PYTHON CODE

1.GUI CODE

```
import customtkinter as ctk
from tkinter import filedialog, messagebox
import pandas as pd
import joblib
import numpy as np
import datetime
import os
import matplotlib.pyplot as plt
from PIL import Image, ImageTk
from sklearn.metrics import accuracy_score
# === Load model and scaler ===
MODEL_PATH = r"C:\Final Year Project(SEMS)\1\Latest\trained_ml_model.pkl"
SCALER_PATH = r"C:\Final Year Project(SEMS)\1\Latest\scaler.pkl"
model = joblib.load(MODEL_PATH)
scaler = joblib.load(SCALER_PATH)
# === Configure UI ===
ctk.set_appearance_mode("System")
ctk.set_default_color_theme("blue")
app = ctk.CTk()
app.title("EMG Classifier")
app.geometry("600x600")
app.resizable(False, False)
# === Fonts ===
TITLE_FONT = ctk.CTkFont(size=20, weight="bold")
TEXT_FONT = ctk.CTkFont(size=14)
# === Labels ===
label_title = ctk.CTkLabel(app, text="EMG Classifier", font=TITLE_FONT)
label_title.pack(pady=(20, 10))
result_label = ctk.CTkLabel(app, text="Upload a CSV to start", font=TEXT_FONT)
```

```
result_label.pack(pady=(0, 20))

# === Globals ===
last_predictions = None
last_file_path = None

# === Predict Function ===
def predict_file():
    global last_predictions, last_file_path
    file_path = filedialog.askopenfilename(filetypes=[("CSV files", "*.csv")])
    if not file_path:
        return
    try:
        df = pd.read_csv(file_path)
        last_file_path = file_path


        scaled = scaler.transform(df.drop(columns=["Label"], errors="ignore"))
        predictions = model.predict(scaled)
        last_predictions = predictions
        normal = np.sum(predictions == 0)
        abnormal = np.sum(predictions == 1)
        color = "green" if abnormal == 0 else "red" if normal == 0 else "orange"
        result_label.configure(
            text=f"✅ Prediction Complete\nNormal: {normal}\nAbnormal: {abnormal}",
            text_color=color
        )
    except Exception as e:
        messagebox.showerror("Error", f"Prediction failed:\n{str(e)}")
# === Accuracy / Chart Display ===
def show_chart():
    if last_predictions is None or last_file_path is None:
        messagebox.showinfo("Info", "Run a prediction first.")
        return
    try:
        df = pd.read_csv(last_file_path)
        if "Label" not in df.columns:
```

```
            messagebox.showinfo("Info", "No 'Label' column found in file.")
            return

        y_true = df["Label"].values
        y_pred = last_predictions
        acc = accuracy_score(y_true, y_pred) * 100
        correct = np.sum(y_true == y_pred)
        incorrect = len(y_true) - correct
        # Plot accuracy bar chart
        plt.figure(figsize=(6, 3))
        bars = plt.bar(["Correct", "Incorrect"], [correct, incorrect], color=["green", "red"])
        for bar in bars:
            height = bar.get_height()
            plt.text(bar.get_x() + bar.get_width()/2, height + 1, f"{height:.0f}", ha='center')
        plt.title(f"Prediction Accuracy: {acc:.2f}%")
        plt.ylabel("Sample Count")
        plt.tight_layout()
        chart_path = "accuracy_chart.png"
        plt.savefig(chart_path)
        plt.close()
        img = Image.open(chart_path).resize((500, 250))
        chart_img = ImageTk.PhotoImage(img)
        image_label.configure(image=chart_img)
        image_label.image = chart_img
    except Exception as e:
        messagebox.showerror("Error", f"Chart generation failed:\n{str(e)}")
# === Export Results ===
def export_result():
    if last_predictions is None:
        messagebox.showinfo("Info", "No prediction to export.")
        return
    try:
        df = pd.read_csv(last_file_path)
    except:
```

```
        df = pd.DataFrame()
    normal = np.sum(last_predictions == 0)
    abnormal = np.sum(last_predictions == 1)
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    result_text = f"EMG Prediction Result - {timestamp}\n"
    result_text += f"Total samples: {len(last_predictions)}\nNormal: {normal}\nAbnormal: {abnormal}\n"
    if "Label" in df.columns:
        y_true = df["Label"].values
        acc = accuracy_score(y_true, last_predictions) * 100
        result_text += f"Prediction Accuracy: {acc:.2f}%\n"
    out_file = f"emg_result_{timestamp}.txt"
    with open(out_file, "w") as f:
        f.write(result_text)
    messagebox.showinfo("Export Complete", f"Saved to:\n{os.path.abspath(out_file)}")
# === Buttons ===
ctk.CTkButton(app, text="📂 Upload CSV & Predict", command=predict_file).pack(pady=5)
ctk.CTkButton(app, text="📊 Show Accuracy Chart", command=show_chart).pack(pady=5)
ctk.CTkButton(app, text="📥 Export Result", command=export_result).pack(pady=5)
# === Image Chart Label ===
image_label = ctk.CTkLabel(app, text="")
image_label.pack(pady=20)
# === Start GUI ===
app.mainloop()
```

2.TESTING CODE

```
import pandas as pd
import numpy as np
import joblib
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
# === Load saved model and scaler ===
model = joblib.load(r"C:\Final Year Project(SEMS)\1\Latest\trained_ml_model.pkl")
scaler = joblib.load(r"C:\Final Year Project(SEMS)\1\Latest\scaler.pkl")
```

```
# === Load CSV files ===
normal_real = pd.read_csv(r"C:\Final Year Project(SEMS)\1\normal_test.csv")
abnormal_real = pd.read_csv(r"C:\Final Year Project(SEMS)\1\abnormal_test.csv")
synthetic_normal = pd.read_csv(r"C:\Final Year Project(SEMS)\1\synthetic_normal_test.csv")
synthetic_abnormal = pd.read_csv(r"C:\Final Year Project(SEMS)\1\synthetic_abnormal_test.csv")
# === Scale the data ===
normal_real_scaled = scaler.transform(normal_real)
abnormal_real_scaled = scaler.transform(abnormal_real)
synthetic_normal_scaled = scaler.transform(synthetic_normal)
synthetic_abnormal_scaled = scaler.transform(synthetic_abnormal)
# === Make predictions ===
real_normal_pred = model.predict(normal_real_scaled)
real_abnormal_pred = model.predict(abnormal_real_scaled)
synthetic_normal_pred = model.predict(synthetic_normal_scaled)
synthetic_abnormal_pred = model.predict(synthetic_abnormal_scaled)
# === Accuracy calculation ===
def get_accuracy(predictions, true_label):
    correct = np.sum(predictions == true_label)
    total = len(predictions)
    return correct / total * 100
# === Print Summary ===
def summarize_predictions(predictions, true_label):
    correct = np.sum(predictions == true_label)
    total = len(predictions)
    incorrect = total - correct
    return f"Correct: {correct} | Incorrect: {incorrect} | Accuracy: {correct/total*100:.2f}%"
print("📊 TESTING SUMMARY\n")
print("🔍 Real Normal Samples (Expected: 0):")
print(summarize_predictions(real_normal_pred, 0))
print("\n🔍 Real Abnormal Samples (Expected: 1):")
print(summarize_predictions(real_abnormal_pred, 1))
print("\n🔍 Synthetic Normal Samples (Expected: 0):")
print(summarize_predictions(synthetic_normal_pred, 0))
```

```
print("\n🔍 Synthetic Abnormal Samples (Expected: 1):")
print(summarize_predictions(synthetic_abnormal_pred, 1))
# === Classification report (optional)
X_synthetic = np.vstack([synthetic_normal_scaled, synthetic_abnormal_scaled])
y_true_synthetic = np.array([0]*len(synthetic_normal_pred) + [1]*len(synthetic_abnormal_pred))
y_pred_synthetic = np.concatenate([synthetic_normal_pred, synthetic_abnormal_pred])
print("\n📄 Classification Report (Synthetic Combined):")
print(classification_report(y_true_synthetic, y_pred_synthetic))
# === Plot accuracy bar graph ===
labels = ['Real Normal', 'Real Abnormal', 'Synthetic Normal', 'Synthetic Abnormal']
accuracies = [
    get_accuracy(real_normal_pred, 0),
    get_accuracy(real_abnormal_pred, 1),
    get_accuracy(synthetic_normal_pred, 0),
    get_accuracy(synthetic_abnormal_pred, 1)
]
plt.figure(figsize=(10, 6))
bars = plt.bar(labels, accuracies)
# Add labels on bars
for bar in bars:
    height = bar.get_height()
    plt.annotate(f'{height:.2f}%', xy=(bar.get_x() + bar.get_width() / 2, height),
                 xytext=(0, 3), textcoords='offset points', ha='center', va='bottom')
plt.ylim(0, 105)
plt.title("Prediction Accuracy on Different Test Sets")
plt.ylabel("Accuracy (%)")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

3.ML MODEL CODE

```
import pandas as pd
```

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
# Load dataset
file_path = r"C:\Final Year Project(SEMS)\final_emg_dataset_large_2000.csv"
df = pd.read_csv(file_path)
# Separate features and labels
X = df.drop(columns=["Label"])
y = df["Label"]
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Train model (Random Forest)
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
# Make predictions
y_pred = model.predict(X_test)
# Evaluate model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)
# Print results
print(f"Model Accuracy: {accuracy * 100:.2f}%")
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", report)
```

4.ESP-32 CODE

```
const int M1 = 12; //27
```

```
const int M2 = 13; //14
const int M3 = 27; //12
const int M4 = 14; //13
#include <Wire.h>
#include <Adafruit_INA219.h>
Adafruit_INA219 ina219;
const int pwmPin = 4; // Pin number for PWM output
uint32_t start_delay=0;
uint32_t on_time=300;
uint32_t off_time=300;
uint32_t dead_time=33033;
uint32_t gape=33033;
uint32_t pulse_gape=1000000;
int no_of_pulse=15;
int max_int = 10;
// Variable to store the last time the square wave toggled
unsigned long lastToggleTime = 0;
// Variable to store the state of the square wave
bool squareWaveState = true;
int count=1;
int count2=1;
float req_mA = 0; //required current intensity of the pulse
uint8_t dutyCycle = 127;
int mem = 0;
void setup() {
pinMode(M1, OUTPUT);
pinMode(M2, OUTPUT);
pinMode(M3, OUTPUT);
pinMode(M4, OUTPUT);
digitalWrite(M1,1);
digitalWrite(M2,0);
digitalWrite(M3,1);
digitalWrite(M4,0);
Serial.begin(115200);
```

```
ledcSetup(0, 300000, 8); // Channel 0, 5 kHz frequency, 8-bit
resolution
ledcAttachPin(pwmPin, 0);
ledcWrite(0, dutyCycle);
while (!Serial)
delay(1);
Serial.println("Hello!");
if (! ina219.begin())
{
Serial.println("Failed to find INA219 chip");
while (1) { delay(10); }
}
Serial.println("Measuring voltage and current with INA219 ...");
}
void loop(){
uint32_t currentTime = micros();
// Check if it's time to toggle the square wave
ledcWrite(0, dutyCycle);
if (squareWaveState && (currentTime - lastToggleTime >= start_delay)&&
count==0) {
digitalWrite(M3,0);
digitalWrite(M2,1);
// Update the last toggle time
lastToggleTime = currentTime;
count++;
}
else if (squareWaveState && (currentTime - lastToggleTime >= on_time)&&
count==1) {
squareWaveState = false;
digitalWrite(M3,1);
digitalWrite(M2,0);
// Update the last toggle time
lastToggleTime = currentTime;
count++;
```

```
}
else if (!squareWaveState && (currentTime - lastToggleTime >= dead_time)&&
count==2) {
digitalWrite(M1,0);
digitalWrite(M4,1);
// Update the last toggle time
lastToggleTime = currentTime;
count++;
}
count==3) {
squareWaveState = true;
digitalWrite(M1,1);
digitalWrite(M4,0);
// Update the last toggle time
lastToggleTime = currentTime;
}
else if (squareWaveState && (currentTime - lastToggleTime >= gape)&&
count==4) {
digitalWrite(M3,0);
digitalWrite(M2,1);
// Update the last toggle time
lastToggleTime = currentTime;
count=1;
count2++;
}
if(count2<=5)
req_mA = req_mA + (max_int/5);
else if(count2>=no_of_pulse-3)
req_mA = req_mA - (max_int/5);
else
req_mA = max_int;
int mA_val = abs(current_INA());
currentTime = micros();
while((mA_val != req_mA)&&(count==1||3)&&(currentTime - lastToggleTime >=
```

```
on_time)){
if(mA_val < req_mA){
dutyCycle = dutyCycle + 1;
ledcWrite(0, dutyCycle);
}
else if(mA_val > req_mA){
dutyCycle = dutyCycle - 1;
ledcWrite(0, dutyCycle);
}
currentTime = micros();
if(currentTime - lastToggleTime >= on_time){
break;
mem = dutyCycle;
}
}
while(count2>no_of_pulse)
{
digitalWrite(M3,1);
digitalWrite(M2,0);
digitalWrite(M1,1);
digitalWrite(M4,0);
dutyCycle=mem;
if(micros()-lastToggleTime>pulse_gape)
{
count=1;
count2=1;
lastToggleTime = micros();
break;
}
}
}
float current_INA(){
float shuntvoltage = 0;
float busvoltage = 0;
```

```
float current_mA = 0;
float loadvoltage = 0;
float power_mW = 0;
shuntvoltage = ina219.getShuntVoltage_mV();
busvoltage = ina219.getBusVoltage_V();
current_mA = ina219.getCurrent_mA();
power_mW = ina219.getPower_mW();
loadvoltage = busvoltage + (shuntvoltage / 1000);
Serial.print("Current: "); Serial.print(current_mA); Serial.println("
mA");
return current_mA+300;
}
```

5.BIO-AMP CANDY CODE

```
#define SAMPLE_RATE 500
#define BAUD_RATE 115200
#define INPUT_PIN A0
#define BUFFER_SIZE 128

int circular_buffer[BUFFER_SIZE];
int data_index, sum;

void setup() {
        // Serial connection begin
        Serial.begin(BAUD_RATE);
}

void loop() {
        // Calculate elapsed time
        static unsigned long past = 0;
        unsigned long present = micros();
        unsigned long interval = present - past;
```

```
        past = present;

        // Run timer
        static long timer = 0;
        timer -= interval;

        // Sample and get envelop
        if(timer < 0) {
                timer += 1000000 / SAMPLE_RATE;
                int sensor_value = analogRead(INPUT_PIN);
                int signal = EMGFilter(sensor_value);
                int envelop = getEnvelop(abs(signal));
                Serial.print(signal);
                Serial.print(",");
                Serial.println(envelop);
        }
}
// Envelop detection algorithm
int getEnvelop(int abs_emg){
        sum -= circular_buffer[data_index];
        sum += abs_emg;
        circular_buffer[data_index] = abs_emg;
        data_index = (data_index + 1) % BUFFER_SIZE;
        return (sum/BUFFER_SIZE) * 2;
}

// Band-Pass Butterworth IIR digital filter, generated using filter_gen.py.
// Sampling rate: 500.0 Hz, frequency: [74.5, 149.5] Hz.
// Filter is order 4, implemented as second-order sections (biquads).
// Reference:
// https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.butter.html
// https://courses.ideate.cmu.edu/16-223/f2020/Arduino/FilterDemos/filter_gen.py
float EMGFilter(float input)
{
```

```
  float output = input;
  {
    static float z1, z2; // filter section state
    float x = output - 0.05159732*z1 - 0.36347401*z2;
    output = 0.01856301*x + 0.03712602*z1 + 0.01856301*z2;
    z2 = z1;
    z1 = x;
  }
  {
    static float z1, z2; // filter section state
    float x = output - -0.53945795*z1 - 0.39764934*z2;
    output = 1.00000000*x + -2.00000000*z1 + 1.00000000*z2;
    z2 = z1;
    z1 = x;
  }
  {
    static float z1, z2; // filter section state
    float x = output - 0.47319594*z1 - 0.70744137*z2;
    output = 1.00000000*x + 2.00000000*z1 + 1.00000000*z2;
    z2 = z1;
    z1 = x;
  }
  {
    static float z1, z2; // filter section state
    float x = output - -1.00211112*z1 - 0.74520226*z2;
    output = 1.00000000*x + -2.00000000*z1 + 1.00000000*z2;
    z2 = z1;
    z1 = x;
  }
  return output;
}
```