A Guide for the Project

By Po-Teng Tseng

Introduction

All files can be found on Tracs under: Resources / IOT2016 /
This guide can be found under: Resources / IOT2016 / Fall-Detection / Boden's work /

So far, we have two **Android apps**:

1. drunk-smartwatch-master (/ drunk-smartwatch-master-c364948349de24202db2ff3bd53e10f9592b0233.zip)
   The original app for collecting data.
2. Fall detection app (/ Fall-Detection / Fall detection app 03-03-2017.zip)
   The modified app with model in it for prediction.

One **R script**:

1. BACtoCSV.r (/ Fall-Detection / Boden's work / BACtoCSV.r)
   The script helps you to convert the raw data from drunk-smartwatch-master to an appropriate format. Or you can also convert the data by yourself (Excel, Google spreadsheet… etc).

One **Java program**:

1. testmodel.java (/ Fall-Detection / fall detection model creation.zip)
   The program for generating SVM model. Put the generated model file into Fall detection app to change the model you want to use with it.

Basic workflow:
Connect the watch to the phone
→ Run App on the phone (drunk-smartwatch-master)
→ Collecting raw data
→ Get the raw data file from the phone
→ Convert the data to appropriate format (BACtoCSV.r or anything you like)
→ Run the program to generate models (testmodel.java)
→ Put the model into the app
→ Run the prediction and see the results (Fall detection app)

How to run the app and things to prepare.

Hardware needed:

1. An Android device with decent Android version.
2. USB to microUSB wire (or Samsung charger).
3. A Microsoft Band

Software needed:

1. Download and install Android Studio (it would be the main platform we use)
   You can look at the official website to get the most precise tutorial from Google.
2. Install Microsoft Band app on your phone. Pair you phone and watch.
   The manual of the watch will cover this part if you don't know how to do it.
3. Download the app from Tracs and extract it.

Launch Android Studio and click "Open existing project", and choose the app you just extracted. If there is any error and it asks you to update something, just update all of them.

Click the "**Run**" button. After you click "**Run**", it will show you all the devices connecting to your pc/tablet and all virtual machines you've built by Android Studio.

If you still don't see your device, check if you installed the appropriate "driver" on your pc/laptop. The driver shouldn't be just a USB driver or something which lets you takes stuff from your phone to computer, so if you already can do this, it doesn't mean you already had a driver installed.
All brands have different drivers and you can find them by yourself.
You can also find a webpage with links all all brands' drivers on Android Studio official website.

To run the app on your phone (recommended), connect your device to your pc/tablet via your cable, and click the "**Run**" button. (If you're using a phone with Android system, you have to turn on the **USB debug mode**.)

If you're having issue when the app says your watch is not pairing with your phone, try get "microsoft-band-1.3.20307.2.jar" file from Tracs and replace it with the current microsoft-band jar file.

Document location:
/0/Documents/SmartWatchSamples/data/
or
/storage/emulated/0/Documents/SmartWatchSamples/data/

Create a new model

Be careful when you're creating a new Weka model for the app. Reading a new model might give you an error if the model was created in a different environment from where the app is built. The different SDK versions will prevent models from being read.
A good way to do is creating a new Java class in your Android Studio project at where the app is, and put the program of creating models into that class so that both of them share the same environment.

Parameter Changes

SensorService.java

Time interval for each data point:

```
private final Collector collector = new Collector(4f);
```

4f means there are 4 data being collected in 1 second, so it's 250ms for each data point.
Ex: Make it 8f would reduce the time interval to 125ms.

Add new listener:

```
/** This thread creates and attempts to maintain connection with MS Band smartwatch. */
private class SensorCollectorThread extends Thread {
  boolean collecting = true;

  @Override
  public void run() {
      // Notify that collection is starting.
      for (ConnectionStatusListener l : connectionStatusListeners) l.onCollectionStarting();

      // Begin the main loop.
      while (!Thread.interrupted() && this.collecting) {
          // Try to connect to smartwatch until connected.
          while (!linker.connect()) {
              try { Thread.sleep(1000);
              } catch (InterruptedException e) { /** Nothing to do here. */ }
          }

          // Add the sensors to the data collector.
          collector.addSensor(androidAccelerometerSensor);
          collector.addSensors(linker);
          collector.addSensor(locationSensor);
          collector.addSensor(isDrinkingSensor);
```

The data will be presented as the order you set at here.

CSVSampleWriter.java

The number of data being processed for each prediction:

```
SampleAccumulator.StorageConfig storageConfig = new
SampleAccumulator.StorageConfig(
        SampleAccumulator.StorageConfig.TriggerType.NUM_SAMPLES, 25
);
```

The default number is 25.
It means if the time interval for each data point is 250ms, it takes 25 * 250ms = 6250ms = 6.25s to get a prediction.

Sliding window definition:

```
ax[0] = Double.parseDouble(samples[count-2][0]);      //first ax = newest sample; column 0(which holds x acceleration)
ax[1] = Double.parseDouble(samples[count-1][0]);    //second ax = previous sample; column 0
ax[2] = Double.parseDouble(samples[count][0]);     //...
ay[0] = Double.parseDouble(samples[count-2][1]);      //first ay = newest sample; column 1(which holds y acceleration)
ay[1] = Double.parseDouble(samples[count-1][1]);     //....
ay[2] = Double.parseDouble(samples[count][1]);     //...
az[0] = Double.parseDouble(samples[count-2][2]);      //first az = newest sample; column 2(which holds z acceleration)
az[1] = Double.parseDouble(samples[count-1][2]);     //...
az[2] = Double.parseDouble(samples[count][2]);     //...
```

The default definition compares the next two data with the current data to get Amin and Amax for current data point.
If you change this part, change also the if condition into (count > 4) before this block should prevent some crushes.

MSBandLinker.java

Sample rates for sensors:

```
MSBandSensor sensor = new AccelerometerSensor(this.client, SampleRate.MS128);
```

MS128 should be 128 ms

(The document mignt be continuing in the future.)