Test Your Algorithm

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.

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.

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**.

Example

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.

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

In [1]:
```python
import glob

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

# set the sampling rate to 125Hz
fs = 125

# windowing parameters (win_length_s = 8s and win_shift_s = 2s)
win_length_s  = 8
win_shift_s = 2

win_length = win_length_s * fs
win_shift = win_shift_s * fs

# bpm boundary
lower_bpm, high_bpm = 40, 240

# frequency bins boundary
low, high = lower_bpm/60, high_bpm/60

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

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

    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 corresponding
            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 metric.

    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 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 ground_truth(ref_fl):
    """ loads the ground truth values
    args:
        ref_fl: Name of the .mat file that contains reference data
    Returns:
        A numpy array of the ground truth values for the input ref_fl
    """
    target = sp.io.loadmat(ref_fl)['BPM0']
    return target

def bandpass_filter(signal, fs, low, high):
    """
    filters the input signal between cut-off values.
    Args:
        signal: A numpy array representing the signal to filter
        fs: sampling rate
        low = required lower bound of frequency bin
        high = required upper bound of frequency bin
    Returns:
        A numpy array of the filtered signal
    """
    b, a = sp.signal.butter(3, (low, high), btype='bandpass', fs=fs)
    return sp.signal.filtfilt(b, a, signal)

def feature_generator(signal, fs):
    """
    uses the input signals to generate features within the specified windows
    Args:
        signal: bandpass filtered ppg signal
        fs: sampling rate
    Returns:
        tuple of features derived from the input signals

        The derived features are
         1. peaks
         2. frequencies at the peaks
         3. frequency of the signal, and
         4. the fourier transform of the signal
         5. mean of the signal
         6. energy of the signal
    """

    # fft of ppg signal
    fft_len = 2*len(signal)
    freq = np.fft.rfftfreq(fft_len, 1/fs)
    fft = np.abs(np.fft.rfft(signal, fft_len))

    # Filter fft between low_bpm and high_pbm
    fft[freq <= 70/60] = 0.0 # changed lower_bpm to 70 instead of 40 to bring the error down
    fft[freq >= 190/60] = 0.0 # changed lower_bpm to 190 instead of 240 for the same reason

    # ppg peaks and peak frequencies
    peaks = sp.signal.find_peaks(fft, height=2000)[0]
    peaks_freq = freq[peaks]

    # signal mean  and energy
    signal_filtered = bandpass_filter(signal, fs, low, high)
    mean = np.mean(signal_filtered)
    energy = np.sum(np.square(signal_filtered - mean))

    return freq, fft, peaks, peaks_freq, mean, energy


def RunPulseRateAlgorithm(data_fl, ref_fl):
    """
    Uses the data and reference files, computes pulse rate estimates and returns errors and confidence.

    Args:
        data_fl: Data file containing input ppg and accelerometer signals
        ref_fl: refernece file containing ground truth values
    Returns:
        Numpy arrays of errors and confidence computed at 8s window
    """

    # Load data using LoadTroikaDataFile
    ppg, accx, accy, accz = LoadTroikaDataFile(data_fl)

    # load the ground truth
    target = ground_truth(ref_fl)

    # filter signals
    ppg = bandpass_filter(ppg, fs, low, high)
    accx = bandpass_filter(accx, fs, low, high)
    accy = bandpass_filter(accy, fs, low, high)
    accz = bandpass_filter(accz, fs, low, high)

    acc = np.sqrt(accx**2 + accy**2 + accz**2)

    # compute pulse rates
    tol = 0.001
    est, confidence = [],[]

    for i in range(0, len(ppg) - win_length, win_shift):

        ppg_freq, ppg_fft, ppg_peaks, ppg_peaks_freq, mean_ppg, energy_ppg = feature_generator(ppg[i:i+
win_length], fs)
        acc_freq, acc_fft, acc_peaks, acc_peaks_freq, mean_acc, energy_acc = feature_generator(acc[i:i+
win_length], fs)

        max_ppg = ppg_freq[np.argmax(energy_ppg)]
        max_acc = acc_freq[np.argmax(energy_acc)]

        j = 1
        while np.abs(max_ppg-max_acc) <= tol and j <=2:
            j+=1
            max_ppg = ppg_freq[np.argsort(ppg_fft, axis=0)[-j]]
            max_acc = acc_freq[np.argsort(acc_fft, axis=0)[-j]]

        est_freq = max_ppg
        est.append(est_freq*60)

        # compute confidence
        win_freq = 30/60
        conf = np.sum(ppg_fft[(ppg_freq >= est_freq - win_freq) & (ppg_freq <= est_freq + win_freq)])/
np.sum(ppg_fft)
        confidence.append(conf)

    # Return per-estimate mean absolute error and confidence as a 2-tuple of numpy arrays.
    errors = np.abs(np.diag(est-target))
    return errors, np.array(confidence)
```