unit_test

September 14, 2020

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.

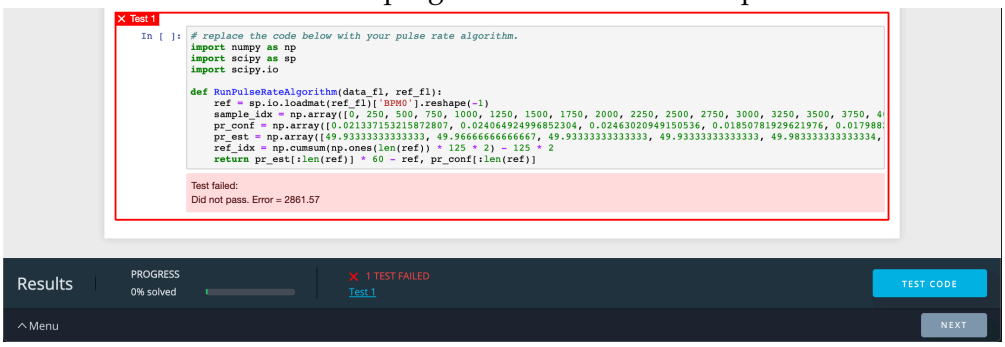
2. In the bottom right, click the Test Run button.

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.

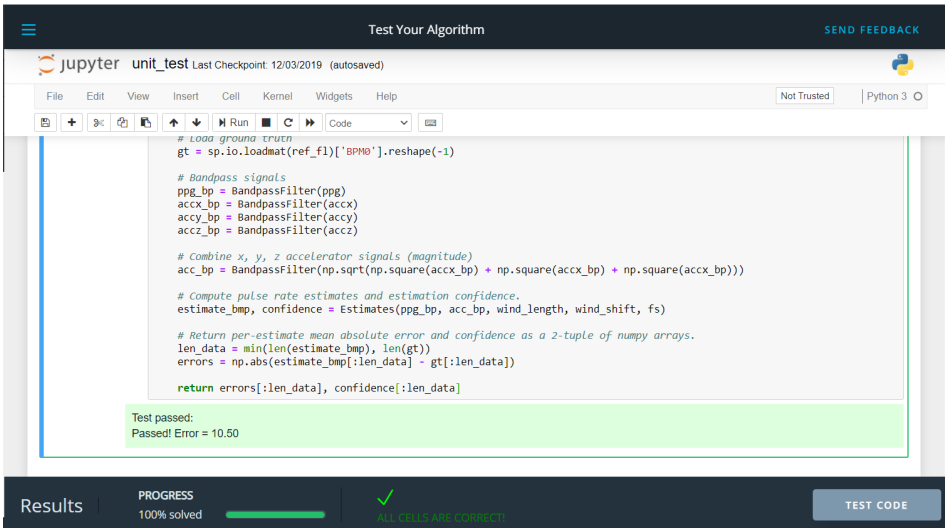
1.1.2 Pass

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 [ ]: # replace the code below with your pulse rate algorithm.
        import glob

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

        from matplotlib import pyplot as plt
        %matplotlib inline

        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 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 BandpassFilter(signal, lowcut=40./60, highcut=240./60, fs=125):
    """
    Loads the signal and passes it through a Butterworth bandpass filter.
    Args:
        signal: array_like, signal data to be filtered
        lowcat: float, low cut frequency in Hz
        highcut: float, high cut frequency in Hz
        fs: float, the sampling frequency of the digital system in Hz
    Returns:
        array_like, Band Pass filtered Signal
    """
    # Initiate bandpass filter
    b, a = sp.signal.butter(3, (lowcut, highcut), btype='bandpass', fs=fs)
    # Apply Butterworth bandpass filter and return filtered signal
    return sp.signal.filtfilt(b, a, signal)

def FourierTransform(signal, fs=125, zerofill=2):
    """
    Loads the signal and do a Discrete Fourier Transform on the signal
    Args:
        signal: array_like, signal data to apply Discrete Fourier Transform
        fs: float, the sampling frequency of the digital system in Hz
        zerofill: int, zero fill spectrum
    Returns:
        array_like, frequency and magnitude of the signal
    """
    # Compute discrete Fourier Transform sample frequencies
    fftlen = len(signal) * zerofill # for zero padding
    freqs = np.fft.rfftfreq(fftlen, 1/fs)
    # Compute the one-dimensional discrete Fourier Transform for real input
    fft = np.abs(np.fft.rfft(signal, fftlen))
    return freqs, fft
```

```
def spectrogram_show(signal, fs, signal_name, ylimits=(0.5, 5.5), estimates=None):
    '''
    Plot spectrogram with or without estimates
    Args:
        signal: array_like, signal data to apply Discrete Fourier Transform
        fs: float, the sampling frequency of the digital system in Hz
        signal_name: string, name of the signal to show on the image
        ylimits: tuple, (0.5, 5.5) is default
        estimates: None or array, the estimated frequencies
    Returns:
        Plot spectrogram
    '''
    plt.figure(figsize=(12, 8))
    spec, freqs, t, _ = plt.specgram(signal, NFFT=fs*4, Fs=fs, noverlap=0);
    plt.xlabel('Time (sec)')
    plt.ylabel('Frequency (Hz)')
    plt.ylim(ylimits)
    if not (estimates is None):
        plt.hlines(estimates, 0, len(signal)/fs, 'k')
        plt.title(f'{signal_name} Signal Spectrogram with estimates')
    else:
        plt.title(f'{signal_name} Signal Spectrogram')
    plt.show()

def signal_fft_show(signal, fs, xlimits=(0, 10), ylimits=None, peaks=None):
    '''
    Plot signal and FFT with or without estimates
    Args:
        signal: array_like, signal data to apply Discrete Fourier Transform
        fs: float, the sampling frequency of the digital system in Hz
        xlimits: tuple, (0, 10) is default
        ylimits: tuple, None is default
        peaks: None or array, the estimated frequencies
    Returns:
        Plot signal and it's FFT
    '''
    freqs, fft = FourierTransform(signal, fs=fs)
    # smooth FFT with SavitzkyGolay filter
    fft = scipy.signal.savgol_filter(fft, 5, 2)
    ts = np.arange(0, len(signal)/fs, 1/fs)
    plt.figure(figsize=(12, 8))
    plt.subplot(2,1,1)
    plt.plot(ts, signal)
    plt.title('Time-Domain')
    plt.xlabel('Time (sec)')
    plt.subplot(2,1,2)
    plt.plot(freqs, fft)
```

```
    if not (peaks is None):
        plt.plot(freqs[peaks], fft[peaks], 'r.', ms=10)
    plt.title('Frequency-Domain')
    plt.xlabel('Frequency (Hz)')
    plt.tight_layout()
    plt.xlim(xlimits)
    if ylimits:
        plt.ylim(ylimits)
    plt.show()

def plot_signal_fft_spectrogram_estimates(signal, fs, signal_name, threshold, distance,
    '''
    Plot signal, FFT, and spectrogram with simple estimates
    Args:
        signal: array_like, signal data to apply Discrete Fourier Transform
        fs: int, the sampling frequency of the digital system in Hz
        signal_name: string, name of the signal to show on the image
        threshold: float, the Y cut base on max Y value
        distance: integer, min distance between peaks
        freqs_limits: tuple, (0.5, 5.5) is default
    Returns:
        Plot signal, it's FFT, spectrogram with simple estimates
        print out peak estimates and confidence ratio
    '''
    freqs, fft = FourierTransform(signal, fs=fs)
    # smooth FFT with SavitzkyGolay filter
    fft = scipy.signal.savgol_filter(fft, 3, 2)
    pks = sp.signal.find_peaks(fft, height=np.max(fft)*threshold, distance=distance)[0]
    estimates = freqs[pks]
    signal_fft_show(signal, fs, xlimits=freqs_limits, peaks=pks)
    spectrogram_show(signal, fs, signal_name, ylimits=freqs_limits, estimates=estimates)
    print('Peaks esstimates Hz:', estimates)
    print('Peaks esstimates Per/Min:', estimates*60)
    confidence = []
    for indx in range(len(pks)):
        pk_l = freqs[int(pks[indx] - 10)]
        pk_m = freqs[int(pks[indx] + 10)]
        fft_pk = fft[(freqs >= pk_l) & (freqs <= pk_m)]
        freqs_pk = freqs[(freqs >= pk_l) & (freqs <= pk_m)]
        plt.plot(freqs_pk, fft_pk)
        plt.show()
        pk_confidence = np.sum(fft[(freqs >= pk_l) & (freqs <= pk_m)] / np.sum(fft))
        confidence.append(pk_confidence)
        print('Peak:', freqs[pks[indx]])
        print('Confidence (ratio area under peak / fft area): ', pk_confidence)

    return estimates, pks, confidence
```

```
def FindPeaks(signal, fs, threshold, distance, sg_filter=True):
    '''
    Find peaks in the spectrum and calculate confidence of the peaks defined as a ratio
    of sum of the frequency spectrum near the pulse rate estimate and the sum of the ent
    Args:
        signal: array_like, signal data to apply Discrete Fourier Transform
        fs: int, the sampling frequency of the digital system in Hz
        threshold: float, the Y cut base on max Y value
        distance: integer, min distance between peaks
        sg_filte: True or False, applying SavitzkyGolay filter to smooth spectrum
    Return:
        estimates: list, peaks estimates in Hz
        pks: list, peaks index in FFT spectrum
        confidence: list, confidence ratio
        freqs: array_like, frequencies
        fft: array_like, fft
    '''
    # Fourier transform
    freqs, fft = FourierTransform(signal, fs=fs, zerofill=2)

    # smooth FFT with SavitzkyGolay filter if set True
    if sg_filter:
        fft = scipy.signal.savgol_filter(fft, 5, 3)

    # get peaks
    pks = sp.signal.find_peaks(fft, height=np.max(fft)*threshold, distance=distance)[0]

    # compute peaks
    estimates = freqs[pks]

    # compute estimates
    confidence = []
    for indx in range(len(pks)):
        pk_l = freqs[int(pks[indx] - 10)]
        pk_m = freqs[int(pks[indx] + 10)]
        fft_pk = fft[(freqs >= pk_l) & (freqs <= pk_m)]
        freqs_pk = freqs[(freqs >= pk_l) & (freqs <= pk_m)]
        pk_confidence = np.sum(fft[(freqs >= pk_l) & (freqs <= pk_m)] / np.sum(fft))
        confidence.append(pk_confidence)

    return estimates, pks, confidence, freqs, fft

def Estimates(ppg_bp, acc_bp, wind_length, wind_shift, fs):
    '''
    Estimate heart rate in BMP
    Args:
        ppg_bp: array_like, bandpassed signal data from photoplethysmography sensor
        acc_bp: array_like, bandpassed magnitude signal data from accelerator sensor
```

```
    wind_length: int, time frame in seconds to collect signal for BMP estimation
    wind_shift: int, time frame in seconds to output the BMP estimate
    fs: int, the sampling frequency of the digital system in Hz
Return:
Hear rate estimate in BMP and confidence
'''
estimate_bmp, confidence = [], []

for indx in range(0, len(ppg_bp) - wind_length*fs, wind_shift*fs):
    ppg_wind = ppg_bp[indx:indx+wind_length*fs]
    acc_wind = acc_bp[indx:indx+wind_length*fs]

    # get potential estimates
    estimates_ppg, pks_ppg, confidence_ppg, freqs_ppg, fft_ppg = \
                                    FindPeaks(ppg_wind, fs, 0.3, 1, sg_filter=True)
    estimates_acc, pks_acc, confidence_acc, freqs_acc, fft_acc = \
                                    FindPeaks(acc_wind, fs, 0.3, 20, sg_filter=True)

    # create exclution indexes base on acc peaks
    exclude_acc = [list(range(tmp - 2, tmp + 3)) for tmp in pks_acc]
    exclude_acc_np = np.array(exclude_acc).flatten()

    # check that there is a ppg peak
    if len(estimates_ppg) == 0:
        estimate_bmp_tmp = freqs_ppg[np.argsort(fft_ppg, axis=0)[::-1][0]] * 60
        confidence_tmp = 0.0
    elif len(estimates_ppg) == 1:
        estimate_bmp_tmp = freqs_ppg[pks_ppg[0]] * 60
        confidence_tmp = confidence_ppg[0]
    else:
        # sort peaks and check if acc peak is not overlapping with the ppg peaks
        # the criteria is, the acc peak shouldn be within FWHM of the acc peaks
        estimate_ppg_indx = []
        pks_ppg_sorted = np.argsort(fft_ppg[pks_ppg], axis=0)[::-1]
        #pks_acc_sorted = np.argsort(fft_acc[pks_acc], axis=0)[::-1]
        for indx_pks in pks_ppg_sorted:
            #for fft_max_acc in pks_acc_sorted:
            if (pks_ppg[indx_pks] not in exclude_acc_np): #& (indx_pks not in estima
                estimate_ppg_indx.append(indx_pks)
        if len(estimate_ppg_indx) == 0:
            estimate_ppg_indx = pks_ppg_sorted
        estimate_bmp_tmp = freqs_ppg[pks_ppg[estimate_ppg_indx[0]]] * 60
        confidence_tmp = confidence_ppg[estimate_ppg_indx[0]]

    # in case of BMP jumps more than 20 BMP from previous 2 sec,
    # the BMP and confidence will be computed as anaverage of the last 3 calculation
    if len(estimate_bmp) > 1:
        if abs(estimate_bmp[-1] - estimate_bmp_tmp) >= 20:
```

```
                estimate_bmp_tmp = (estimate_bmp_tmp + estimate_bmp[-1] + estimate_bmp[-
                confidence_tmp = (confidence_tmp + confidence[-1] + confidence[-2])/3
            # add estimated BMP and Confidence to the list
            estimate_bmp.append(estimate_bmp_tmp)
            confidence.append(confidence_tmp)

    return np.array(estimate_bmp), np.array(confidence)

def RunPulseRateAlgorithm(data_fl, ref_fl):
    # Set sampling rate and window lenght and shift
    fs = 125
    wind_length = 8
    wind_shift = 2

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

    # Load ground truth
    gt = sp.io.loadmat(ref_fl)['BPM0'].reshape(-1)

    # Bandpass signals
    ppg_bp = BandpassFilter(ppg)
    accx_bp = BandpassFilter(accx)
    accy_bp = BandpassFilter(accy)
    accz_bp = BandpassFilter(accz)

    # Combine x, y, z accelerator signals (magnitude)
    acc_bp = BandpassFilter(np.sqrt(np.square(accx_bp) + np.square(accx_bp) + np.square(

    # Compute pulse rate estimates and estimation confidence.
    estimate_bmp, confidence = Estimates(ppg_bp, acc_bp, wind_length, wind_shift, fs)

    # Return per-estimate mean absolute error and confidence as a 2-tuple of numpy array
    len_data = min(len(estimate_bmp), len(gt))
    errors = np.abs(estimate_bmp[:len_data] - gt[:len_data])

    return errors[:len_data], confidence[:len_data]
```