unit_test

June 16, 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.

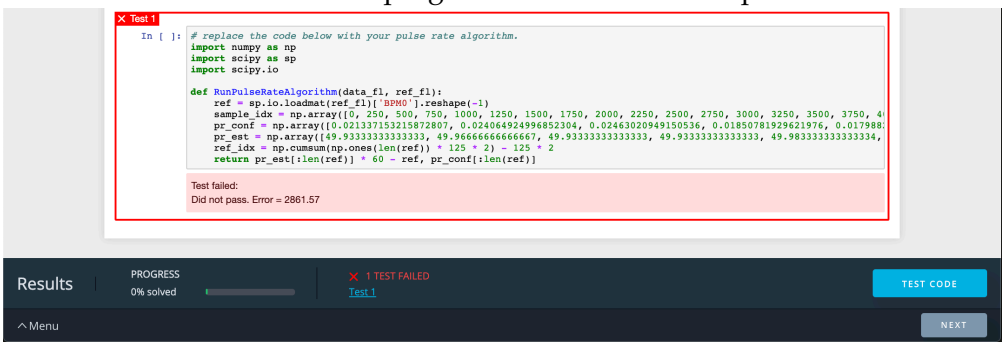
2. In the bottom right, click the Test Run button.

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.

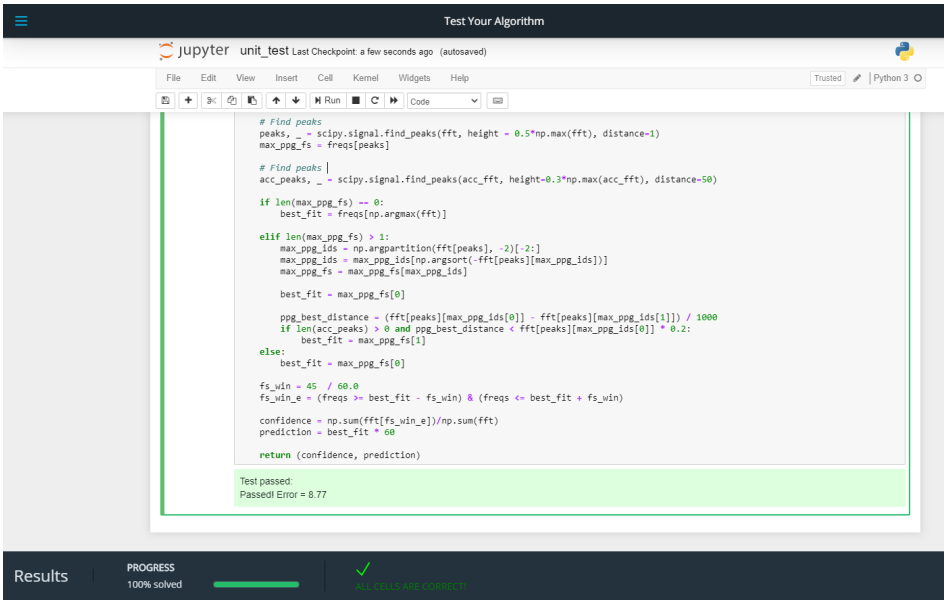
1.1.2 Pass

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 [ ]: import glob

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

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

    # load the reference signal
    ground_truth = scipy.io.loadmat(ref_fl)['BPM0'].reshape(-1)

    errors = []
    confidences = []

    start_indices, end_indices = get_start_end(len(accx), len(ground_truth))

    for i, start in enumerate(start_indices):
        end = end_indices[i]
        ref = ground_truth[i]

        # Bandpass filtering the signals
        w_ppg =  bandpass_filter(ppg[start:end])
        w_accx = bandpass_filter(accx[start:end])
        w_accy = bandpass_filter(accy[start:end])
        w_accz = bandpass_filter(accz[start:end])

        conf, pred = get_predictions(w_ppg, w_accx, w_accy, w_accz)

        errors.append(np.abs(pred - ref))
        confidences.append(conf)

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

fs = 125
```

```
window_shift = 2
window_length = 10

def get_start_end(sig_len, ref_len):
    """ Returns the start and end indices of a signal """
    n = ref_len if ref_len < sig_len else sig_len
    start = (np.cumsum(np.ones(n) * fs * window_shift) - fs * window_shift).astype(int)
    return (start, start + window_length * fs) # windows length of 10

def bandpass_filter(signal, band_pass = (40/60, 240/60), fs = fs):
    """ Performs a bandpass filter on the signal. """

    b,a = scipy.signal.butter(3, band_pass, fs=fs, btype= 'bandpass')

    # Perform forward and backward digital butterworth filter
    return scipy.signal.filtfilt(b,a,signal)


def filter_signal(signal, band_pass = (40/60, 240/60)):
    """ Performs bandpass filter and zeros outrange frequencies"""

def get_predictions(ppg, accx, accy, accz):
    """ Performs estimations on sensor signals

        Args:
            ppg : Photoplethysmography (PPG) sensor signal
            accx: Accelerometer sensor signal along x-axis
            accy: Accelerometer sensor signal along y-axis
            accz: Accelerometer sensor signal along z-axis

        Returns:
            The estimated frequency (in BPM) along with its confidence score.
    """

    n = len(ppg) * 4
    freqs = np.fft.rfftfreq(n, 1/fs)

    # Zeroing outrange frequencies
    fft = np.abs(np.fft.rfft(ppg, n))
    fft[freqs <= 40/60.0] = 0.0
    fft[freqs >= 240/60.0] = 0.0

    # Calculate the magnitude
    acc_abs = np.sqrt(accx ** 2 + accy ** 2 + accz ** 2)

    # Zeroing outrange frequencies
    acc_fft = np.abs(np.fft.rfft(acc_abs, n))
    acc_fft[freqs <= 40/60.0] = 0.0
```

```
acc_fft[freqs >= 240/60.0] = 0.0

# Find peaks
peaks, _ = scipy.signal.find_peaks(fft, height = 0.5*np.max(fft), distance=1)
max_ppg_fs = freqs[peaks]

# Find peaks
acc_peaks, _ = scipy.signal.find_peaks(acc_fft, height=0.3*np.max(acc_fft), distance

if len(max_ppg_fs) == 0:
    best_fit = freqs[np.argmax(fft)]

elif len(max_ppg_fs) > 1:
    max_ppg_ids = np.argpartition(fft[peaks], -2)[-2:]
    max_ppg_ids = max_ppg_ids[np.argsort(-fft[peaks][max_ppg_ids])]
    max_ppg_fs = max_ppg_fs[max_ppg_ids]

    best_fit = max_ppg_fs[0]

    ppg_best_distance = (fft[peaks][max_ppg_ids[0]] - fft[peaks][max_ppg_ids[1]]) /
    if len(acc_peaks) > 0 and ppg_best_distance < fft[peaks][max_ppg_ids[0]] * 0.2:
        best_fit = max_ppg_fs[1]
else:
    best_fit = max_ppg_fs[0]

fs_win = 45  / 60.0
fs_win_e = (freqs >= best_fit - fs_win) & (freqs <= best_fit + fs_win)

confidence = np.sum(fft[fs_win_e])/np.sum(fft)
prediction = best_fit * 60

return (confidence, prediction)
```