unit_test

July 22, 2023

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.

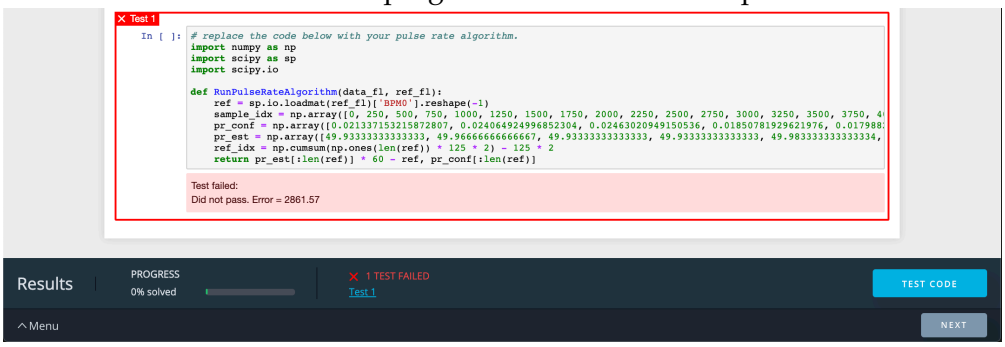
2. In the bottom right, click the Test Run button.

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.

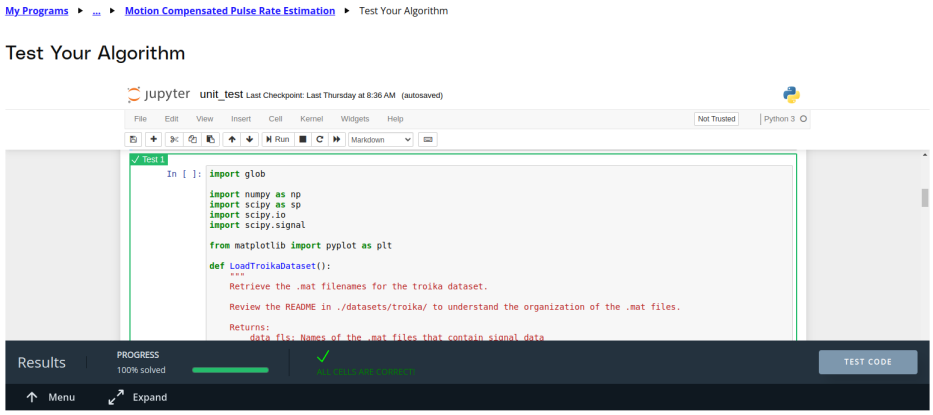
1.1.2 Pass

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 [ ]: import glob

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

        from matplotlib import pyplot as plt

        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 RunPulseRateAlgorithm(data_fl, ref_fl):
    """
    Run an algorith to estimate pulse rate with confidence for input datafile.

    Args:
        data_fl: (str) a path of a data file
        ref_fl: (str) a path of a reference data file

    Returns:
        errors: (number) errors between estimated pulse rate and reference pulse rate of
        confidence: (number) confidence value of the estimated pulse rate  of the file

    """
    # Set parameters
    fs = 125
    pass_band = (40/60, 240/60)
    cutoff_freq = 12
    verbose = False
    window_size = 8*fs # Ground truth BPM provided in 8 second windows
    window_shift = 2*fs # Successive ground truth windows overlap by 2 seconds

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

    # Load ground truth pulse rate
    # It was measurec each 8 second
    ground_truth = scipy.io.loadmat(ref_fl)['BPM0'].reshape(-1)

    # Compute pulse rate estimates and estimation confidence.
    ########################################
    # ------------ Pre Process ------------ #
    ########################################
    # Bandpass filter
    ppg_filtered = BandpassFilter(ppg, pass_band, fs)
    accx_filtered = BandpassFilter(accx, pass_band, fs)
    accy_filtered = BandpassFilter(accy, pass_band, fs)
    accz_filtered = BandpassFilter(accz, pass_band, fs)
```

```
    # The accelerometer channels are aggregated into a magnitude signal.
    acc_filtered = np.sqrt(accx_filtered ** 2 + accy_filtered ** 2 + accz_filtered ** 2)

    ##########################################
    # ------------ Main Process ------------ #
    ##########################################
    # Estimate pulse rate with confidence
    estimated_pulse_rates = []
    confidences = []
    estimated_pulse_rate_previous = 1

    for i in range(len(ground_truth)):
        estimated_pulse_rate, confidence = EstimatePulseRate(ppg_filtered[window_shift*i
                                                             acc_filtered[window_shift*i
                                                             fs, pass_band, estimated_pu
        estimated_pulse_rates.append(estimated_pulse_rate)
        confidences.append(confidence)

        estimated_pulse_rate_previous = estimated_pulse_rate

    ################################################
    # ------------ Evaluation Process ------------ #
    ################################################
    # errors = np.abs(np.subtract(estimated_pulse_rates, ground_truth))
    errors = np.abs(np.subtract(np.convolve(np.array(estimated_pulse_rates), np.ones(5)/
    mae = np.mean(errors)

    # Return per-estimate mean absolute error and confidence as a 2-tuple of numpy array
    errors, confidence = np.array(errors), np.array(confidences)
    return errors, confidence

def EstimatePulseRate(ppg, acc, fs, pass_band, estimated_pulse_rate_previous, verbose=Fa
    """
    Estimate Pulse rate with confidence using ppg and accelerometer data.

    Args:
        ppg: (np.array) a PPG signal
        acc: (np.array) an accelerometer signal
        fs: (number) a sampling rate of ppg/acc
        pass_band: (tuple) a bass band of band-pass filter for ppg/acc
        estimated_pulse_rate_previous: (number) previous estimated value of pulse rate
        verbose: (bool) Displays intermediate calculation results and diagrams

    Returns:
        estimated_pulse_rate: (number) estimated value of pulse rate
        confidence: (number) confidence value of the estimated pulse rate

    """
```

```
# Do FFT
freqs_ppg, fft_mag_ppg = DoFFT(ppg, fs)
freqs_acc, fft_mag_acc = DoFFT(acc, fs)

# Apply filter
fft_mag_ppg[freqs_ppg <= pass_band[0]] = 0.0
fft_mag_ppg[freqs_ppg >= pass_band[1]] = 0.0
fft_mag_acc[freqs_acc <= pass_band[0]] = 0.0
fft_mag_acc[freqs_acc >= pass_band[1]] = 0.0

# Calculate maximum frequency of ppg and acc
maximum_freq_ppg = freqs_ppg[np.argmax(fft_mag_ppg, axis=0)]
maximum_freq_acc = freqs_acc[np.argmax(fft_mag_acc, axis=0)]

# Exclude the peak frequency band of acc from the frequency band of ppg
fft_mag_ppg_new = fft_mag_ppg.copy()
fft_mag_th = np.percentile(fft_mag_acc[(freqs_acc >= (40/60)) & (freqs_acc <= (240/6
# fft_mag_ppg_new[fft_mag_acc>fft_mag_th] = 0.0
fft_mag_ppg_new[np.convolve(fft_mag_acc>fft_mag_th, np.ones(3), 'same').astype(np.bo

# Inverse FFT
ppg_new = DoInverseFFT(fft_mag_ppg_new)
pks_ppg_new = DetectPeaks(ppg_new, height=1, distance=None)

# Detect peaks of ppg
pks_ppg = DetectPeaks(ppg, height=1, distance=None)
pks_fft_mag_ppg = DetectPeaks(fft_mag_ppg, height=2000)
pks_fft_mag_ppg_new = DetectPeaks(fft_mag_ppg_new, height=2000)

# If the maximum frequency of ppg and the maximum frequency of acc are the same,
# calculate the pulse rate by excluding the maximum frequency of acc from ppg
if(maximum_freq_ppg == maximum_freq_acc):

    # Extract 5 maximum values, excluding the maximum value of acc
    est_range = np.argsort(fft_mag_ppg[fft_mag_ppg!=maximum_freq_acc], axis=0)[-5:]

    # Select the closest value from the previous estimate
    freq_pulse_rate = freqs_ppg[est_range[np.argmin(np.abs(freqs_ppg[est_range] - (e

    # Estimate pulse rate with confidence
    estimated_pulse_rate = freq_pulse_rate * 60
    confidence = CalculateConfidence(freq_pulse_rate, freqs_ppg, fft_mag_ppg)
else:

    # Extract 5 maximum values
    est_range = np.argsort(fft_mag_ppg, axis=0)[-5:]
```

```
            # Select the closest value from the previous estimate
            freq_pulse_rate = freqs_ppg[est_range[np.argmin(np.abs(freqs_ppg[est_range] - (e

            # Estimate pulse rate with confidence
            estimated_pulse_rate = freq_pulse_rate * 60
            confidence = CalculateConfidence(freq_pulse_rate, freqs_ppg, fft_mag_ppg)

        return estimated_pulse_rate, confidence

    def BandpassFilter(signal, pass_band, fs):
        """
        Applies bandpass filter.

        Args:
            signal: (np.array) an input signal
            pass_band: (tuple) a pass band. Frequency components outside
                the two elements in the tuple will be removed.
            fs: (number) a sampling rate of <signal>

        Returns:
            (np.array) The filtered signal
        """
        b, a = sp.signal.butter(3, pass_band, btype='bandpass', fs=fs)
        return sp.signal.filtfilt(b, a, signal)

    def LowpassFilter(signal, cutoff_freq, fs):
        """
        Applies lowpass filter.

        Args:
            signal: (np.array) an input signal
            cutoff_freq: (number) a frequency to want to cut off
            fs: (number) a sampling rate of <signal>

        Returns:
            (np.array) a filtered signal
        """
        b, a = sp.signal.butter(3, cutoff_freq, btype='lowpass', fs=fs)
        return sp.signal.filtfilt(b, a, signal)

    def DetectPeaks(signal, height=10, distance=35):
        """
        Detects peaks of the signal.

        Args:
            signal: (np.array) an input signal
            height: (number) a threshold of height of peak
            distance: (number) a threshold of distance between peaks
```

```
    Returns:
        signal: (np.array) am index of input signal's peaks

    """
    pks = sp.signal.find_peaks(signal, height, distance)[0]
    return pks

def DoFFT(signal_time_domain, fs):
    """
    Applies Fast Fourier Transform.

    Args:
        signal_time_domain: (np.array) an input time-domain signal
        fs: (number) a sampling rate of <signal>

    Returns:
        freqs: (np.array) a FFT frequency of the input signal
        fft_mag: (np.array) a FFT amplitude spectrum of the input signal

    """
    freqs = np.fft.rfftfreq(len(signal_time_domain), 1/fs)
    fft_mag = np.abs(np.fft.rfft(signal_time_domain))
    return freqs, fft_mag

def DoInverseFFT(signal_freq_domain):
    """
    Applies Inverse Fast Fourier Transform.

    Args:
        signal_freq_domain: (np.array) an input frequency-domain signal

    Returns:
        (np.array) a time-domain signal

    """

    return np.fft.irfft(signal_freq_domain)

def CalculateConfidence(freq_estimated_pulse_rate, freqs_ppg, fft_mag_ppg):
    """
    Calculates a confidence value for the chosen frequency by computing the ratio of ene

    Args:
        freq_estimated_pulse_rate: (number) a pulse rate estimated by the algorithm
        freqs_ppg: (np.array) a FFT frequency of ppg
        fft_mag_ppg: (np.array) a FFT amplitude spectrum of ppg
```

```
    Returns:
        confidence: (number) a confidence of pulse rate estimated by the algorithm

    """
    window = (freqs_ppg >= freq_estimated_pulse_rate - (40/60.0)) & (freqs_ppg <= freq_e
    confidence = np.sum(fft_mag_ppg[window])/np.sum(fft_mag_ppg)

    return confidence
```