Hyedora / 2020_Summer_Individual_study

Code Issues Pull requests Actions Projects Wiki Security Insights •••

master ▾ •••

2020_Summer_Individual_study / Building_ref1.py / <> Jump to ▾

Hyedora Plotting scattered data left

1 contributor

Raw Blame

279 lines (255 sloc) 11.7 KB

```
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
CLASSIFYING_METHOD = 2  # 1 or 2
SEGMENT_N = 3
PLOT_SCATTERED_DATA = True
PLOT_CONFUSION_MATRIX = False

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)
```

```python
    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(datas**2))

def compute_RMS_for_each_windows(windows):
    init=1
    for window in windows:
        if init==1:
            RMSs=np.array([[compute_RMS(window)]])
            init=0
            continue
        RMSs=np.append(RMSs, [[compute_RMS(window)]], axis=0)
    return RMSs

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)
        # Segmentation : Compute RMS for each channel
        RMSwindows_per_channel=compute_RMS_for_each_windows(windows_per_channel)
        if i_ch==0:
            RMS_one_try=np.array(RMSwindows_per_channel)
            continue
        RMS_one_try=np.append(RMS_one_try, RMSwindows_per_channel, axis=1)  # Adding column
    return RMS_one_try

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 ACTIVE_filter(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 sequence
            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 : Determine whether ACTIVE : delete if the window is not ACTIVE
            for i_win in reversed(range(len(RMS_gestures[i_ges][i_try]))):
                if not i_win in range(seg_start, seg_start+seg_len):
                    del RMS_gestures[i_ges][i_try][i_win]
    return RMS_gestures

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 mean_normalization(ACTIVE_RMS_gestures):
    for i_ges in range(len(ACTIVE_RMS_gestures)):
        for i_try in range(len(ACTIVE_RMS_gestures[i_ges])):
            for i_win in range(len(ACTIVE_RMS_gestures[i_ges][i_try])):
                delta=max(ACTIVE_RMS_gestures[i_ges][i_try][i_win])-min(ACTIVE_RMS_gestures[i_
                Mean=np.mean(ACTIVE_RMS_gestures[i_ges][i_try][i_win])
                for i_ch in range(len(ACTIVE_RMS_gestures[i_ges][i_try][i_win])):
                    ACTIVE_RMS_gestures[i_ges][i_try][i_win][i_ch]=(ACTIVE_RMS_gestures[i_ges]
    return ACTIVE_RMS_gestures

def segment_windowing(mean_normalized_RMS,CLASSIFYING_METHOD,N=3):
    gesture_flattened = np.reshape(mean_normalized_RMS, -1)
    if CLASSIFYING_METHOD==1:
        init_try=1
        for segment in gesture_flattened:
            channels=np.array(segment).transpose()
            chs_windows=np.array([])
            init_ch=1
            for channel in channels:
                ch_windows=np.array([])
                for i in range(N):
                    ch_windows=np.append(ch_windows, [compute_RMS(channel[(len(channel)//N)*i:
                if init_ch==1:
                    chs_windows=np.array([ch_windows])
                    init_ch=0
                    continue
                chs_windows=np.append(chs_windows, [ch_windows], axis=0)
            if init_try==1:
                tries_windows=np.array([chs_windows.transpose()])
                init_try=0
                continue
            tries_windows=np.append(tries_windows, [chs_windows.transpose()], axis=0)
        X=np.reshape(tries_windows, (tries_windows.shape[0],tries_windows.shape[1]*tries_windo
    elif CLASSIFYING_METHOD==2:
        init_try=1
        for segment in gesture_flattened:
            if init_try==1:
                X=np.array(np.array(segment))
                init_try=0
                continue
            X=np.append(X, np.array(segment), axis=0)
    else: raise ValueError("CLASSIFYING_METHOD only can be 1 or 2")
    return X, construct_label(mean_normalized_RMS,CLASSIFYING_METHOD)

def construct_label(mean_normalized_RMS,CLASSIFYING_METHOD):
    y=np.array([])
    if CLASSIFYING_METHOD==1:
        for i_ges in range(len(mean_normalized_RMS)):
            y=np.append(y, [i_ges for i_try in range(mean_normalized_RMS.shape[1])])
    elif CLASSIFYING_METHOD==2:
        for i_ges in range(mean_normalized_RMS.shape[0]):
            for i_try in range(mean_normalized_RMS.shape[1]):
                y=np.append(y, [i_ges for i_win in range(len(mean_normalized_RMS[i_ges][i_try]
    return 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, ytickl
    plt.xlabel('true label')
    plt.ylabel('predicted label')
    plt.axis('auto')
    plt.show()

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

def check(x):
    print("length: ", len(x))
    print("type: ", type(x))
    print("shape: ", x.shape)
    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 numbe
    gestures[0]=gestures[0][[1,3,6,7,10,12,18,24,25,29]]

    # Signal Preprocessing & Data processing for segmentation
    init_gesture=1
    for gesture in gestures:
        init_try=1
        for one_try in gesture:
            RMS_one_try = create_168_dimensional_window_vectors(one_try[0]) # one_try[0] : char
            if init_try == 1:
                RMS_tries_for_gesture = np.array([RMS_one_try])
                init_try=0
                continue
            RMS_tries_for_gesture = np.append(RMS_tries_for_gesture, [RMS_one_try], axis=0) # /
        if init_gesture==1:
            RMS_gestures = np.array([RMS_tries_for_gesture])
            init_gesture=0
            continue
        RMS_gestures = np.append(RMS_gestures, [RMS_tries_for_gesture], axis=0) # Adding block

    # Segmentation : Data processing : Base normalization
    RMS_gestures=base_normalization(RMS_gestures)
    # Segmentation : Data processing : 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
    ACTIVE_RMS_gestures=ACTIVE_filter(RMS_gestures.tolist())
    # Feature extraction : Mean normalization for all channels in each window
    mean_normalized_RMS=mean_normalization(np.array(ACTIVE_RMS_gestures))
    # Naive Bayes classifier : Construct X and y
    X, y = segment_windowing(mean_normalized_RMS,CLASSIFYING_METHOD,SEGMENT_N)
    kinds=[i_ges for i_ges in range(mean_normalized_RMS.shape[0])]
    # Naive Bayes classifier : Basic method : NOT LOOCV
    gnb = GaussianNB()
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_RATIO, random_sta
    y_pred = gnb.fit(X_train, y_train).predict(X_test)
    if PLOT_SCATTERED_DATA:
        plot_scattered_data(X_test, y_pred)
    print("Number of mislabeled prediction out of a total %d prediction : %d" % (X_test.shape[
    if PLOT_CONFUSION_MATRIX:
        plot_confusion_matrix(y_test, kinds, y_pred)

main()
```