler\_local, "scaler.pkl")

print("Model and scaler saved to disk.")

#######################################

# MAIN BLOCK #

#######################################

if \_\_name\_\_ == '\_\_main\_\_':

# Check the command-line arguments to decide what to run.

if len(sys.argv) > 1:

mode = sys.argv[1].lower()

if mode == "train":

print("Running training and evaluation...")

run\_training()

elif mode == "api":

print("Starting the API server on port 5000...")

# When running the API, we need the model and scaler to be available.

# We assume they have been saved already; if not, run training first.

try:

model, scaler = get\_model\_and\_scaler()

except Exception as e:

print("Error loading model/scaler. Please run training first. Error:", e)

sys.exit(1)

app.run(debug=True, port=5000)

else:

print("Usage: python combined\_prototype.py [train|api]")

else:

print("Usage: python combined\_prototype.py [train|api]")