```
import os
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import io
from scipy.signal import butter, lfilter, freqz
from statistics import median
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix
from sklearn.datasets import make_blobs
WINDOW_SIZE = 150    # 20:9.76ms, 150:73.2ms
TEST_RATIO = 0.3
SEGMENT_N = 3
PLOT_SCATTERED_DATA = False
PLOT_CONFUSION_MATRIX = True

def load_mat_files(dataDir):
    mats = []
    for file in os.listdir(dataDir):
        mats.append(io.loadmat(dataDir+file)['gestures'])
    return mats

def butter_bandpass_filter(data, lowcut=20.0, highcut=400.0, fs=2048, order=4):
    nyq = 0.5 * fs
    low = lowcut / nyq
    high = highcut / nyq
    b, a = butter(order, [low, high], btype='band')
    y = lfilter(b, a, data)
    return y

def plot_bandpass_filtered_data(data):
    plt.figure(1)
    plt.clf()
    plt.plot(data, label='Noisy signal')

    y = butter_bandpass_filter(data)
    plt.plot(y, label='Filtered signal')
    plt.xlabel('time (seconds)')
    plt.grid(True)
    plt.axis()
    plt.legend(loc='upper left')
    plt.show()

def divide_to_windows(datas, window_size=WINDOW_SIZE):
    windows=np.delete(datas,
list(range((len(datas)//window_size)*window_size,len(datas))))
    windows=np.reshape(windows,((len(datas)//window_size,window_size)))
    return windows

def compute_RMS(datas):
    return np.sqrt(np.mean(np.array(datas)**2))

def compute_RMS_gestures(gestures):
    RMS_gestures=np.array([[[[0.0 for i_ch in range(gestures.shape[3])] for i_win in
range(gestures.shape[2])] for i_try in range(gestures.shape[1])] for i_ges in
range(gestures.shape[0])])
    for i_ges in range(gestures.shape[0]):
        for i_try in range(gestures.shape[1]):
            for i_win in range(gestures.shape[2]):
```

```
                for i_ch in range(gestures.shape[3]):
                    RMS_gestures[i_ges][i_try][i_win]
[i_ch]=compute_RMS(gestures[i_ges][i_try][i_win][i_ch])
    return RMS_gestures

def create_168_dimensional_window_vectors(channels):
    for i_ch in range(len(channels)):
        # Segmentation : Data processing : Discard useless data
        if (i_ch+1)%8 == 0:
            continue
        # Preprocessing : Apply butterworth band-pass filter]
        filtered_channel=butter_bandpass_filter(channels[i_ch])
        # Segmentation : Data processing : Divide continuous data into 150 samples
window
        windows_per_channel=divide_to_windows(filtered_channel)     #
windows_per_channel : (40, 150)
        if i_ch==0:
            pre_processed_one_try=np.array(windows_per_channel)
            continue
        pre_processed_one_try=np.append(pre_processed_one_try, windows_per_channel,
axis=1) # Adding column
    return np.reshape(pre_processed_one_try,
(pre_processed_one_try.shape[0],-1,WINDOW_SIZE))

def average_for_channel(gesture):
    average=np.array([])
    for i_ch in range(gesture.shape[2]):
        sum=0
        for i_win in range(gesture.shape[1]):
            for i_try in range(gesture.shape[0]):
                sum+=gesture[i_try][i_win][i_ch]
        average=np.append(average, [sum/(gesture.shape[1]*gesture.shape[0])])
    return average

def base_normalization(RMS_gestures):
    average_channel_idle_gesture=average_for_channel(RMS_gestures[0])
    for i_ges in range(RMS_gestures.shape[0]):   # Including idle gesture
        for i_try in range(RMS_gestures.shape[1]):
            for i_win in range(RMS_gestures.shape[2]):
                for i_ch in range(RMS_gestures.shape[3]):
                    RMS_gestures[i_ges][i_try][i_win][i_ch]-
=average_channel_idle_gesture[i_ch]
    return RMS_gestures

def extract_ACTIVE_window_i(RMS_gestures):
    for i_ges in range(len(RMS_gestures)):
        for i_try in range(len(RMS_gestures[i_ges])):
            # Segmentation : Determine whether ACTIVE : Compute summarized RMS
            sum_RMSs=[sum(window) for window in RMS_gestures[i_ges][i_try]]
            threshold=sum(sum_RMSs)/len(sum_RMSs)
            # Segmentation : Determine whether ACTIVE
            i_ACTIVEs=[]
            for i_win in range(len(RMS_gestures[i_ges][i_try])):
                if sum_RMSs[i_win] > threshold and i_win>0:     # Exclude 0th index
                    i_ACTIVEs.append(i_win)
            for i in range(len(i_ACTIVEs)):
                if i==0:
                    continue
                if i_ACTIVEs[i]-i_ACTIVEs[i-1] == 2:
                    i_ACTIVEs.insert(i, i_ACTIVEs[i-1]+1)
```

```
            # Segmentation : Determine whether ACTIVE : Select the longest contiguous sequences
            segs=[]
            contiguous = 0
            for i in range(len(i_ACTIVEs)):
                if i == len(i_ACTIVEs)-1:
                    if contiguous!=0:
                        segs.append((start, contiguous))
                    break
                if i_ACTIVEs[i+1]-i_ACTIVEs[i] == 1:
                    if contiguous == 0:
                        start=i_ACTIVEs[i]
                    contiguous+=1
                else:
                    if contiguous != 0:
                        contiguous+=1
                        segs.append((start, contiguous))
                        contiguous=0
            seg_start, seg_len = sorted(segs, key=lambda seg: seg[1], reverse=True)[0]
            # Segmentation : Return ACTIVE window indexes
            if i_try==0:
                i_one_try_ACTIVE = np.array([[seg_start, seg_len]])
                continue
            i_one_try_ACTIVE = np.append(i_one_try_ACTIVE, [[seg_start, seg_len]], axis=0)
        if i_ges==0:
            i_ACTIVE_windows = np.array([i_one_try_ACTIVE])
            continue
        i_ACTIVE_windows = np.append(i_ACTIVE_windows, [i_one_try_ACTIVE], axis=0)
    return i_ACTIVE_windows

def medfilt(channel, kernel_size=3):
    filtered=np.zeros(len(channel))
    for i in range(len(channel)):
        if i-kernel_size//2 <0 or i+kernel_size//2 >=len(channel):
            continue
        filtered[i]=median([channel[j] for j in range(i-kernel_size//2, i+kernel_size//2+1)])
    return filtered

def ACTIVE_filter(i_ACTIVE_windows, pre_processed_gestures):
    # ACTIVE_filter : delete if the window is not ACTIVE
    list_pre_processed_gestures=pre_processed_gestures.tolist()
    for i_ges in range(len(list_pre_processed_gestures)):
        for i_try in range(len(list_pre_processed_gestures[i_ges])):
            for i_win in reversed(range(len(list_pre_processed_gestures[i_ges][i_try]))):
                if not i_win in range(i_ACTIVE_windows[i_ges][i_try][0], i_ACTIVE_windows[i_ges][i_try][0]+i_ACTIVE_windows[i_ges][i_try][1]):
                    del list_pre_processed_gestures[i_ges][i_try][i_win]
    return np.array(list_pre_processed_gestures)

def Repartition_N_Compute_RMS(ACTIVE_pre_processed_gestures, N=SEGMENT_N):
    # List all the data of each channel without partitioning into windows
    ACTIVE_N_gestures=[[[[] for i_ch in range(len(ACTIVE_pre_processed_gestures[0][0][0]))] for i_try in range(ACTIVE_pre_processed_gestures.shape[1])] for i_ges in range(ACTIVE_pre_processed_gestures.shape[0])]     # CONSTANT
    for i_ges in range(len(ACTIVE_pre_processed_gestures)):
        for i_try in range(len(ACTIVE_pre_processed_gestures[i_ges])):
```

```
            for i_seg in range(len(ACTIVE_pre_processed_gestures[i_ges][i_try])):
                for i_ch in range(len(ACTIVE_pre_processed_gestures[i_ges][i_try][i_seg])):
                    ACTIVE_N_gestures[i_ges][i_try][i_ch].extend(ACTIVE_pre_processed_gestures[i_ges][i_try][i_seg][i_ch])
    # Compute RMS in N large windows
    for i_ges in range(len(ACTIVE_N_gestures)):
        for i_try in range(len(ACTIVE_N_gestures[i_ges])):
            for i_ch in range(len(ACTIVE_N_gestures[i_ges][i_try])):
                RMSs=[]
                for i  in range(N):
                    RMSs.append(compute_RMS(ACTIVE_N_gestures[i_ges][i_try][i_ch][(len(ACTIVE_N_gestures[i_ges][i_try][i_ch])//N)*i:(len(ACTIVE_N_gestures[i_ges][i_try][i_ch])//N)*(i+1)]))
                ACTIVE_N_gestures[i_ges][i_try][i_ch]=np.array(RMSs)
            ACTIVE_N_gestures[i_ges][i_try]=np.array(ACTIVE_N_gestures[i_ges][i_try]).transpose()   # Change (4,10,168,N) -> (4,10,N,168)
    return np.array(ACTIVE_N_gestures)

def mean_normalization(ACTIVE_N_RMS_gestures):
    for i_ges in range(len(ACTIVE_N_RMS_gestures)):
        for i_try in range(len(ACTIVE_N_RMS_gestures[i_ges])):
            for i_Lwin in range(len(ACTIVE_N_RMS_gestures[i_ges][i_try])):
                delta=max(ACTIVE_N_RMS_gestures[i_ges][i_try][i_Lwin])-min(ACTIVE_N_RMS_gestures[i_ges][i_try][i_Lwin])
                Mean=np.mean(ACTIVE_N_RMS_gestures[i_ges][i_try][i_Lwin])
                for i_ch in range(len(ACTIVE_N_RMS_gestures[i_ges][i_try][i_Lwin])):
                    ACTIVE_N_RMS_gestures[i_ges][i_try][i_Lwin][i_ch]=(ACTIVE_N_RMS_gestures[i_ges][i_try][i_Lwin][i_ch]-Mean)/delta
    return ACTIVE_N_RMS_gestures

def construct_X_y(mean_normalized_RMS):
    X=np.reshape(mean_normalized_RMS, (mean_normalized_RMS.shape[0]*mean_normalized_RMS.shape[1]*mean_normalized_RMS.shape[2], mean_normalized_RMS.shape[3]))
    y=np.array([])
    for i_ges in range(mean_normalized_RMS.shape[0]):
        for i_try in range(mean_normalized_RMS.shape[1]):
            for i_Lwin in range(mean_normalized_RMS.shape[2]):
                y=np.append(y, [i_ges])
    return X, y

def plot_confusion_matrix(y_test, kinds, y_pred):
    mat = confusion_matrix(y_test, y_pred)
    sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False, xticklabels=kinds, yticklabels=kinds)
    plt.xlabel('true label')
    plt.ylabel('predicted label')
    plt.axis('auto')
    plt.show()

def plot_scattered_data(X, y):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
    plt.show()

def check(x, prin=0):
    print("length: ", len(x))
    print("type: ", type(x))
    print("shape: ", x.shape)
    if prin==1: print(x)
```

```
    raise ValueError("-------------WORKING LINE--------------")

def check_segment_len(ACTIVE_RMS_gestures):
    for i in range(len(ACTIVE_RMS_gestures)):
        print("%d번째 gesture의 각 try의 segment 길이들 : " %i, end='')
        for j in range(len(ACTIVE_RMS_gestures[i])):
            print(len(ACTIVE_RMS_gestures[i][j]), end=' ')
        print()

def main():
    #loading .mat files consist of 0,1,2,3(,11,17,18,21,23,24,25 not for light) gestures
    gestures = load_mat_files("./data ref1_subject1_session1_light/")  # gestures : list
    #In idle gesture, we just use 2,4,7,8,11,13,19,25,26,30th tries in order to match the number of datas
    gestures[0]=gestures[0][[1,3,6,7,10,12,18,24,25,29]]

    # Signal Pre-processing & Construct windows
    init_gesture=1
    for gesture in gestures:
        init_try=1
        for one_try in gesture:
            pre_processed_one_try = create_168_dimensional_window_vectors(one_try[0]) # one_try[0] : channels, ndarray
            if init_try == 1:
                pre_processed_tries_for_gesture = np.array([pre_processed_one_try])
                init_try=0
                continue
            pre_processed_tries_for_gesture = np.append(pre_processed_tries_for_gesture, [pre_processed_one_try], axis=0)    # Adding height
        if init_gesture==1:
            pre_processed_gestures = np.array([pre_processed_tries_for_gesture])
            init_gesture=0
            continue
        pre_processed_gestures = np.append(pre_processed_gestures, [pre_processed_tries_for_gesture], axis=0)   # Adding blocks

    # Segmentation : Compute RMS
    RMS_gestures=compute_RMS_gestures(pre_processed_gestures)
    # Segmentation : Base normalization
    RMS_gestures=base_normalization(RMS_gestures)
    # Segmentation : Median filtering
    for i_ges in range(len(RMS_gestures)):
        for i_try in range(len(RMS_gestures[i_ges])):
            channels=RMS_gestures[i_ges][i_try].transpose()
            for i_ch in range(len(channels)):
                channels[i_ch]=medfilt(channels[i_ch])
            RMS_gestures[i_ges][i_try]=channels.transpose()
    # Segmentation : Dertermine which window is ACTIVE
    i_ACTIVE_windows=extract_ACTIVE_window_i(RMS_gestures.tolist())

    # Feature extraction : Filter only ACTIVE windows
    ACTIVE_pre_processed_gestures=ACTIVE_filter(i_ACTIVE_windows, pre_processed_gestures)
    # Feature extraction : Partition existing windows into N large windows and compute RMS for each large window
    ACTIVE_N_RMS_gestures=Repartition_N_Compute_RMS(ACTIVE_pre_processed_gestures, SEGMENT_N)
```

```
264     # Feature extraction : Mean normalization for all channels in each window
265     mean_normalized_RMS=mean_normalization(ACTIVE_N_RMS_gestures)
266 
267     # Naive Bayes classifier : Construct X and y
268     X, y = construct_X_y(mean_normalized_RMS)
269     kinds=[i_ges for i_ges in range(mean_normalized_RMS.shape[0])]
270     # Naive Bayes classifier : Basic method : NOT LOOCV
271     gnb = GaussianNB()
272     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_RATIO,
    random_state=0)
273     y_pred = gnb.fit(X_train, y_train).predict(X_test)
274     if PLOT_SCATTERED_DATA:
275         plot_scattered_data(X_test, y_pred)
276     print("Number of mislabeled prediction out of a total %d prediction : %d" %
    (X_test.shape[0], (y_test != y_pred).sum()))
277     if PLOT_CONFUSION_MATRIX:
278         plot_confusion_matrix(y_test, kinds, y_pred)
279 
280 main()
```