pulse_rate_starter

June 22, 2020

0.1 Part 1: Pulse Rate Algorithm

0.1.1 Contents

Fill out this notebook as part of your final project submission.

You will have to complete both the Code and Project Write-up sections. - The Section 0.1.3 is where you will write a **pulse rate algorithm** and already includes the starter code. - Imports - These are the imports needed for Part 1 of the final project. - glob - numpy - scipy - The Section 0.1.4 to describe why you wrote the algorithm for the specific case.

0.1.2 Dataset

You will be using the **Troika**[1] dataset to build your algorithm. Find the dataset under `datasets/troika/training_data`. The `README` in that folder will tell you how to interpret the data. The starter code contains a function to help load these files.

1. Zhilin Zhang, Zhouyue Pi, Benyuan Liu, "TROIKA: A General Framework for Heart Rate Monitoring Using Wrist-Type Photoplethysmographic Signals During Intensive Physical Exercise,"IEEE Trans. on Biomedical Engineering, vol. 62, no. 2, pp. 522-531, February 2015. Link

---

0.1.3 Code

```
In [1]: import glob

        import numpy as np
        import scipy as sp
        import scipy.io, scipy.signal
        from sklearn.ensemble import RandomForestRegressor
        from sklearn.model_selection import KFold


        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_estim0tes = 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 Iterator(data_len, ref_len, fs=125, win_len=6, win_shift=2):
    """
    Generate left index and right index for iteration
    """
    left = (np.cumsum(np.ones(min(data_len, ref_len)) * fs * win_shift) - fs * win_shift
    return left, left + fs * win_len

def BandpassFilter(signal, pass_band=(40/60,240/60), fs=125):
    b, a = sp.signal.butter(2, pass_band, btype='bandpass', fs=fs)
    return sp.signal.filtfilt(b, a, signal)

def Creator(ppg, accx, accy, accz, fs=125):
    """
    Create features for model fitting
    """
    n = len(ppg) * 4
    freqs = np.fft.rfftfreq(n, 1/fs)
    fft = np.abs(np.fft.rfft(ppg, n))
    fft[freqs <= 40/60] = 0.0
    fft[freqs >= 240/60] = 0.0
```

```
    mag_acc = np.sqrt(accx**2 + accy**2 + accz**2)

    # FFT for acc
    acc_fft = np.abs(np.fft.rfft(mag_acc, n))
    acc_fft[freqs <= 40/60] = 0.0
    acc_fft[freqs >= 240/60] = 0.0

    # max frequency for ppg
    ppg_feature = freqs[np.argmax(fft)]
    # max frequency for acc
    acc_feature = freqs[np.argmax(acc_fft)]
    return np.array([ppg_feature, acc_feature])

def BuildModel():
    # Retrieve filenames through LoadTroikaDataset
    data_fls, ref_fls = LoadTroikaDataset()
    features, labels, signals = [], [], []

    for data_fl, ref_fl in zip(data_fls, ref_fls):
        signal = LoadTroikaDataFile(data_fl)
        ref = np.array([_[0] for _ in scipy.io.loadmat(ref_fl)['BPMO']])
        ls, rs = Iterator(signal.shape[1], len(ref))
        for i in range(len(ls)):
            left, right = ls[i], rs[i]
            ppg = BandpassFilter(signal[0, left:right])
            accx = BandpassFilter(signal[1, left:right])
            accy = BandpassFilter(signal[2, left:right])
            accz = BandpassFilter(signal[3, left:right])

            features.append(Creator(ppg, accx, accy, accz))
            labels.append(ref[i])
            signals.append([ppg, accx, accy, accz])

    features, labels = np.array(features), np.array(labels)
    model = RandomForestRegressor(n_estimators=300, max_depth=16)
    for train_idx, test_idx in KFold(n_splits=5).split(features, labels):
        X_train, y_train = features[train_idx], labels[train_idx]
        X_test, y_test = features[test_idx], labels[test_idx]
        model.fit(X_train, y_train)

    return model

def RunPulseRateAlgorithm(data_fl, ref_fl):
    # Load data using LoadTroikaDataFile
    signal = LoadTroikaDataFile(data_fl)
    features, labels, signals = [], [], []
    ref = np.array([_[0] for _ in scipy.io.loadmat(ref_fl)['BPMO']])
```

```
ls, rs = Iterator(signal.shape[1], len(ref))
for i in range(len(ls)):
    left, right = ls[i], rs[i]
    ppg = BandpassFilter(signal[0, left:right])
    accx = BandpassFilter(signal[1, left:right])
    accy = BandpassFilter(signal[2, left:right])
    accz = BandpassFilter(signal[3, left:right])

    features.append(Creator(ppg, accx, accy, accz))
    labels.append(ref[i])
    signals.append([ppg, accx, accy, accz])

features, labels = np.array(features), np.array(labels)
model = BuildModel()
# Compute pulse rate estimates and estimation confidence.
errors, confidence = [], []

for i in range(len(signals)):
    feature, label = features[i], labels[i]
    ppg, accx, accy, accz = signals[i]
    pred = model.predict(np.reshape(feature, (1, -1)))[0]
    ppg = BandpassFilter(ppg)
    accx = BandpassFilter(accx)
    accy = BandpassFilter(accy)
    accz = BandpassFilter(accz)

    n = len(ppg) * 3
    freqs = np.fft.rfftfreq(n, 1/125)
    fft = np.abs(np.fft.rfft(ppg, n))
    fft[freqs <= 40/60] = 0.0
    fft[freqs >= 240/60] = 0.0

    # max frequency
    pred_fs = pred / 55
    fs_win = 30 / 60
    fs_win = (freqs >= pred_fs - fs_win) & (freqs <= pred_fs + fs_win)
    confid = np.sum(fft[fs_win]) / np.sum(fft)

    errors.append(np.abs(pred - label))
    confidence.append(confid)

# Return per-estimate mean absolute error and confidence as a 2-tuple of numpy array
# errors, confidence = np.ones(100), np.ones(100)  # Dummy placeholders. Remove
return errors, confidence
```

0.1.4 Project Write-up

Answer the following prompts to demonstrate understanding of the algorithm you wrote for this specific context.

- **Code Description** - Include details so someone unfamiliar with your project will know how to run your code and use your algorithm.
- **Data Description** - Describe the dataset that was used to train and test the algorithm. Include its short-comings and what data would be required to build a more complete dataset.
- **Algorithm Description** will include the following:
- how the algorithm works
- the specific aspects of the physiology that it takes advantage of
- a describtion of the algorithm outputs
- caveats on algorithm outputs
- common failure modes
- **Algorithm Performance** - Detail how performance was computed (eg. using cross-validation or train-test split) and what metrics were optimized for. Include error metrics that would be relevant to users of your algorithm. Caveat your performance numbers by acknowledging how generalizable they may or may not be on different datasets.

Your write-up goes here...

- **Code Description**
- To run the code, RunPulseRateAlgorithm will take in two filenames and return a tuple of two numpy arrays--per-estimate pulse rate error and confidence values. The Evaluate function can be also called on the Troika dataset to compute an aggregate error metric.
- **Data Description**
- The Troika dataset is used to train and test the algorithm, including PPG signal from two wrists, but just one is applied, and IMU signal from three dimensions. ECG signal and both PPG signals could be required to improve performance.
- **Algorithm Description**
- A regression algorithm RandomForestRegressor is trained to fit the provided data.
- It takes advantage of the ability for PPG sensor to detect the amount of red blood cells under the wrists.
- The algorithm outputs are two numpy arrays: per-estimate pulse rate error and confidence values
- Per-estimate pulse rate error is the absolute difference between the predicted value and the ground truth, and confidence value is the relative difference.
- Common failure modes include hand motion and gesture movement.
- **Algorithm Performance**
- Performance is computed based on K-fold cross-validation, and error is the absolute bias between the predicted value and the ground truth. Test result is Error = 10.98, but due to the small sample size, it might not be generalizable on other datasets.

0.1.5 Next Steps

You will now go to **Test Your Algorithm** to apply a unit test to confirm that your algorithm met the success criteria.