unit_test

June 11, 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.

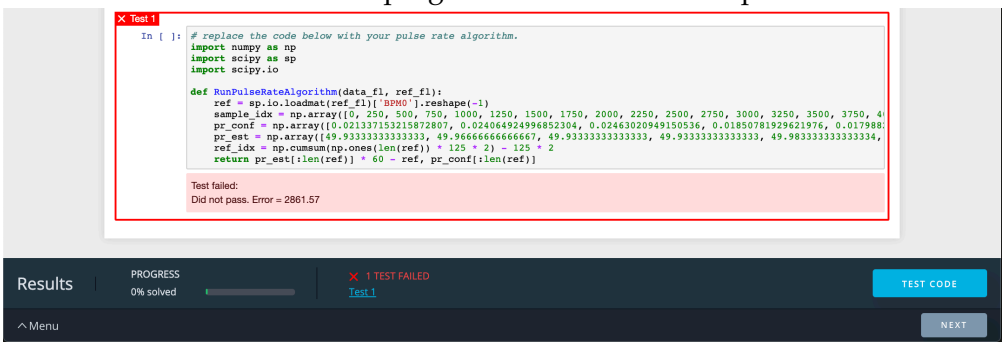
2. In the bottom right, click the Test Run button.

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.

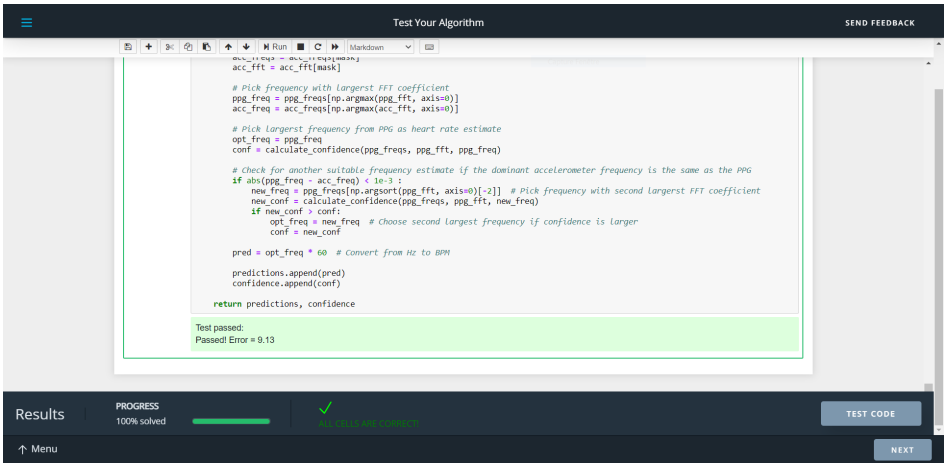
1.1.2 Pass

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 [ ]: import glob

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

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

            Usage:
                data_fls, ref_fls = load_troika_dataset()
                ppg, accx, accy, accz = load_troika_data_file(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 aggregate_error_metric(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 aggregate_error_metric.
    """
    # Retrieve dataset files
    data_fls, ref_fls = load_troika_dataset()
    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 aggregate_error_metric(errs, confs)

    def RunPulseRateAlgorithm(data_fl, ref_fl):  # Name does not follow PEP8 style guideline
        """
        Perform pulse rate estimation and calculate error and confidence estimate for each p

        Args:
            data_fl: file path to .mat file containing raw signals from PPG sensor and accel
            ref_fl: file path to .mat file containing reference heart rates

        Returns:
            errors: pulse rate errors (in BPM)
            confidence: confidence estimates
        """
        # Load data using load_troika_data_file
        ppg, accx, accy, accz = load_troika_data_file(data_fl)

        # Bandpass filter the signals
        ppg = bandpass_filter(ppg)
        accx = bandpass_filter(accx)
        accy = bandpass_filter(accy)
        accz = bandpass_filter(accz)

        # Compute pulse rate estimates and estimation confidence.

        # Consider only magnitude of acceleration
        acc = np.sqrt(accx**2 + accy**2 + accz**2)

        # Load labels and convert them from column vector to row vector
        labels = scipy.io.loadmat(ref_fl)['BPM0'].reshape(-1)

        # Calculate heart rate estimates and confidence
        predictions, confidence = predict(acc, ppg)

        # Calculate errors
        errors = np.abs(np.subtract(predictions, labels))

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

        return errors, confidence
```

```python
def bandpass_filter(s, fs=125):
    """
    Bandpass filter the signal between 40 and 240 BPM.

    Args:
        s: raw signal from sensor
        fs: sampling frequency in Hz

    Returns:
        Bandpass-filtered signal.
    """
    N = 3 # Order of the filter
    Wn = [40/60, 240/60] # Cutoff frequencies
    b, a = scipy.signal.butter(N, Wn, btype='bandpass', fs=fs)
    return scipy.signal.filtfilt(b, a, s)

def fft(signal, fs=125):
    """
    Calculate the Fast Fourier Transfrom of a sequence

    Args:
        signal: time-series
        fs: sampling frequency in Hz

    Returns:
        freqs: frequency bins
        fft: magnitude of FFT
    """
    n_samples = len(signal)  # No zero padding
    freqs = np.fft.rfftfreq(n_samples, 1/fs)
    fft = np.abs(np.fft.rfft(signal, n_samples))
    return freqs, fft

def calculate_confidence(freqs, fft, freq, window_half_width=1):
    """
    Calculate the confidence value for a frequency as a heart rate estimate

    The confidence quantifies how much energy in the frequency spectrum is concentrated

    Args:
        freqs: frequenqy bins used for FFT
        fft: magnitude of FFT
        freq: frequency corresponding to heart rate (in Hz)

    Returns:
        confidence: confidence estimate
    """
```

```
    # Window of 2 Hz centered on freq
    window = (freqs > freq - window_half_width) & (freqs < freq + window_half_width)
    confidence = np.sum(fft[window]) / np.sum(fft)
    return confidence

def predict(acc, ppg, fs = 125, window_length_s=8, window_shift_s=2):
    """
    Calculate heart rate from PPG and accelerometer data

    Args:
        acc: time series representing magnitude of acceleration
        ppg: time series representing PPG signal
        fs: sampling frequency in Hz
        window_length_s: window_length in seconds
        window_shift_s: window shift in seconds

    Returns:
        predictions: heart rate predicitons
        confidence: confidence estimates
    """
    # We go through the signals using overlapping windows
    window_length = window_length_s * fs  # 1000 samples
    window_shift = window_shift_s * fs  # 250 samples

    predictions = []
    confidence = []

    for start in range(0, len(ppg) - window_length + 1, window_shift):

        # Use only current window for estimation
        ppg_window = ppg[start:start+window_length]
        acc_window = acc[start:start+window_length]

        # Perform Fourier transform
        ppg_freqs, ppg_fft = fft(ppg_window)
        acc_freqs, acc_fft = fft(acc_window)

        # Remove unwanted frequencies
        mask = (ppg_freqs>40/60) & (ppg_freqs<240/60)
        ppg_freqs = ppg_freqs[mask]
        ppg_fft = ppg_fft[mask]
        mask = (acc_freqs>40/60) & (acc_freqs<240/60)
        acc_freqs = acc_freqs[mask]
        acc_fft = acc_fft[mask]

        # Pick frequency with largerst FFT coefficient
        ppg_freq = ppg_freqs[np.argmax(ppg_fft, axis=0)]
        acc_freq = acc_freqs[np.argmax(acc_fft, axis=0)]
```

```
        # Pick largerst frequency from PPG as heart rate estimate
        opt_freq = ppg_freq
        conf = calculate_confidence(ppg_freqs, ppg_fft, ppg_freq)

        # Check for another suitable frequency estimate if the dominant accelerometer fr
        if abs(ppg_freq - acc_freq) < 1e-3 :
            new_freq = ppg_freqs[np.argsort(ppg_fft, axis=0)[-2]]  # Pick frequency with
            new_conf = calculate_confidence(ppg_freqs, ppg_fft, new_freq)
            if new_conf > conf:
                opt_freq = new_freq  # Choose second largest frequency if confidence is
                conf = new_conf

        pred = opt_freq * 60  # Convert from Hz to BPM

        predictions.append(pred)
        confidence.append(conf)

    return predictions, confidence
```