University of Pisa
MSc in Computer Engineering
Mobile and Social Sensing Systems

COVID-A APP: Hand Activity Recognition Project

Daniela Comola
Leonardo Fontanelli
Eugenia Petrangeli

Academic Year 2019/2020

Contents

1 Introduction

The implemented application for Android Smartphone, is an app aimed at fighting *COVID-19*. Hence, the main goal of the app is to help reducing the transmission of the virus, by monitoring the user in different areas:

- **Tracking the user's movements**. By knowing his location, the app will be able to understand if the user came in contact with an infected person.
- **Tracking the user's activity**. The behavior of the user determine a risk index, witch is updated positively or negatively considering the actions performed by the user.

This is achieved with several modules in the app, every module is implemented by a different group.
The module implemented by our group regards the *"Hand Washing Activity Recognition"*.
Since the **World Health Organization (WHO)** stated that[1] *"washing hands is the most effective remedy against Coronavirus. Soap is the most powerful weapon against the virus"*, the detection of the washing hand activity becomes one of the most important part of a Coronavirus detection App.
The general functioning of the module is the following:
The module starts sensing the activity, with a smartwatch companion app, every time the user come back home. At first it turns on the sensor at low frequency and starts sensing; when the app detects that the user could have started an hand activity, it increases the frequency in order to classify correctly the activity he's performing. If the app detects that the user has washed his hands, it updates positively the risk index. Otherwise, if the app doesn't detect an hand washing activity in a period of time, the risk index is updated negatively.
A recent study[2] showed that the best part of the body in order to detect and correctly classify the hand activities, is the wrist. Consequently, a smartwatch is the best way in order to classify the aforesaid activities. Hence, the application has a *WearOS* companion app for the smartwatch, that sends the sensors data to the smartphone where a classifier model is installed, for classification.

[1] `https://www.repubblica.it/salute/medicina-e-ricerca/2020/05/04/news/coronavirus_oms_igiene_mani_e_salvavita_il_world_day_handhygiene-255647291/`

[2] `http://www.gierad.com/projects/handactivities/`

2 Classifier

As previously said, in order to correctly detect the washing hand activities, the app needs a classifier.
The model of the classifier has been trained and exploited with *Weka*[3], which is an open source machine learning software that can be accessed through a graphical user interface, standard terminal applications, or a Java API. Some several steps has to be performed in order to obtain the final model for classifying.

2.1 Collecting Data

The first step regards collecting data in order to build a dataset for training the classifier.
As already said, the best place in order to detect hand activities is the wrist. Hence, the data have been collected using a smartwatch. In particular we've used a *Tickwatch E2 with WearOS*[4]

Figure 1: The Tickwatch smartwatch

Some watch's sensors have been exploited, in order to collect data. In particular we've used:

- **Accelerometer:** is an hardware-based sensor, that retrieves the acceleration force data for the three coordinate axes.
- **Gyroscope:** is an hardware-based sensor that returns rate of rotation data for the three coordinate axes.
- **Linear Acceleration:** is a software-based sensor that provides three-dimensional values representing acceleration along each device axis.
- **Gravity:** is a software-based sensor, that provides a three dimensional vector indicating the direction and magnitude of gravity.
- **Rotation Vector:** is a software-based sensor, represents the orientation of the device as a combination of an angle and an axis, in which the device has rotated through an angle θ around an axis (x, y, or z).
  The values returned by the sensor are relative to *Azimuth, Pitch* and *Roll.* The *Azimuth*, related to our application, would return the orientation of

[3]`https://www.cs.waikato.ac.nz/~ml/weka/`
[4]`https://www.amazon.it/dp/B07M9QV4N2/ref=cm_sw_em_r_mt_dp_U_UeN1EbPRKMGCM`

the user when he's in front of the sink. Given that we're interested in recognizing the orientation with every sink, the *Azimuth* is misleading for our purpose. Hence, we only keep the values from *Pitch* and *Roll*.

For gathering the initial dataset, a preliminary app on smartphone, with a companion smartwatch app, have been developed in order to collect data from the sensors. The smartwatch app has 2 buttons in the *MainActivity* interface, that are responsible to "start" and "stop" the experiments.

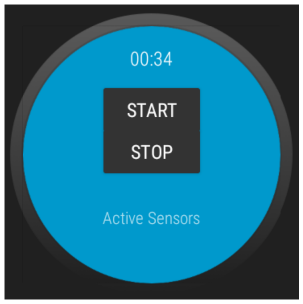
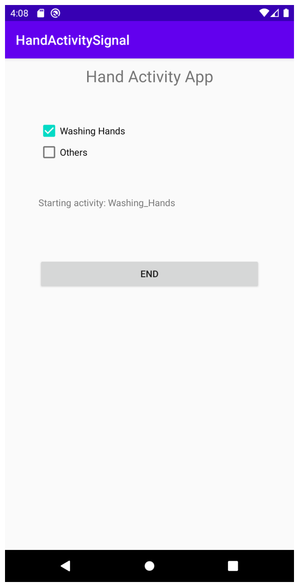


Figure 2: The preliminary App's UI on Smartwatch



Starting activity: Washing_Hands

Figure 3: The preliminary App's UI on Smartphone

The sensing has been performed around *50Hz* that correspond to *"SENSOR_DELAY_GAME"* in the Android Sensor API. When an experiment ends, 5 csv files, corresponding at each sensor, are saved and they are sent to the

smartphone via *DataLayer API's Assets*.
The experiments are differentiated and classified with 2 activities: since the app is interested in retrieving when the user has washed his hands and nothing else, the first class is *"Others"* and the second one is *"Washing_Hands"*.
Several tests have been performed by different people, with the smartwatch worn, regarding both classes (performing different activities for the "Others" class) in order to have a large dataset and to avoid false positive classification.

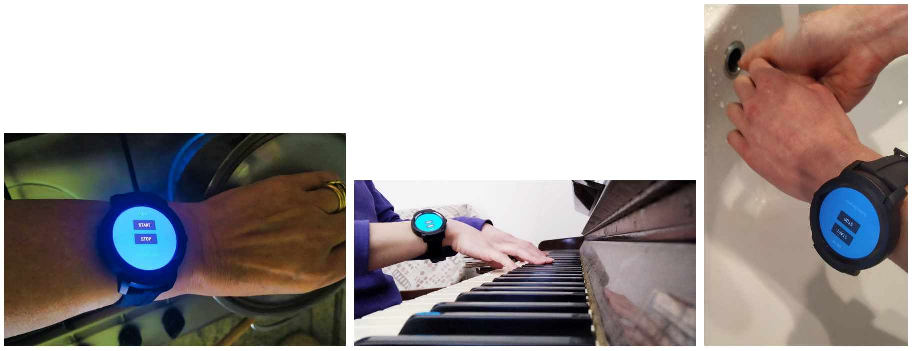


Figure 4: Some activity performed in order to gather data for the dataset

Wake-up and Non-Wake-up sensors Sensors are often defined by wake-up and non-wake-up pairs, both sensors sharing the same type, but differing by their wake-up characteristic. Non-wake-up sensors are sensors that do not prevent the SoC from going into suspend mode and do not wake the SoC up to report data. In opposition to non-wake-up sensors, wake-up sensors ensure that their data is delivered independently of the state of the SoC. The power consumption of the device with the non-wake-up mode is usually 100 times less than in the "On" mode.
In our code, by using the *PowerManager* class and the *wakeLock* object, with the *acquire()* and *release()* methods we're able to force the processor to be awake for the period of the sensing. Hence, with the idea of preserving battery, we prepared 2 kinds of experiments: one by using non-wake-up sensor and one with wake-up sensors in order to see if there is a compromise between power consumption and accuracy of the classifier.

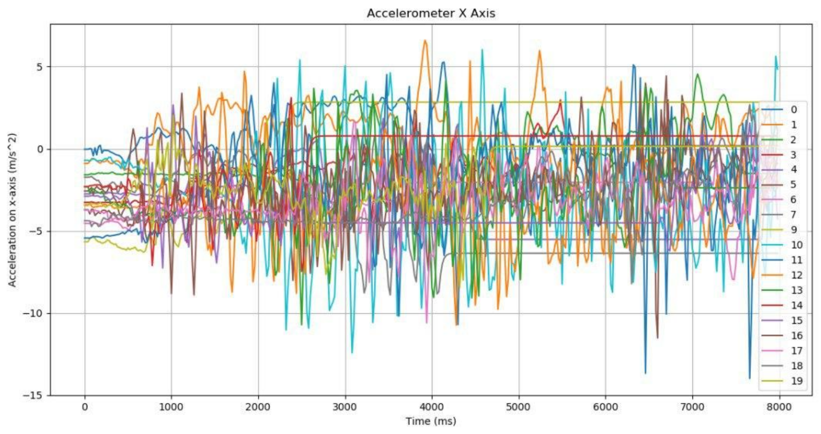
2.2 Feature Extraction

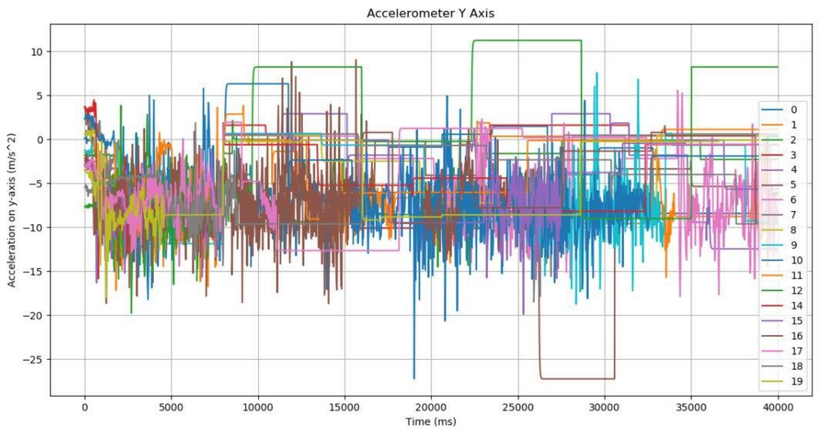
The preliminary developed app on the smartphone, after retrieving the files, has to prepare the *".arff"* file, that is the format file for the Weka software that will contain the features selected (in the form of *"@attribute"* list) and the relative values (after the *"@data"* keyword).
Regarding the features to be inserted in the file, we decided to use 4 kind of metrics: **Mean, Standard Deviation, Kurtosis and Skewness.**
The collected signals are of 40 seconds, divided into fragments of 8 seconds,

and also the signal that will be presented to the classifier in the final app is of 8 seconds as well. The first 10 samples from the washing hands traces are removed given that, plotting the signals (as shown below), we saw that the first 0,2 seconds are not related to the washing hands activity but to the few instants before starting the experiments, in which hands were at rest.
We decided to give in input 8 seconds of the signal in order to allow the classifier to recognize a more complex activity composed by different actions, all at once. If we had chosen to input a signal of shorter duration we would have trained the classifier to recognize the individual actions that make up hand washing. These individual actions, however, could also be significant for different activities. So we decided to give importance to the sequence of actions that make up a hand washed.
In order to increase the precision of the classifier, we decided to compute contiguous and fixed windows on the signal. Washing hands is a complex activity composed by lots of different atomic actions. To be more precise, we thought that is better to compute the features in small windows in order to differentiate all those atomic actions. Moreover, the app computes 2 kinds of *".arff"* files with 2 different windows sizes (performed both for the non-wake-up and wake-up experiments).
The first file has 2 seconds windows, and the second file has 4 seconds windows. This gave the possibility to test which file allows to obtain the best performance in terms of accuracy for the training of the classifier.
In this phase, by analysing the signal of the non-wake-up sensors, we realized that there were present some missing values: some part of the signals were not captured by the sensors. Hence, we've decided to replace the missing parts in order to have a coherent sampling of the signals, in particular to have a temporal correspondence of the windows for every signal.
The replacement has been performed by inspecting the signal step by step following the timestamps and checking if there is a missing value in the range of the following timestamp (summing the actual timestamp + $0.03ms$). If a missing value is detected, it is replaced with the mean of the following sensed sample and the previous. Unfortunately, with non-wake-up sensors can happen that the SoC can not be awakened for a long time, so the unreported sensors data, during an experiment, could be a lot. When this happens, also between different subsequent fragments there could be many missing values (also an entire fragment), so we need to replace all of them. When two subsequent fragments has so many missing values and one of them is the first of the subsequent fragment, we replaced it considering the last element that is present in the previous fragment. The latter approach, if there are some consecutive missing values at the beginning of the fragment, will lead to a series of equal values. This is not the best approach, but is the most simple one to implement in an Android app. This replacement had a consequence on the computed features, because the computation of *Kurtosis* and *Skewness* values (done with the *Apache Commons Math API*[5]) leads to *"NaN"* value as result when there is a window with a series

[5]`https://commons.apache.org/proper/commons-math/`

of replaced and equals values. By inspecting the signal via *Excel*, we noticed that for the *Kurtosis*, in the window of replaced values, it returns always the value -2.041. Hence this will be the default value when the function returns *NaN* for the *Kurtosis*.
Regarding the *Skewness* values, we used the same approach as before and we noticed that the missing values tended to 0 for that feature. Hence, we decided to use 0 as default value in the window for the *NaN* result of the function.





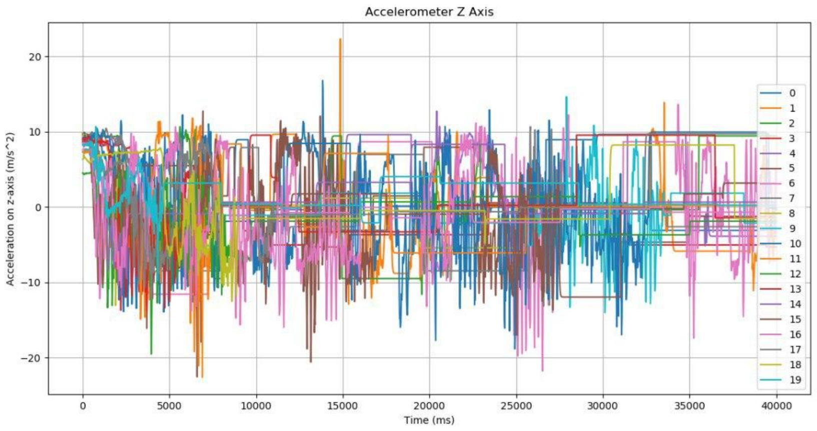


Figure 5: The signal of the non-wake-up accelerometer plotted for the 3 axis after the replacement of the missing values

2.3 Feature Selection

This phase has been performed in the *Weka* software GUI and the *".arff"* files are the input of the process.

Figure 6: The Weka software logo

As a result of the previous phase, we have 4 *".arff"* files, 2 coming from the non-wake-up sensors experiment, and 2 from the wake-up sensor experiment.
In order to select the feature, an evaluator and a search method has to be selected. After several tests and runs of the selection phase, the best evaluator has resulted as *InfoGainAttributeEval*[6] which evaluates attributes individually by measuring information gain with respect to the class. With the same premise, the best search methods has resulted the *Ranker*[7] which ranks attributes by their individual evaluations.
As result of the feature selection, the attribute selected has rank greater than 0. In our case, the selected attributes for the "non-wake-up" experiment are the following:

[6]`https://weka.sourceforge.io/doc.dev/weka/attributeSelection/InfoGainAttributeEval.html`

[7]`https://weka.sourceforge.io/doc.stable/weka/attributeSelection/Ranker.html`

AccX_win1_mean, AccX_win1_stDv, AccX_win2_mean, AccX_win2_stDv, AccY_win1_mean, AccY_win1_stDv, AccY_win2_mean, AccY_win2_stDv, AccZ_win1_mean, AccZ_win1_stDv, AccZ_win2_mean, AccZ_win2_stDv, GyrX_win1_mean, GyrX_win1_stDv, GyrX_win2_mean, GyrX_win2_stDv, GyrY_win1_mean, GyrY_win1_stDv, GyrY_win2_mean, GyrY_win2_stDv, GyrZ_win1_mean, GyrZ_win1_stDv, GyrZ_win1_skewness, GyrZ_win2_mean, GyrZ_win2_stDv, LinAccX_win1_mean, LinAccX_win1_stDv, LinAccX_win2_mean, LinAccX_win2_stDv, LinAccY_win1_mean, LinAccY_win1_stDv, LinAccY_win2_mean, LinAccY_win2_stDv, LinAccZ_win1_mean, LinAccZ_win1_stDv, LinAccZ_win2_mean, LinAccZ_win2_stDv, LinAccZ_win2_skewness, GravX_win1_mean, GravX_win2_mean, GravX_win2_stDv, GravY_win1_mean, GravZ_win1_mean, GravZ_win1_stDv, GravZ_win2_mean, GravZ_win2_stDv, RotPitch_win1_mean, RotPitch_win1_stDv, RotPitch_win1_skewness, RotRoll_win1_mean, RotRoll_win1_stDv, RotRoll_win2_mean, RotRoll_win2_stDv.

For the "wake-up" file the attribute selected are the following:

AccX_win1_mean, AccX_win1_stDv, AccX_win1_kurtosis, AccX_win2_mean, AccX_win2_stDv, AccX_win2_kurtosis, AccY_win1_mean, AccY_win1_stDv, AccY_win2_mean, AccY_win2_stDv, AccZ_win1_mean, AccZ_win1_stDv, AccZ_win1_kurtosis, AccZ_win1_skewness, AccZ_win2_mean, AccZ_win2_stDv, AccZ_win2_kurtosis, GyrX_win1_stDv, GyrX_win1_kurtosis, GyrX_win2_stDv, GyrY_win1_mean, GyrY_win1_stDv, GyrY_win1_skewness, GyrY_win2_mean, GyrY_win2_stDv, GyrY_win2_skewness, GyrZ_win1_mean, GyrZ_win1_stDv, GyrZ_win1_kurtosis, GyrZ_win1_skewness, GyrZ_win2_mean, GyrZ_win2_stDv, GyrZ_win2_skewness, LinAccX_win1_mean, LinAccX_win1_stDv, LinAccX_win2_mean, LinAccX_win2_stDv, LinAccY_win1_mean, LinAccY_win1_stDv, LinAccY_win1_kurtosis, LinAccY_win2_stDv, LinAccZ_win1_mean, LinAccZ_win1_stDv, LinAccZ_win2_mean, LinAccZ_win2_stDv, LinAccZ_win2_skewness, GravX_win1_mean, GravX_win1_stDv, GravX_win1_kurtosis, GravX_win2_mean, GravX_win2_stDv, GravY_win1_mean, GravY_win1_stDv, GravY_win1_skewness, GravY_win2_mean, GravY_win2_stDv, GravY_win2_skewness, GravZ_win1_mean, GravZ_win1_stDv, GravZ_win1_skewness, GravZ_win2_mean, GravZ_win2_stDv, GravZ_win2_skewness, RotPitch_win1_mean, RotPitch_win1_stDv, RotPitch_win1_skewness, RotPitch_win2_mean, RotPitch_win2_stDv, RotPitch_win2_skewness, RotRoll_win1_mean, RotRoll_win1_skewness, RotRoll_win2_mean, RotRoll_win2_stDv, RotRoll_win2_kurtosis,[8]

[8]The feature showed here for both the experiments regards the 4 seconds window file, that will result in the following section as the best one for the classifier

2.4 Choice of classifier

After the selection phase, a classifier has to be chosen in order to train and extract a model. The first kind of classifier that has been tested is the *Multi-Layer Perceptron*[9], a classifier that uses backpropagation to learn a multi-layer perceptron to classify instances.
After many training tests, the best classifier resulted to be the *RandomForest*[10] that create a forest of random trees.
For every test on the classifiers and for comparing them, we performed 10 runs with different seeds, and we took the mean of the accuracy in order to have a reliable statistic regarding the performance of the classifier. We also considered the False and True Positive produced by each classifier.
Regarding the two *".arff"* files (and the relative windows created) of the previous section, with the RandomForest, the best one resulted to be the 4 seconds windows even with a barely difference. For the "wake-up experiment", the mean True Positive rate for 10 experiments was of 119,8 for the "Washing Hands class" and 142,2 for the "Others class" in the 2 seconds file. And for the 4 seconds file the True Positive rate for 10 experiments was of 121,3 for the "Washing Hands class" and 145,2 for the "Others class". (The feature showed in the previous section are relative to the 4 seconds files, that were the best for classifier).

2.5 Implementing the classifier in Android

After obtaining the *".model"* file from the *Weka* software, this must be inserted in the final app, and has to be exploited in order to classify the activities.
The model has been put in the *"/app/src/main/assets"* folder in *Android Studio*, and it can be retrieved with the *"Context.getAssets()"* method and has to be assigned to the classifier object in the constructor.
Moreover, with the *Weka API*, we've been able to load the *"unlabeled.arff"* file, that contains the feature collected by the smartwatch that needs to be classified. This operation is achieved by *"Instances"*[11] class, that reads all the instances of a *".arff"* file.
Then, with the *"setClassIndex()"* method performed to the *"Instances"* variable, we assign the class index that is in the last line of the *@attribute* list, this line contains the labels to be applied to the signal that have to be classified.
Finally, with the *"classifyInstance()"* of the RandomForest class applied on the Instance, we obtain a double value that corresponds to the classified activity: if the value is equal to 0.0, the activity is classified as *"Others"*, otherwise if it's equal to 1.0, the activity is classified as *"Washing_Hands"*.
By testing many times the classifier in different situation, we discovered that the best one was trained with the "wake-up experiments". The "non-wake-up", even if the result in accuracy was very high, in practice misinterpreted many

[9] `https://weka.sourceforge.io/doc.dev/weka/classifiers/functions/MultilayerPerceptron.html`

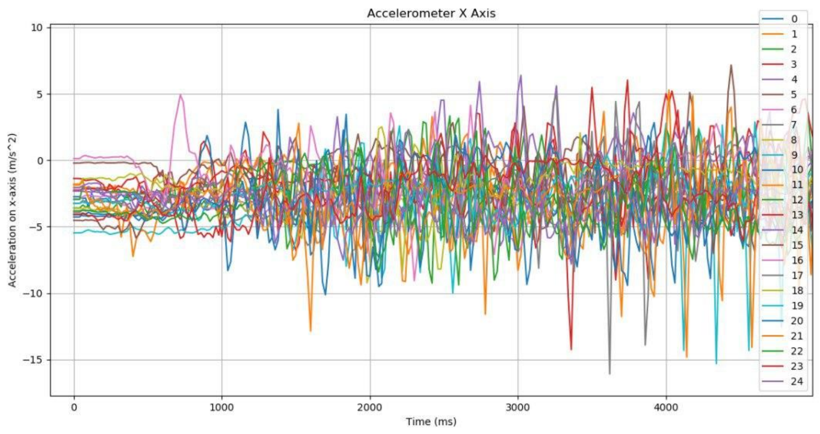
[10] `https://weka.sourceforge.io/doc.dev/weka/classifiers/trees/RandomForest.html`

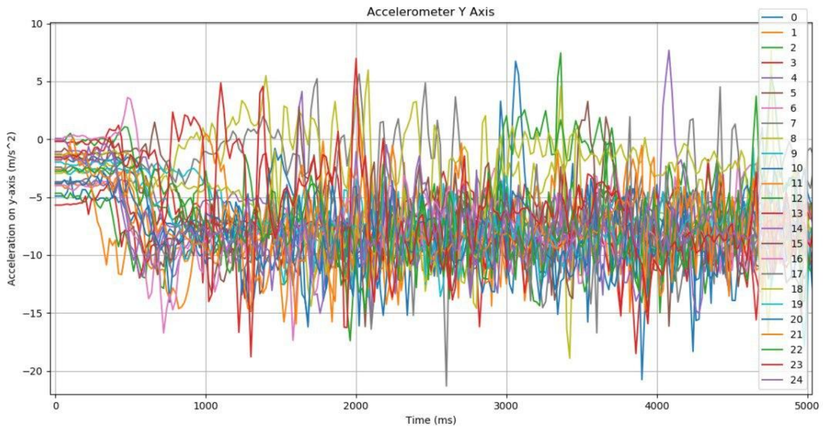
[11] `https://weka.sourceforge.io/doc.dev/weka/core/Instances.html`

classification. Hence, from now on we decide to proceed only with the ”wake-up” classifier.

3 Hand Activity Module

The detailed functioning of the module is the following: the service is started every time the user returns at his home. For that event, the module receives an intent in broadcast from the module that tracks the user's location; the received intent with the action attribute *"UserAtHome"* triggers the sensing process.
The sensing process is performed by the smartwatch module of the app, hence the smartphone, once it has received the intent, has to send a message to it in order to notify that the user is at home and to turn on the sensors. The message is sent by the smartphone with a "START" String, that will be interpreted by the smartwatch as a signal to start sensing.
In the smartphone, a timer of 7 minutes is then initialized and started, because it's the time that the user has in order to wash his hands. And now it waits for data coming from the smartwatch.
In the smartwatch, after the reception of the message, the sensors are turned on at low frequency with *SENSOR_DELAY_NORMAL* value in the Android Sensor API, until the values of the axis are at $-6.0 \leq X \leq -2.5$, $-9.0 \leq Y \leq 0.0$, $-10.0 \leq Z \leq 10$ for the accelerometer sensor, that corresponds to the stretched arms position (tested in the data collection phase), that is the position that people have in front of the sink while they're washing hands.





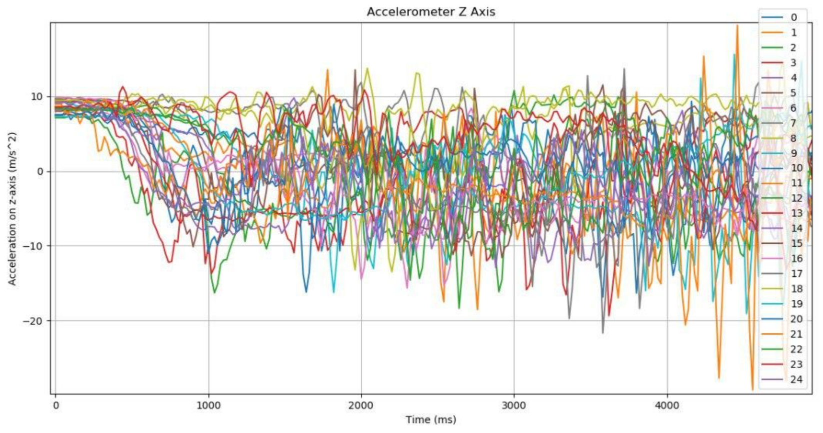


Figure 7: The signal of the wake-up accelerometer plotted for the 3 axis

Hence, if the accelerometer values reach that interval, the smartwatch starts sensing at high frequency, around (*50Hz*) for 10 seconds and sends the fragment to the smartphone (via message containing assets, with the same functioning of the preliminary app described before). The smartphone prepares the *".arff"* and submit it to the classifier. If the classifier detect an *"others"* activity, the smartwatch by default continue sensing at a low frequency until another "event" is detected, and so on and so forth. The process continues until the timer expires, then the smartphone can conclude the process by sending a stop message, that correspond to a "STOP" String, to the smartwatch and notifying the *risk index module* that the user did not washed his hands and to update negatively

the index. The service now can stop and wait for a new Intent, that corresponds to the returning home again of the user.
Otherwise, if the classifier returns *"Washing_hands"* activity before the timer expires, the process is stopped and the risk index can be updated positively. A "positive" Intent is sent to the *risk index module*. And like in the previous case, the smartphone sends a message to the smartwatch in order to stop the process and the entire service is stopped until the user exit and come back again at home.
Regarding the message exchange between the smartphone and smartwatch, it has been chosen a best effort approach. The message are sent, but there is no 100% guarantee that they will be received by the other part of the communication. Moreover, if no activity message is coming from the smartwatch in the range of 7 minutes (the timer), it is assumed that there is a communication problem and the risk index is not updated.
Also, the service on the smartwatch will be always active after the application is started. It has been used a *WearableListener* that will be activated every time a data is received. Hence, the smartwatch will be always ready to receive data from the smartphone. Moreover, it has been used the *DataItem* in order to enhance the communication because it is considered more reliable in delivering messages than *MessageClient*[12].
The *csv* files from the smartwatch to the smartphone, are sent every 10 seconds. This choice has been taken because if a *washing hands* is detected, the smartwatch's sensors can be turned off immediately. The alternative could have been to collect all data and send everything after 5 minutes, but this could be a waste of power, given that if a "washing hand" is in the middle of the "others" activity, the remaining part of sensed data are useless.

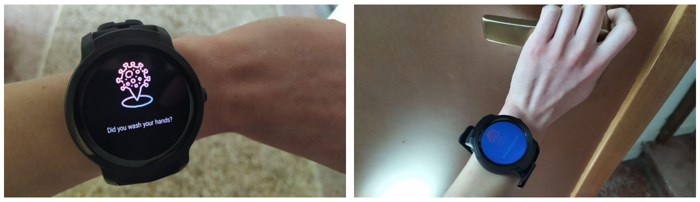


Figure 8: The app UI on the smartwatch

3.1 Code Structure

The Code in the *"washinghandrecognition"* package is divided into 2 modules: **mobile** and **wear**. Regarding the mobile part, the main classes are the following:

- *ClassificationService*: Performs the features extraction and the classification and sends an Intent to the *HandActivityService* with the result of the

[12]`https://developer.android.com/training/wearables/data-layer/messages#DeliverMessage`

classification.

- *Configuration*: It has some useful constants to be used in the other classes.
- *FeatureExtraction*: Creates an *".arff"* file needed as input for the classifier and computes the features. Handles missing samples replacing them with signal's mean.
- *HandActivityService*: *WearableListenerService* that manages the communication between mobile phone and smartwatch. Starts sampling on smartwatch when the user is at home and stops it when a wash hand activity is recognized or when the timer expired or the application is closed. Save data received from the smartwatch and sends an intent to *ClassificationService* in order to starts features extraction and classification operations. Sends the result of the activity classification in broadcast.
- *HomeReceiver*: Broadcast receiver that filters the intent addressed to our module when the user is at home and starts hands activity detection. Filters also intent containing results when an activity is recognized by the classifier.
- *MainActivity*: For our module, it istantiate the *HomeReceiver* and stops the *HandActivityService* when the application is closed by the user.
- *RandomForestClassifier*: Load the module generated from Weka and classifies the activities.

For the wear part, the main classes are:

- *Configuration*: It has some useful constants to be used in the other classes.
- *SensorHandler*: Starts or stop the sensing according to the command received by the *WearActivityService*. There are two timers:
  - A longer one used to specify the maximum sensing period.
  - A smaller one related to the period in which samples are collected with a faster rate, when values of the accelerometer corresponing to a possible in progress washing hands action is detected.
- *ServiceCallbacks*: it is a utility class for the setBackground method from the *SensorHandler* service.
- *WearActivityService*: Handles data sent from the mobile phone. It will start the sensing or stop it, according to the request from the phone. Send collected data to the mobile phone.
- *WearMainActivity*: the MainActivity for the wear module.

3.2 Used Libraries in code

- `com.google.android.gms.wearable` Used for implementing *Asset* and *WearableListenerService*.
- `com.google.android.gms.tasks` Used for the *onSuccessListener*.
- `weka.classifiers.trees` In this library there is the *RandomForest* classifier, which has been used in our application.
- `weka.core` Used for the *Instances* and *SerializationHelper*, that has been used for reading the *".arff"* files.
- `org.apache.commons.math3.stat.descriptive.moment` In this libraries has been used the *evaluate()* methods for the *Mean, Standard Deviation, Skewness, Kurtosis* classes
- `com.opencsv` This library has been used as utility for managing the csv files produced by the smartwatch.