```python
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)

# ------------------------------------
# STEP 1 - DATA COLLECTION
# ------------------------------------
import pandas as pd

#Loading each CSV file
alex_walking_1 = pd.read_csv("alex_walking_1.csv")
alex_walking_2 = pd.read_csv("alex_walking_2.csv")
alex_walking_3 = pd.read_csv("alex_walking_3.csv")

alex_jumping_1 = pd.read_csv("alex_jumping_1.csv")
alex_jumping_2 = pd.read_csv("alex_jumping_2.csv")
alex_jumping_3 = pd.read_csv("alex_jumping_3.csv")

julien_walking_1 = pd.read_csv("julien_walking_1.csv")
julien_walking_2 = pd.read_csv("julien_walking_2.csv")
julien_walking_3 = pd.read_csv("julien_walking_3.csv")
julien_jumping_1 = pd.read_csv("julien_jumping_1.csv")
julien_jumping_2 = pd.read_csv("julien_jumping_2.csv")
julien_jumping_3 = pd.read_csv("julien_jumping_3.csv")

ella_walking_1 = pd.read_csv("ella_walking_1.csv")
ella_walking_2 = pd.read_csv("ella_walking_2.csv")
ella_walking_3 = pd.read_csv("ella_walking_3.csv")
ella_jumping_1 = pd.read_csv("ella_jumping_1.csv")
ella_jumping_2 = pd.read_csv("ella_jumping_2.csv")
ella_jumping_3 = pd.read_csv("ella_jumping_3.csv")

#Function to add activity and subject label to DataFrame
def label_activity_subject(df, activity, subject):
    df["Activity"] = activity
    df["Subject"] = subject

label_activity_subject(alex_walking_1, "walking", "alex")
label_activity_subject(alex_walking_2, "walking", "alex")
label_activity_subject(alex_walking_3, "walking", "alex")
label_activity_subject(alex_jumping_1, "jumping", "alex")
label_activity_subject(alex_jumping_2, "jumping", "alex")
label_activity_subject(alex_jumping_3, "jumping", "alex")

label_activity_subject(julien_walking_1, "walking", "julien")
label_activity_subject(julien_walking_2, "walking", "julien")
label_activity_subject(julien_walking_3, "walking", "julien")
label_activity_subject(julien_jumping_1, "jumping", "julien")
label_activity_subject(julien_jumping_2, "jumping", "julien")
label_activity_subject(julien_jumping_3, "jumping", "julien")

label_activity_subject(ella_walking_1, "walking", "ella")
label_activity_subject(ella_walking_2, "walking", "ella")
label_activity_subject(ella_walking_3, "walking", "ella")
label_activity_subject(ella_jumping_1, "jumping", "ella")
label_activity_subject(ella_jumping_2, "jumping", "ella")
label_activity_subject(ella_jumping_3, "jumping", "ella")
```

```python
#Combine all labeled datasets
combined_data = pd.concat([
    alex_walking_1, alex_walking_2, alex_walking_3,
    alex_jumping_1, alex_jumping_2, alex_jumping_3,
    julien_walking_1, julien_walking_2, julien_walking_3,
    julien_jumping_1, julien_jumping_2, julien_jumping_3,
    ella_walking_1, ella_walking_2, ella_walking_3,
    ella_jumping_1, ella_jumping_2, ella_jumping_3
], ignore_index=True)

print("Step 1 Complete: Data loaded and labeled.")

# ---------------------------------------
# STEP 2 - DATA STORAGE
# --------------------------------------
import h5py
import numpy as np

#Open HDF5 file and create raw group
hdf5_file = h5py.File("activity_data.h5", "w")
raw_group = hdf5_file.create_group("raw")

#Filtering combined data by subject
alex_data = combined_data[combined_data["Subject"] == "alex"]
julien_data = combined_data[combined_data["Subject"] == "julien"]
ella_data = combined_data[combined_data["Subject"] == "ella"]

#Function to store subject data into HDF5
def store_subject_data(h5_group, subject_name, data):
    numeric = data.select_dtypes(include=[np.number])
    strings = data.select_dtypes(include=["object"]).astype("S")
    subj_group = h5_group.create_group(subject_name)
    subj_group.create_dataset("numeric_data", data=numeric.values)
    subj_group.create_dataset("string_data", data=strings.values)
    subj_group.attrs["numeric_columns"] = list(numeric.columns)
    subj_group.attrs["string_columns"] = list(strings.columns)

#Store data for each subject
store_subject_data(raw_group, "alex", alex_data)
store_subject_data(raw_group, "julien", julien_data)
store_subject_data(raw_group, "ella", ella_data)

hdf5_file.close()
print("Step 2 Complete: Raw data stored in HDF5.")

# ------------------------------------
# STEP 3 - VISUALIZATION
# ------------------------------------

#Standardize columns in DataFrame
def standardize_columns(df):
    df.columns = [col.strip() for col in df.columns]
    mapping = {
        "Time": "Time (s)",
        "time": "Time (s)",
        "Time(s)": "Time (s)",
```

```python
        "x": "Acceleration x (m/s^2)",
        "X": "Acceleration x (m/s^2)",
        "y": "Acceleration y (m/s^2)",
        "Y": "Acceleration y (m/s^2)",
        "z": "Acceleration z (m/s^2)",
        "Z": "Acceleration z (m/s^2)",
        "Linear Acceleration x (m/s^2)": "Acceleration x (m/s^2)",
        "Linear Acceleration y (m/s^2)": "Acceleration y (m/s^2)",
        "Linear Acceleration z (m/s^2)": "Acceleration z (m/s^2)"
    }
    df.rename(columns=lambda c: mapping.get(c.strip(), c.strip()),
inplace=True)
    return df

#Load and trim CSV file based on max time
def load_trimmed(path, max_time=10):
    df = pd.read_csv(path)
    df = standardize_columns(df)
    return df[df["Time (s)"] <= max_time]

#Plot acceleration data on given axis
def plot_accel(ax, df, title):
    ax.plot(df["Time (s)"], df["Acceleration x (m/s^2)"], label="X-axis")
    ax.plot(df["Time (s)"], df["Acceleration y (m/s^2)"], label="Y-axis")
    ax.plot(df["Time (s)"], df["Acceleration z (m/s^2)"], label="Z-axis")
    ax.set_title(title)
    ax.set_ylabel("Accel (m/s²)")
    ax.grid(True)
    ax.legend()

#Visualization sample
julien_walk = load_trimmed("julien_walking_1.csv")
julien_jump = load_trimmed("julien_jumping_1.csv")
ella_walk   = load_trimmed("ella_walking_1.csv")
ella_jump   = load_trimmed("ella_jumping_1.csv")
alex_walk   = load_trimmed("alex_walking_1.csv")
alex_jump   = load_trimmed("alex_jumping_1.csv")


#Create subplots for each sample plot
fig, axs = plt.subplots(6, 1, figsize=(14, 16), sharex=True)
plot_accel(axs[0], julien_walk, "Julien - Walking")
plot_accel(axs[1], julien_jump, "Julien - Jumping")
plot_accel(axs[2], ella_walk,   "Ella - Walking")
plot_accel(axs[3], ella_jump,   "Ella - Jumping")
plot_accel(axs[4], alex_walk,   "Alex - Walking")
plot_accel(axs[5], alex_jump,   "Alex - Jumping")

axs[5].set_xlabel("Time (s)")
plt.tight_layout()
plt.show() #Display plots

# -------------------------------------
# STEP 4 - PREPROCESSING
# --------------------------------------

#Fill missing values using forward filling
```

```
preprocessed_data = combined_data.copy()
preprocessed_data.ffill(inplace=True)

#Apply moving average
cols_to_smooth = [
    "Acceleration x (m/s^2)",
    "Acceleration y (m/s^2)",
    "Acceleration z (m/s^2)",
    "Absolute acceleration (m/s^2)"
]
for c in cols_to_smooth:
    if c in preprocessed_data.columns:
        preprocessed_data[c] = preprocessed_data[c].rolling(window=10,
center=True, min_periods=1).mean()

#plot raw vs. smoothed of first 300 julien samples
julien_raw = combined_data[combined_data["Subject"] ==
"julien"].reset_index(drop=True)
julien_pre = preprocessed_data[preprocessed_data["Subject"] ==
"julien"].reset_index(drop=True)

plt.figure(figsize=(10, 4))
plt.plot(
    julien_raw["Time (s)"][:300],
    julien_raw["Absolute acceleration (m/s^2)"][:300],
    label="Raw Abs Accel", alpha=0.5
)
plt.plot(
    julien_pre["Time (s)"][:300],
    julien_pre["Absolute acceleration (m/s^2)"][:300],
    label="Smoothed Abs Accel", linewidth=2
)
plt.title("Absolute Acceleration: Raw vs Smoothed (Julien, first 300
samples)")
plt.xlabel("Time (s)")
plt.ylabel("Acceleration (m/s²)")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()

print("Step 4 Complete: Missing values handled, noise reduced with moving
average.")

# ----------------------------------------
# STEP 5 - FEATURE EXTRACTION + NORMALIZATION
# ----------------------------------------
from scipy.stats import skew, kurtosis
from sklearn.preprocessing import StandardScaler

data = preprocessed_data.copy()

#Determine sampling rate from time differences
time_diffs = data["Time (s)"].diff().dropna()
sampling_rate = 1 / time_diffs.median()
samples_per_window = int(5 * sampling_rate)
```

```
#Specify columns for feature extraction
feature_cols = [
    "Acceleration x (m/s^2)",
    "Acceleration y (m/s^2)",
    "Acceleration z (m/s^2)",
    "Absolute acceleration (m/s^2)"
]

#Function to extract statistical features from a window
def extract_features(window):
    feats = {}
    for col in feature_cols:
        if col not in window.columns:
            continue
        col_data = window[col]
        feats[f"{col}_mean"] = col_data.mean()
        feats[f"{col}_std"] = col_data.std()
        feats[f"{col}_min"] = col_data.min()
        feats[f"{col}_max"] = col_data.max()
        feats[f"{col}_skew"] = skew(col_data)
        feats[f"{col}_kurtosis"] = kurtosis(col_data)
        feats[f"{col}_range"] = col_data.max() - col_data.min()
        feats[f"{col}_median"] = col_data.median()
        feats[f"{col}_var"] = col_data.var()
        feats[f"{col}_mad"] = np.mean(np.abs(col_data - col_data.mean()))
    return feats

X_list = []
y_list = []

# Group by subject, activity
for (subj, act), group_df in data.groupby(["Subject", "Activity"]):
    group_df = group_df.reset_index(drop=True)
    for start in range(0, len(group_df) - samples_per_window,
samples_per_window):
        window = group_df.iloc[start:start + samples_per_window]
        if len(window) == samples_per_window:
            feats = extract_features(window)
            X_list.append(feats)
            y_list.append(act)

X_df = pd.DataFrame(X_list)
y_series = pd.Series(y_list, name="Label")
X_df.fillna(X_df.mean(), inplace=True) #handle missing feature values

#Normalize features using StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_df)
X_scaled_df = pd.DataFrame(X_scaled, columns=X_df.columns)
X_scaled_df["Label"] = y_series.values

print("Step 5 Complete: Extracted and normalized features.")

# ----------------------------------------
# STEP 6 - CLASSIFIER TRAINING + EVALUATION
# ----------------------------------------
from sklearn.model_selection import train_test_split, learning_curve
```

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
ConfusionMatrixDisplay
import joblib

#Preparing features for training
X = X_scaled_df.drop("Label", axis=1)
y = X_scaled_df["Label"]

#Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.1,
    random_state=42,
    stratify=y
)

#Training logistic regression model
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

#Compute model accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy = {accuracy * 100:.2f}%")

#Compute and display confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=model.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=model.classes_)
disp.plot(cmap="Blues")
plt.title("Confusion Matrix")
plt.grid(False)
plt.tight_layout()
plt.show()

#Generate learning curve for further evaluation
train_sizes, train_scores, valid_scores = learning_curve(
    estimator=LogisticRegression(max_iter=1000),
    X=X_train,
    y=y_train,
    train_sizes=np.linspace(0.1, 1.0, 5),
    cv=5,
    scoring='accuracy',
    shuffle=True,
    random_state=42
)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
valid_mean = np.mean(valid_scores, axis=1)
valid_std = np.std(valid_scores, axis=1)

plt.figure()
plt.plot(train_sizes, train_mean, 'o-', label='Training Accuracy')
plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std,
```

```
alpha=0.1)
plt.plot(train_sizes, valid_mean, 'o-', label='Validation Accuracy')
plt.fill_between(train_sizes, valid_mean - valid_std, valid_mean + valid_std,
alpha=0.1)
plt.title("Learning Curve (Logistic Regression)")
plt.xlabel("Number of Training Samples")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)
plt.show()

print("Step 6 Complete: Model trained, tested, and learning curves
generated.")

#Save trained model and scaler for future use (GUI)
joblib.dump(model, "logreg_model.pkl")
joblib.dump(scaler, "scaler.pkl")
print("Saved logistic regression model to 'logreg_model.pkl' and scaler to
'scaler.pkl'.")

# --------------------------------------
# STEP 7 - STORE PREPROCESSED + SEGMENTED INTO HDF5
# -------------------------------------
with h5py.File("activity_data.h5", "a") as h5f:
    #Storing preprocessed data per subject
    preprocessed_group = h5f.require_group("preprocessed")
    for subject in preprocessed_data["Subject"].unique():
        subject_data = preprocessed_data[preprocessed_data["Subject"] ==
subject]
        numeric = subject_data.select_dtypes(include=[np.number])
        strings = subject_data.select_dtypes(include=["object"]).astype("S")

        subj_group = preprocessed_group.create_group(subject)
        subj_group.create_dataset("numeric_data", data=numeric.values)
        subj_group.create_dataset("string_data", data=strings.values)
        subj_group.attrs["numeric_columns"] = list(numeric.columns)
        subj_group.attrs["string_columns"] = list(strings.columns)

    print("Preprocessed data stored into HDF5.")

    segmented_group = h5f.require_group("segmented")
    segmented_group.create_dataset("train", data=X_train.values)
    segmented_group.create_dataset("train_labels",
data=np.array(y_train).astype("S"))
    segmented_group.create_dataset("test", data=X_test.values)
    segmented_group.create_dataset("test_labels",
data=np.array(y_test).astype("S"))

    print("Segmented train/test data stored into HDF5.")
```

GRAPHICAL USER INTERFACE CODE:

```
import tkinter as tk
from tkinter import filedialog, messagebox, scrolledtext
import pandas as pd
import numpy as np
import joblib

import matplotlib
matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from scipy.stats import skew, kurtosis
import matplotlib.patches as mpatches

#Loading trained model and scaler
model = joblib.load("logreg_model.pkl")
scaler = joblib.load("scaler.pkl")

#Data columns
TIME_COL = "Time (s)"
AX_COL   = "Acceleration x (m/s^2)"
AY_COL   = "Acceleration y (m/s^2)"
AZ_COL   = "Acceleration z (m/s^2)"
ABS_COL  = "Absolute acceleration (m/s^2)"

feature_cols = [AX_COL, AY_COL, AZ_COL, ABS_COL]

#Defining helper functions
def fill_missing_values(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.ffill(inplace=True)
    return df

def smooth_signals(df: pd.DataFrame, window_size=10) -> pd.DataFrame:
    df = df.copy()
    for col in feature_cols:
        if col in df.columns:
            df[col] = df[col].rolling(window=window_size, center=True,
min_periods=1).mean()
    return df

def extract_window_features(window: pd.DataFrame) -> dict:
    feats = {}
    for col in feature_cols:
        if col not in window.columns:
            continue
        col_data = window[col]
        feats[f"{col}_mean"]     = col_data.mean()
        feats[f"{col}_std"]      = col_data.std()
        feats[f"{col}_min"]      = col_data.min()
        feats[f"{col}_max"]      = col_data.max()
        feats[f"{col}_skew"]     = skew(col_data)
        feats[f"{col}_kurtosis"] = kurtosis(col_data)
        feats[f"{col}_range"]    = col_data.max() - col_data.min()
        feats[f"{col}_median"]   = col_data.median()
```

```
        feats[f"{col}_var"]       = col_data.var()
        feats[f"{col}_mad"]       = np.mean(np.abs(col_data - 
col_data.mean()))
    return feats

def segment_and_extract(df: pd.DataFrame, window_sec=5):
    dt = df[TIME_COL].diff().dropna()
    if dt.empty or dt.median() == 0:
        return pd.DataFrame(), []

    sampling_rate = 1 / dt.median()
    samples_per_window = int(window_sec * sampling_rate)

    X_list = []
    window_times = []
    start_idx = 0
    while start_idx + samples_per_window <= len(df):
        window_data = df.iloc[start_idx : start_idx + samples_per_window]
        feats = extract_window_features(window_data)
        X_list.append(feats)

        # Record the time span
        t0 = window_data[TIME_COL].iloc[0]
        t1 = window_data[TIME_COL].iloc[-1]
        window_times.append((t0, t1))

        start_idx += samples_per_window

    X_df = pd.DataFrame(X_list)
    #filling any potential gaps
    if not X_df.empty:
        X_df.fillna(X_df.mean(), inplace=True)
    return X_df, window_times

#Classify Data and show in GUI
def classify_csv(file_path: str, output_text: tk.Text, plot_frame: tk.Frame):
    output_text.delete("1.0", tk.END)

    #Load CSV
    try:
        df = pd.read_csv(file_path)
    except Exception as e:
        messagebox.showerror("Error", f"Failed to read CSV:\n{e}")
        return

    if TIME_COL not in df.columns:
        messagebox.showerror("Error", f"CSV missing '{TIME_COL}' column.")
        return

    #fill + smooth
    df = fill_missing_values(df)
    df = smooth_signals(df, window_size=10)

    # segment + extract
    X_df, window_times = segment_and_extract(df, window_sec=5)
    if X_df.empty:
        messagebox.showwarning("No windows", "No complete 5-second windows
```

```
found.")
            return

    X_scaled = scaler.transform(X_df)

    #predict
    preds = model.predict(X_scaled)

    #Display textual results
    output_text.insert(tk.END, f"File: {file_path}\n\n")
    output_text.insert(tk.END, "WINDOW_ID   TIME_RANGE         PREDICTION\n")
    for i, (start_t, end_t) in enumerate(window_times):
        label = preds[i]
        window_id = i + 1
        output_text.insert(
            tk.END,
            f"{window_id:>9}   [{start_t:.2f} - {end_t:.2f}]   {label}\n"
        )

    #Plot
    for widget in plot_frame.winfo_children():
        widget.destroy()

    fig, ax = plt.subplots(figsize=(6,4), dpi=100)
    line_x, = ax.plot(df[TIME_COL], df[AX_COL], label="Accel X")
    line_y, = ax.plot(df[TIME_COL], df[AY_COL], label="Accel Y")
    line_z, = ax.plot(df[TIME_COL], df[AZ_COL], label="Accel Z")

    ax.set_xlabel("Time (s)")
    ax.set_ylabel("Acceleration (m/s^2)")
    ax.set_title("Accelerations with Predicted Windows")
    ax.grid(True)

    y_min, y_max = ax.get_ylim()
    for i, (start_t, end_t) in enumerate(window_times):
        label = preds[i]
        color = "green" if label == "walking" else "red"
        ax.add_patch(Rectangle(
            (start_t, y_min),
            end_t - start_t,
            y_max - y_min,
            color=color,
            alpha=0.1
        ))

    #Build one legend that includes X, Y, Z plus walking/jumping patches
    walk_patch = mpatches.Patch(color='green', alpha=0.1, label='Walking
Window')
    jump_patch = mpatches.Patch(color='red', alpha=0.1, label='Jumping
Window')

    #Current line handles + labels
    lines, labels = ax.get_legend_handles_labels()

    lines += [walk_patch, jump_patch]
    labels += ["Walking Window", "Jumping Window"]
```

```python
    ax.legend(lines, labels, loc="upper right")

    canvas = FigureCanvasTkAgg(fig, master=plot_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)

#MAIN GUI
def open_file_dialog(output_text: tk.Text, plot_frame: tk.Frame):
    file_path = filedialog.askopenfilename(
        title="Select Accelerometer CSV",
        filetypes=[("CSV Files", "*.csv"), ("All Files", "*.*")]
    )
    if file_path:
        classify_csv(file_path, output_text, plot_frame)

def main():
    root = tk.Tk()
    root.title("Walking vs Jumping Classifier")

    #top frame for buttons
    top_frame = tk.Frame(root)
    top_frame.pack(side=tk.TOP, fill=tk.X)

    #Scrolled text for output
    output_text = scrolledtext.ScrolledText(root, width=70, height=22,
font=("Courier", 10))
    output_text.pack(side=tk.BOTTOM, fill=tk.X, padx=5, pady=5)

    #Frame for plot
    plot_frame = tk.Frame(root)
    plot_frame.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=True)

    browse_btn = tk.Button(top_frame, text="Browse CSV", command=lambda:
open_file_dialog(output_text, plot_frame))
    browse_btn.pack(side=tk.LEFT, padx=10, pady=10)

    quit_btn = tk.Button(top_frame, text="Quit", command=root.destroy)
    quit_btn.pack(side=tk.RIGHT, padx=10, pady=10)

    root.mainloop()

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