unit_test

July 31, 2021

1 Test Your Algorithm

1.1 Instructions

1. From the **Pulse Rate Algorithm** Notebook you can do one of the following:

- Copy over all the **Code** section to the following Code block.
- Download as a Python (`.py`) and copy the code to the following Code block.

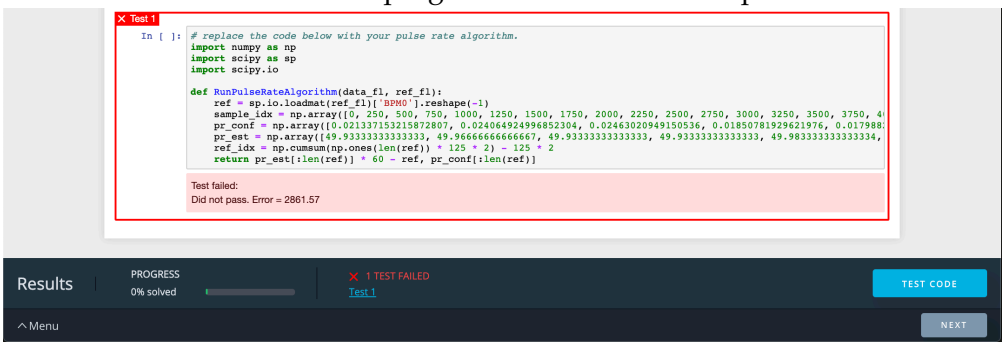
2. In the bottom right, click the Test Run button.

1.1.1 Didn't Pass

If your code didn't pass the test, go back to the previous Concept or to your local setup and continue iterating on your algorithm and try to bring your training error down before testing again.

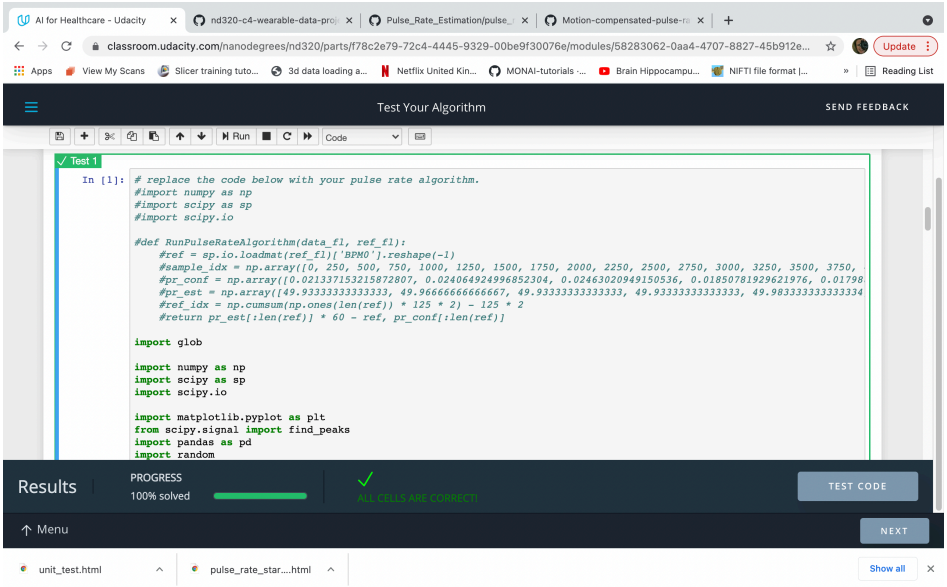
1.1.2 Pass

If your code passes the test, complete the following! You **must** include a screenshot of your code and the Test being **Passed**. Here is what the starter filler code looks like when the test is run and should be similar. A passed test will include in the notebook a green outline plus a box with **Test passed:** and in the Results bar at the bottom the progress bar will be at 100% plus a checkmark with **All cells passed**.



1. Take a screenshot of your code passing the test, make sure it is in the format `.png`. If not a `.png` image, you will have to edit the Markdown render the image after Step 3. Here is an example of what the `passed.png` would look like
2. Upload the screenshot to the same folder or directory as this jupyter notebook.

3. Rename the screenshot to `passed.png` and it should show up below.



4. Download this jupyter notebook as a `.pdf` file.
5. Continue to Part 2 of the Project.

```
In [1]: # replace the code below with your pulse rate algorithm.
        #import numpy as np
        #import scipy as sp
        #import scipy.io

        #def RunPulseRateAlgorithm(data_fl, ref_fl):
            #ref = sp.io.loadmat(ref_fl)['BPM0'].reshape(-1)
            #sample_idx = np.array([0, 250, 500, 750, 1000, 1250, 1500, 1750, 2000, 2250, 2500,
            #pr_conf = np.array([0.021337153215872807, 0.024064924996852304, 0.0246302094915053
            #pr_est = np.array([49.93333333333333, 49.96666666666667, 49.93333333333333, 49.9333
            #ref_idx = np.cumsum(np.ones(len(ref)) * 125 * 2) - 125 * 2
            #return pr_est[:len(ref)] * 60 - ref, pr_conf[:len(ref)]

        import glob

        import numpy as np
        import scipy as sp
        import scipy.io

        import matplotlib.pyplot as plt
        from scipy.signal import find_peaks
        import pandas as pd
        import random
        import math

        def LoadTroikaDataset():
            """
            Retrieve the .mat filenames for the troika dataset.
```

```
    Review the README in ./datasets/troika/ to understand the organization of the .mat f

    Returns:
        data_fls: Names of the .mat files that contain signal data
        ref_fls: Names of the .mat files that contain reference data
        <data_fls> and <ref_fls> are ordered correspondingly, so that ref_fls[5] is the
            reference data for data_fls[5], etc...
    """
    data_dir = "./datasets/troika/training_data"
    data_fls = sorted(glob.glob(data_dir + "/DATA_*.mat"))
    ref_fls = sorted(glob.glob(data_dir + "/REF_*.mat"))
    return data_fls, ref_fls

def LoadTroikaDataFile(data_fl):
    """
    Loads and extracts signals from a troika data file.

    Usage:
        data_fls, ref_fls = LoadTroikaDataset()
        ppg, accx, accy, accz = LoadTroikaDataFile(data_fls[0])

    Args:
        data_fl: (str) filepath to a troika .mat file.

    Returns:
        numpy arrays for ppg, accx, accy, accz signals.
    """
    data = sp.io.loadmat(data_fl)['sig']
    return data[2:]

def AggregateErrorMetric(pr_errors, confidence_est):
    """
    Computes an aggregate error metric based on confidence estimates.

    Computes the MAE at 90% availability.

    Args:
        pr_errors: a numpy array of errors between pulse rate estimates and correspondin
            reference heart rates.
        confidence_est: a numpy array of confidence estimates for each pulse rate
            error.

    Returns:
        the MAE at 90% availability
    """
    # Higher confidence means a better estimate. The best 90% of the estimates
    #    are above the 10th percentile confidence.
```

```
    percentile90_confidence = np.percentile(confidence_est, 10)

    # Find the errors of the best pulse rate estimates
    best_estimates = pr_errors[confidence_est >= percentile90_confidence]

    # Return the mean absolute error
    return np.mean(np.abs(best_estimates))

def Evaluate():
    """
    Top-level function evaluation function.

    Runs the pulse rate algorithm on the Troika dataset and returns an aggregate error m

    Returns:
        Pulse rate error on the Troika dataset. See AggregateErrorMetric.
    """
    # Retrieve dataset files
    data_fls, ref_fls = LoadTroikaDataset()
    errs, confs = [], []
    for data_fl, ref_fl in list(zip(data_fls, ref_fls))[:]:
        # Run the pulse rate algorithm on each trial in the dataset
        errors, confidence = RunPulseRateAlgorithm(data_fl, ref_fl)
        errs.append(errors)
        confs.append(confidence)
        # Compute aggregate error metric
    errs = np.hstack(errs)
    confs = np.hstack(confs)
    return AggregateErrorMetric(errs, confs)

def RunPulseRateAlgorithm(data_fl, ref_fl):
    '''Given a sample data file with PPG and 3 accelerometer channels and reference file
       compute pulse rates every two seconds.
       Parameters:
           data_fl: .mat file containing PPG and X, Y, Z accelerometer data from Troika
           ref_fl: .mat file containing ground truth heart rates from Troika dataset

       Returns:
           errors: numpy array with differences between predicted and reference heart ra
           confidence: numpy array with confidence values for heart rate predictions
    '''
    # Load data using LoadTroikaDataFile

    Fs = 125 # Troika data has sampling rate of 125 Hz

    ppg, accx, accy, accz = LoadTroikaDataFile(data_fl)

    winSize = 8*Fs # Ground truth BPM provided in 8 second windows
```

```
    winShift = 2*Fs # Successive ground truth windows overlap by 2 seconds

    ref = sp.io.loadmat(ref_fl)

    errs = []
    confs = []

    # For each 8 second window, compute a predicted BPM and confidence and compare to gr
    offset = 0
    for eval_window_idx in range(len(ref['BPM0'])):

        # Set verbose to True to visualize plot analysis
        verbose = False
        # verbose = True if eval_window_idx == 28 else False

        window_start = offset
        window_end = winSize+offset
        offset += winShift

        if verbose:
            print(f"Win start,end: {window_start}, {window_end}")

        ppg_window = ppg[window_start:window_end]
        accx_window = accx[window_start:window_end]
        accy_window = accy[window_start:window_end]
        accz_window = accz[window_start:window_end]

        pred, conf = AnalyzeWindow(ppg_window, accx_window, accy_window, accz_window, Fs

        groundTruthBPM = ref['BPM0'][eval_window_idx][0]
        if verbose:
            print('Ground Truth BPM: ', groundTruthBPM)

        predError = groundTruthBPM - pred
        errs.append(predError)
        confs.append(conf)

    errors, confidence = np.array(errs), np.array(confs)
    return errors, confidence

def AnalyzeWindow(ppg, accx, accy, accz, Fs=125, verbose=False):
    ''' Analyze a single 8 second window of PPG and Accelerometer data.
        Parameters:
            ppg: numpy array with ppg values
            accx/y/z: numpy arrays with per-axis accelerometer data
            Fs: sampling rate used by both PPG and accelerometer sensors
            verbose: display plots and logging information.
```

```
    Returns:
        prediction: Tuple of (BPM prediction, confidence) for this window.
'''

ppg_bandpass = BandpassFilter(ppg, fs=Fs)
accx_bandpass = BandpassFilter(accx, fs=Fs)
accy_bandpass = BandpassFilter(accy, fs=Fs)
accz_bandpass = BandpassFilter(accz, fs=Fs)

# Aggregate accelerometer data into single signal

accy_mean = accy-np.mean(accy_bandpass) # Center Y values
acc_mag_unfiltered = np.sqrt(accx_bandpass**2+accy_mean**2+accz_bandpass**2)
acc_mag = BandpassFilter(acc_mag_unfiltered, fs=Fs)

peaks = find_peaks(ppg_bandpass, height = 10, distance=35)[0]

if verbose:
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,4))
    ax1.title.set_text('Signal with Time Domain FindPeaks()')
    ax1.plot(ppg_bandpass)
    ax1.plot(peaks, ppg_bandpass[peaks], "x")

    ax2.title.set_text('Aggregated Accelerometer Data')
    ax2.plot(acc_mag, color="purple")
    plt.show()

# Use FFT length larger than the input signal size for higher spectral resolution.
fft_len=len(ppg_bandpass)*4

# Create an array of frequency bins
freqs = np.fft.rfftfreq(fft_len, 1 / Fs) # bins of width 0.12207031

# The frequencies between 40 BPM and 240 BPM Hz
low_freqs = (freqs >= (40/60)) & (freqs <= (240/60))

mag_freq_ppg, fft_ppg = FreqTransform(ppg_bandpass, freqs, low_freqs, fft_len)
mag_freq_acc, fft_acc = FreqTransform(acc_mag, freqs, low_freqs, fft_len)

peaks_ppg = find_peaks(mag_freq_ppg, height=30, distance=1)[0]
peaks_acc = find_peaks(mag_freq_acc, height=30, distance=1)[0]

# Sort peaks in order of peak magnitude
sorted_freq_peaks_ppg = sorted(peaks_ppg, key=lambda i:mag_freq_ppg[i], reverse=True
sorted_freq_peaks_acc = sorted(peaks_acc, key=lambda i:mag_freq_acc[i], reverse=True

# Use the frequency peak with the highest magnitude, unless the peak is also present
use_peak = sorted_freq_peaks_ppg[0]
```

```
    for i in range(len(sorted_freq_peaks_ppg)):
        # Check nearest two peaks also
        cond1 = sorted_freq_peaks_ppg[i] in sorted_freq_peaks_acc
        cond2 = sorted_freq_peaks_ppg[i]-1 in sorted_freq_peaks_acc
        cond3 = sorted_freq_peaks_ppg[i]+1 in sorted_freq_peaks_acc
        if cond1 or cond2 or cond3:
            continue
        else:
            use_peak = sorted_freq_peaks_ppg[i]
            break

    chosen_freq = freqs[low_freqs][use_peak]
    prediction = chosen_freq * 60
    confidence = CalcConfidence(chosen_freq, freqs, fft_ppg)

    if verbose:
        plt.title("PPG Frequency Magnitude")
        plt.plot(mag_freq_ppg)
        plt.plot(peaks_ppg, mag_freq_ppg[peaks_ppg], "x")
        plt.show()

        plt.title("ACC Frequency Magnitude")
        plt.plot(mag_freq_acc, color="purple")
        plt.plot(peaks_acc, mag_freq_acc[peaks_acc], "x")
        plt.show()

        print("PPG Freq Peaks: ", peaks_ppg)
        print("ACC Freq Peaks: ", peaks_acc)

        print("PPG Freq Peaks Sorted: ", sorted_freq_peaks_ppg)
        print("ACC Freq Peaks Sorted: ", sorted_freq_peaks_acc)
        print("Use peak: ", use_peak)
        print(f"Predicted BPM: {prediction}, {chosen_freq} (Hz), Confidence: {confidence

    return (prediction, confidence)

def BandpassFilter(signal, fs):
    '''Bandpass filter the signal between 40 and 240 BPM'''

    # Convert to Hz
    lo, hi = 40/60, 240/60

    b, a = sp.signal.butter(3, (lo, hi), btype='bandpass', fs=fs)
    return sp.signal.filtfilt(b, a, signal)

def FreqTransform(x, freqs, low_freqs, fft_len):
    '''Compute and return FFT and magnitude of FFT for given low frequencies
        Parameters:
```

```
            x: numpy array input signal to transform
            freqs: full list of FFT frequency bins
            low_freqs: low frequency bins between 40 BPM and 240 BPM
            fft_len: length of FFT to compute

        Returns:
            mag_freq_x: magnitude of lower frequencies of the FFT transformed signal
            fft_x: FFT of normalized input signal
    '''

    # Take an FFT of the normalized signal
    norm_x = (x - np.mean(x))/(max(x)-min(x))
    fft_x = np.fft.rfft(norm_x, fft_len)

    # Calculate magnitude of the lower frequencies
    mag_freq_x = np.abs(fft_x)[low_freqs]

    return mag_freq_x, fft_x

def CalcConfidence(chosen_freq, freqs, fft_ppg):
    '''Calculates a confidence value for a given frequency by computing
       the ratio of energy concentrated near that frequency compared to the full signal.
       Parameters:
            chosen_freq: frequency prediction for heart rate.
            freqs: full list of FFT frequency bins
            fft_ppg: FFT of normalized PPG signal

       Returns:
            conf_val: Confidence value for heart rate prediction.
    '''
    win = (40/60.0)
    win_freqs = (freqs >= chosen_freq - win) & (freqs <= chosen_freq + win)
    abs_fft_ppg = np.abs(fft_ppg)

    # Sum frequency spectrum near pulse rate estimate and divide by sum of entire spectr
    conf_val = np.sum(abs_fft_ppg[win_freqs])/np.sum(abs_fft_ppg)

    return conf_val
```

In [ ]: