unit_test

September 20, 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.

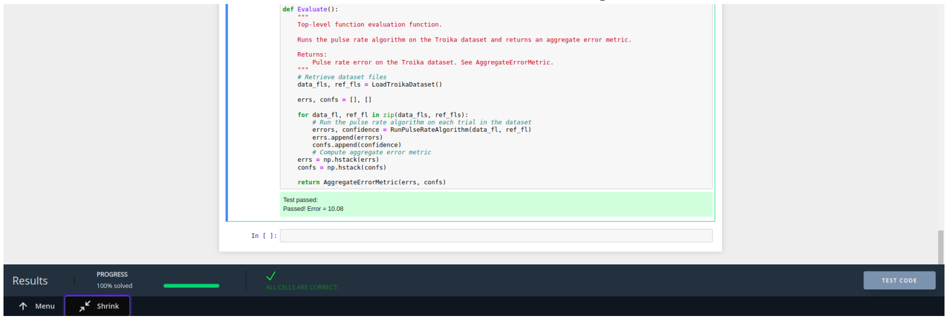
1.1.2 Pass

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 [8]:
```python
import glob
import joblib
import numpy as np
import scipy as sp
from tqdm import tqdm
from itertools import chain
from sklearn.model_selection import train_test_split, KFold
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error
import scipy.io
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.
```

```python
    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 = scipy.io.loadmat(data_fl)['sig']
    return data[2:]



def bandpass_filter(signal, pass_band, fs):
    """Bandpass Filter.

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

    Returns:
        (np.array) The filtered signal
    """
    # Design the bandpass filter using Butterworth filter design
    b, a = scipy.signal.butter(3, pass_band, btype='bandpass', fs=fs)

    # Apply the filter to the signal using filtfilt
    filtered_signal = scipy.signal.filtfilt(b, a, signal)

    return filtered_signal



def fast_fourier_transform(signal, fs):
    """
    Compute the Fast Fourier Transform of a signal.

    Args:
        signal (numpy.ndarray): The input signal.
        fs (float): The sampling rate of the signal.

    Returns:
        numpy.ndarray: FFT frequencies.
        numpy.ndarray: FFT of the signal.
```

```python
    """

    n = len(signal)
    fft_len = n * 4
    fft_freqs = np.fft.rfftfreq(fft_len, 1 / fs)
    fft_result = np.fft.rfft(signal, fft_len)
    return fft_freqs, fft_result



def get_dominant_frequency(signal, fs=125):
    """
    Calculate the dominant frequency of a signal.

    Args:
        signal (numpy.ndarray): The input signal.

    Returns:
        float: The dominant frequency of the signal.
    """
    low_freq = 40/60
    high_freq = 240/60

    # Apply bandpass filter
    filtered_signal = bandpass_filter(signal, pass_band=(low_freq, high_freq), fs=fs)

    # Compute FFT
    fftfreqs, fft = fast_fourier_transform(filtered_signal, fs=fs)

    # Zero out frequencies outside the range
    fftfreqs[(fftfreqs < low_freq) | (fftfreqs > high_freq)] = 0.0

    # Calculate magnitude spectrum
    mag_fft = np.abs(fft)

    # Find dominant frequency
    dom_freq = fftfreqs[np.argmax(mag_fft)]

    return dom_freq



def extract_features(window_data):
    """
    Extract features from a window of data.

    Args:
        window_data (numpy.ndarray): The data window containing PPG, accx, accy, and acc
```

```python
    Returns:
        numpy.ndarray: Dominant frequency features [dom_ppg_freq, dom_accx_freq, dom_acc
    """
    ppg, accx, accy, accz = window_data

    # Calculate dominant frequency for each signal
    dom_ppg_freq = get_dominant_frequency(ppg)
    dom_accx_freq = get_dominant_frequency(accx)
    dom_accy_freq = get_dominant_frequency(accy)
    dom_accz_freq = get_dominant_frequency(accz)

    # Return features as an array
    return np.array([dom_ppg_freq, dom_accx_freq, dom_accy_freq, dom_accz_freq])



def prepare_data_and_labels(data_fl, ref_fl):
    """
    Prepare sensor data and corresponding labels from data and reference files.

    Args:
        data_fl (str): Path to the sensor data file.
        ref_fl (str): Path to the reference file containing heart rate labels.

    Returns:
        signals (list): List of windowed sensor data.
        features (list): List of extracted features.
        labels (list): List of corresponding heart rate labels.
    """
    # Constants
    fs = 125
    win_length = 8*fs
    shift = 2*fs

    # Load data and refs
    data = LoadTroikaDataFile(data_fl)
    refs =  list(chain(*scipy.io.loadmat(ref_fl)['BPM0']))

    # Initialize lists to store data and labels
    signals, features, labels = [], [], []

    # Extract features
    for i, w_idx in enumerate(range(0, len(data[0]) - (win_length + shift), shift)):
            window_data = data[:, w_idx:w_idx+win_length]
            ppg_feature, accx_feature, accy_feature, accz_feature = extract_features(win

            features.append(np.array([ppg_feature, accx_feature, accy_feature, accz_feat
```

```python
            labels.append(refs[i])
            signals.append(window_data)

    return signals, features, labels



def aggregate_training_data():
    """
    Aggregate training data from multiple data and reference files.

    Returns:
        all_signals (numpy.ndarray): Array of windowed sensor data.
        all_features (numpy.ndarray): Array of extracted features.
        all_labels (numpy.ndarray): Array of corresponding heart rate labels.
    """
    # Initialize lists to store aggregated data
    all_signals, all_features, all_labels = [], [], []

    # Retrieve data and reference filenames
    data_fls, ref_fls = LoadTroikaDataset()

    # Process each data file and accumulate data and labels
    for data_fl, ref_fl in zip(data_fls, ref_fls):
        signals, features, labels = prepare_data_and_labels(data_fl, ref_fl)
        all_signals.extend(signals)
        all_features.extend(features)
        all_labels.extend(labels)

    return np.array(all_signals), np.array(all_features), np.array(all_labels)



def build_and_train_models():
    """
    Build and train machine learning models using cross-validation.

    This function trains Random Forest Regressor models on the provided data using
    5-fold cross-validation. Trained models are saved, and information about each model'
    performance (MAE) is stored in a .mat file.

    Returns:
        None
    """
    models_info_list = []  # List to store trained models for all folds

    # Load and prepare the data
    _, features, labels = aggregate_training_data()
```

```python
    # Initialize the model
    rf_regressor = RandomForestRegressor(n_estimators=300, max_depth=30)

    # Train the model using cross-validation
    for fold_idx, (train_idx, test_idx) in enumerate(KFold(n_splits=5).split(features, l
        X_train, y_train = features[train_idx], labels[train_idx]
        X_test, y_test = features[test_idx], labels[test_idx]

        # Fit the model on the training data
        rf_regressor.fit(X_train, y_train)

        # Evaluate the model
        mae = mean_absolute_error(y_test, rf_regressor.predict(X_test))
        print(f"Fold {fold_idx}: Mean Absolute Error = {mae}")

        # Save the trained model for the current fold
        model_filename = f"saved_models/rf_model_fold_{fold_idx}.joblib"
        joblib.dump(rf_regressor, model_filename)

        # Append the trained model information to the list
        models_info_list.append((model_filename, mae))

    # Save information about trained models to a .mat file
    sp.io.savemat('saved_models/trained_models_info.mat', mdict={'models_info': models_i
    print("Training complete.")



def load_best_model(models_info_path='saved_models/trained_models_info.mat'):
    """
    Load the best-trained machine learning model based on MAE.

    Args:
        models_info_path (str): Path to the .mat file containing model information.

    Returns:
        best_model (sklearn.ensemble.RandomForestRegressor): The best-trained model.
    """
    # Find the model with the lowest MAE
    trained_models = scipy.io.loadmat(models_info_path)['models_info']
    best_model_filename, _ = min(trained_models, key=lambda x: x[1])

    # Load and return the best model
    best_model = joblib.load(best_model_filename)
    return best_model
```

```python
def RunPulseRateAlgorithm(data_fl, ref_fl):
    """
    Estimate pulse rates from sensor data and calculate estimation confidence.

    Args:
        data_fl (str): Path to the sensor data file.
        ref_fl (str): Path to the reference file containing ground truth heart rate labe

    Returns:
        errors (numpy.ndarray): Array of mean absolute errors (MAE) between estimated an
        confidence (numpy.ndarray): Array of confidence scores (SNR) for each heart rate
    """

    # Constants
    fs = 125
    low_freq = 40/60
    high_freq = 240/60

    # Extract features and labels
    signals, features, labels = prepare_data_and_labels(data_fl, ref_fl)
    features, labels = np.asarray(features), np.asarray(labels)

    # Load the model
    model = load_best_model()

    # Initialize lists to store errors and confidence scores
    errors, confidence = [], []

    # Make heart rate estimations
    for feature, label, signal in zip(features, labels, signals):
        prediction = model.predict(np.reshape(feature, (1, -1)))[0]

        ppg, *_ = signal
        filtered_ppg = bandpass_filter(ppg, pass_band=(low_freq, high_freq), fs=fs)

        # Compute FFT
        fftfreqs, fft = fast_fourier_transform(filtered_ppg, fs=fs)

        # Magnitude spectrum
        mag_fft = np.abs(fft)

        # Zero out frequencies outside the range
        mag_fft[(fftfreqs < low_freq) | (fftfreqs > high_freq)] = 0.0

        # Convert the estimated heart rate to Hz
        hr_f = prediction / 60
```

```python
        # Compute the freq of the first harmonic
        harmonic_f = hr_f * 2

        # Convert window size to Hz
        window_f = 5 / 60

        # Get frequency windows
        fundamental_frequency_window = (fftfreqs > hr_f - window_f) & (fftfreqs < hr_f +
        harmonic_frequency_window = (fftfreqs > harmonic_f - window_f) & (fftfreqs < har

        # Compute signal power and noise power
        signal_power = np.sum(mag_fft[(fundamental_frequency_window) | (harmonic_frequen
        noise_power = np.sum(mag_fft[~((fundamental_frequency_window) | (harmonic_freque

        # Compute confidence (SNR)
        conf = signal_power / noise_power


        errors.append(np.abs(prediction - label))
        confidence.append(conf)

    # Return per-estimate mean absolute error and confidence as a 2-tuple of numpy array
    return errors, confidence



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)
```

In [ ]: