Test of the AWARE framework

Romain Frigerio and Thomas Lucas

11 juin 2015

Contents

Chapter 1

Introduction

1.1 Context of this project

The goal of this project is to get to know a framework - called Aware - that allows a cellphone to determine its context of usage. The main goals are determining what can be done with the plugins already provided by the framework, test the reliability of these plugins, and assess the possibilities offered by the Framework. We will keep working with this framework in the long term (for our summer internship) therefore another important goal for us is to getting really familiar with the framework. From there and if there is enough time, we will propose alternative concepts that could then be implemented and tested.

1.2 Interest of context acquisition

Determining the context in which the phone is being used has many interests and applications. Traditional Artificial Intelligence is hindered by the infinite complexity of the world, but knowing a context an Artificial Intelligence has a way of deciding which behaviour is appropriate. For instance it is easier to recognize a scene when you know what to look for : imagine that you are capable of telling a robot that a user is cooking, then you can tell him to look for the table (after describing what a table looks like). In a nutshell, context allows Artificial Intelligences to know what behaviour is appropriate.

1.3 Our objectives

Some contexts are more easily discriminated than others, our goal here is to prove that one can reliably discriminate situations characterized by relatively straightforward criteria. For instance we will focus on guessing the means of transport used by the user, which could be still, walking, in a bus, in a car, etc. Here simple criteria, such as acceleration, vibrations and noise come to mind,

although less intuitive ones could be found. This task will be divided in two smaller tasks : testing an existing plugin (Google Activity Recognition), and making our own model.

Chapter 2

Mastering AWARE

2.1 Getting started with AWARE

2.1.1 Prerequisites

To reuse our work and understand what will be explained in this document, the following criteria need to be met:

- Android Studio is the IDE we used and on which we will base our explainations on. We recommended using it to follow our instructions.
- Having a good understanding of the JAVA language.
- Having basic knowledge in Android development.

2.1.2 How to install AWARE ?

The first thing you have to do is to install the AWARE framework. It is rather simple to achieve :

- Follow the link bellow and download the framework :

  http://www.awareframework.com/awareframework.apk

- Save the apk file on your Android platform (for example in the Documents/ folder).
- Run the apk file and follow the instructions.

2.1.3 How to install an AWARE plugin ?

Now that you have the AWARE framework installed on your Android platform, you may want to use plugins. There are two types of plugins : the official plugins provided by AWARE and the plugins developed by the community (such as your own plugin for instance).

How to install an official plugin ?

It is the most simple way to get a new plugin installed on your AWARE app.

- Run the AWARE app.
- Choose the Plugins menu in the list in the top left.
- Choose the Plugin to install and click on *INSTALL*.
- Follow the instructions.

How to install an unofficial plugin ?

You may want to use an unofficial plugin developed by yourself or someone of the community. If you don't have any or if you want to learn how to develop your own plugin, skip this part and go to the *'How to create your own plugin ?'*(2.1.4) section.

- Run Android Studio and open your project.
- Plug your Android device in your computer.
- Choose *RUN* or press *ALT+MAJ+X*.
- If a configuration window appears, check *'Do no launch Activity'*.
- Choose your device in the list and click on *OK*.

You should now see your plugin in the Plugins menu of AWARE and be able to activate it.

2.1.4 How to create your own plugin ?

Here is our tutorial on *How to create your own plugin* based on our experience. It tackles the problems we ran into and sums up the official tutorial that we used to learn and which can be found at :

http://www.awareframework.com/creating-a-new-aware-plugin/

If you haven't found the plugin you need, you may want to create your own plugin. To do so, you have to follow these instructions :

Getting the plugin template

- Run Android Studio and choose :

  VCS → *Checkout from version control* → *Github.*

- Paste the following link in the window that appears :

  https://github.com/denzilferreira/aware-plugin-template.git

- Change the parent directory and the directory name.
- Click on clone and wait.

If you encounter problems with Git, follow the instructions provided by Android Studio to fix it. After checkout, don't be concerned about any potential compilation errors.

Dissociate from Git

You need here to dissociate from the initial git repository. There are two steps :

- Choose *File* → *Settings* → *Version Control.* Click on the registered root in the list and then click on the red minus on the right. Click on Apply.
- Delete the .git file in your project repository

You are now working on your own project. You have to give it a custom name.

Custom the name of your plugin

To give your project a customed name you need to follow these steps :

- In your project tree, open app/java/.
- Right click on *com.aware.plugin.template* and choose :

  Refractor → *Rename* → *Rename Package.*

- Choose a name for your plugin and click on Refractor.
- If a list appears asking you to choose what to Refractor, check all and click on *Do Refractor.*

Sometimes some files where *'Template'* can still be found remain. In general, everytime you see *'Template'*, replace it by the name of your own plugin (you can use automatic search to do this). Here are the places where it is normally found :

- In some files in the res/ directory. Mostly in strings.xml.
- Go to *File* → *Project Structure* → *app* → *Flavors (Tab)* → *Application Id.*

The last one is essential because it will ensure that you will not override an other plugin and install your own plugin in AWARE.
You are now able to build your plugin clicking on the *'Sync Project with Gradle Files'* button. If you encounter any problem, just follow the instructions provided by Android Studio.

Understanding the project tree

If you want to develop your own plugin, you have to understand the provided project tree.

- **Plugin.java** : This file contains the core of your plugin. It implements the methods of the lifecycle of the app (onCreate(), onDestroy(), etc).

```
public void onCreate() {
    super.onCreate();

    /* Initialize our plugin's settings */
    if( Aware.getSetting(this, Settings.STATUS_PLUGIN_GETDATA).length()==0 )
        Aware.setSetting(this, Settings.STATUS_PLUGIN_GETDATA, true);
    }

    // Activate sensor (Accelerometer)
    Aware.setSetting(getApplicationContext(),
                     Aware_Preferences.STATUS_ACCELEROMETER,
                     true);
    Aware.setSetting(this, Aware_Preferences.FREQUENCY_ACCELEROMETER, 60000)

    //Ask AWARE to apply your settings
    sendBroadcast(new Intent(Aware.ACTION_AWARE_REFRESH));
}
```

```
public void onDestroy() {
    super.onDestroy();

    Aware.setSetting(this, Settings.STATUS_PLUGIN_GETDATA, false);

    //Deactivate any sensors/plugins you activated here
    Aware.setSetting(getApplicationContext(),
    Aware_Preferences.STATUS_ACCELEROMETER,
    false);

    //Ask AWARE to apply your settings
    sendBroadcast(new Intent(Aware.ACTION_AWARE_REFRESH));
}
```

- **ContextCard.java** : This file contains UI to render in the "stream" view of AWARE client. You have to know how to setup a graphic interface in Android to understand that file. Tutorials are easy to find on the Internet.
- **Settings.java** : This file is an exported activity, and it is the interface that appears when the user presses the settings icon on the list of available plugins in AWARE Client. You may use this to provide the user options to configure your plugin. You may also need to modify the *preferences.xml* file.

2.2 Getting data from sensors with AWARE

2.2.1 Using Database

We mostly used the 'passive' way to get data from sensors in this project. Let's see how to get data from the Accelerometer.
First you have to activate the sensor in your file **Plugin.java** (see section above). Then run a query in the database.
Here is an example which get data (X, Y, Z represented by VALUES_{0,1,2}) from the accelerometer for the timestamps between *'start'* and *'end'*. Then read the data and write it in a String called *'result'*.

```
public static String getAccelerometerData(ContentResolver resolver,
                                           long start,
                                           long end)
{

    // Query for data
    Cursor cursor = resolver.query(
            Accelerometer_Provider.Accelerometer_Data.CONTENT_URI,
            new String[]{Accelerometer_Provider.Accelerometer_Data.VALUES_0,
            Accelerometer_Provider.Accelerometer_Data.VALUES_1,
            Accelerometer_Provider.Accelerometer_Data.VALUES_2,
            Accelerometer_Provider.Accelerometer_Data.TIMESTAMP},
            Accelerometer_Provider.Accelerometer_Data.TIMESTAMP
            + " between " + start + "AND " + end,
            null,
            Accelerometer_Provider.Accelerometer_Data.TIMESTAMP
            + "ASC");

    // Got data ?
    if (!cursor.moveToFirst())
        return null;

    // Data in a string
    String result = "";
    do {
        double X = cursor.getDouble(cursor.getColumnIndex(
            Accelerometer_Provider.Accelerometer_Data.VALUES_0));
                double Y = cursor.getDouble(cursor.getColumnIndex(
                    Accelerometer_Provider.Accelerometer_Data.VALUES_1));
                double Z = cursor.getDouble(cursor.getColumnIndex(
                    Accelerometer_Provider.Accelerometer_Data.VALUES_2));

        result += X + "," + Y + "," + Z + ",\n";
    } while (cursor.moveToNext());
    cursor.close();

    return result;
}
```

2.2.2 Using listeners

If you want to have dynamic access to data you have to register a listener on the sensor. To do so, follow the code bellow :
For instance, we want to follow the screenLock action. First, create a new class extending BroadcastReceiver :

```
public class ScreenStatusListener extends BroadcastReceiver {
    public void onReceive(Context c, Intent intent) {
        /* Add code you want to execute when you recieve information.
        Information of the broadcast is contained in the intent; */
    }
}
Private static ScreenStatusListener ssl = new ScreenStatusListener();
```

Add the code in the *onCreate()* method to follow the action you want to track (refer to the aware website for more info)

```
IntentFilter broadcastFilter = new IntentFilter();
broadcastFilter.addAction(Screen.ACTION_AWARE_SCREEN_UNLOCKED);
registerReceiver(ssl, broadcastFilter);
```

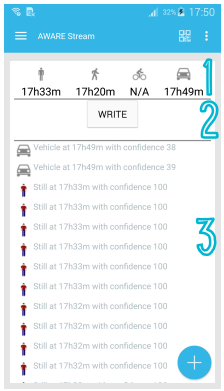
Don't forget to run the *unregisterReceiver()* method in *onDestroy()*.

2.3 Presentation of our modified and new plugins

2.3.1 Google Activity Recognition Modified Plugin

Graphic interface

First we overrode the *Google Activity Recognition Plugin* in order to have a graphic interface that suited our needs. You have to install the plugin before using it. If you don't know how to do it, please go to the section *'How to install an unofficial plugin'*(2.1.3) above.
Let's have a look at the interface :



The section number **1** is similar to the original interface but instead of showing the average time you spent still, on foot, riding a bicycle or in a vehicle, it shows the last time the plugin detected you were still, on foot, riding a bicycle or in a vehicle.
The section number **2** corresponds to a way for the user to store the ten last

activities detected by the plugin. They are stored in a .txt file called *'Log-Google.txt'* stored on your device in *'Documents/Logs/'*. Here is an idea of what the logs look like :

```
Time : 14h16m / Activity Unknown / Confidence 77
Time : 14h16m / Activity Vehicle / Confidence 92
Time : 14h16m / Activity Vehicle / Confidence 77
Time : 14h16m / Activity Vehicle / Confidence 62
Time : 14h15m / Activity Vehicle / Confidence 67
Time : 14h15m / Activity Tilting / Confidence 100
Time : 14h15m / Activity On foot / Confidence 100
Time : 14h15m / Activity On foot / Confidence 92
Time : 14h15m / Activity On foot / Confidence 100
Time : 14h15m / Activity On foot / Confidence 100
```

The section number **3** shows the real time logs of the twenty last activities detected by the plugin. For each activity, if the confidence is greater than **70** then the icon is colored.

NB :

- Even if the plugin works in real time, if you want to see the logs in the interface you have to refresh the stream by choosing :

  Menu (Top Left) $\rightarrow$ *Stream.*

- You can choose the frequency of detection by choosing :

  Menu (Top Left) $\rightarrow$ *Plugins* $\rightarrow$ *Google Activity Recognition* $\rightarrow$ *Settings.*

Tree of the project

If you don't know yet what these files are : **Plugin.java**, **ContextCard.java**, **Settings.java** please refer to the *'How to create your own plugin ?'*(2.1.4) section.
We modified the **Stats.java** file from the original one to fit our needs by adding *getLastTime{still, biking, vehicle, walking}* methods to get the last time the plugin detected these activities and write it in the interface presented above.
We modified the **ContextCard.java** file by adding a *setInfo()* method which update the logs in the graphic interface by choosing what icon to render.
We added some code to write logs in a file. Here is the code used :

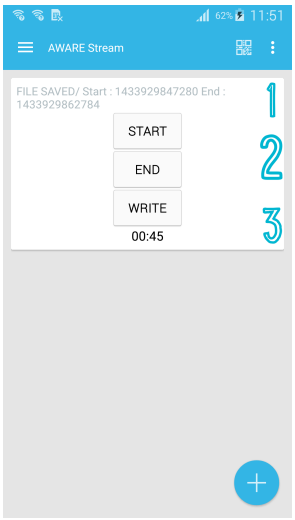
```
public static FileOutputStream fOut;

/* Checks if external storage is available for read and write */
private static boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    return Environment.MEDIA_MOUNTED.equals(state);
}

private static File getStorageDir(String name) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_DOCUMENTS), name);
    if (!file.mkdirs()) {
        Log.e("LOG", "Directory not created");
    }
    return file;
}


// Writing data in a file
File dir = getStorageDir("Logs");
File file = new File(dir, "LogGoogle.txt");
try {
    if (!dir.exists())
        dir.createNewFile();
    if (!file.exists())
        file.createNewFile();

    // true for append
    fOut = new FileOutputStream(file, true);

    } catch (IOException e) {
        e.printStackTrace();
}
```

2.3.2 GetData Plugin

Graphic interface

To develop our own model, we had to create a plugin that retrieves data from the sensors and write it in a log file. Before being written, the data is processed (see 2.4). That is why we created a graphic interface to perform this task.

The section number **1** is a debug String with the timestamps *start* and *end*. It only appears when we push the *write* button.
The section number **2** is the main part of this interface. It provides three buttons :



- A button *Start* which store the timestamp of the begining of the data retrieval.
- A button *End* which store the timestamp of the end of the data retrieval.
- A button *Write* which run the process method and then write the result in a log file.

The section number **3** is a chronometer which allow us to track the time spent on the retrieval.

Tree of the project

If you don't know yet what these files are : **Plugin.java**, **ContextCard.java**, **Settings.java** please refer to the *'How to create your own plugin ?'*(2.1.4) section.
We created some classes in order to process the data :

- **DataHandler.java** is the main class of the plugin. It provides several method.
  - *getData{Accelerometer, Gyroscope}()* are the methods used to retrieve **raw** data from those sensors and write it in a log file called *Log{Accelerometer, Gyroscope}.txt*.
  - *getData()* is the method that combine the two methods cited above.
  - *getDataProcessedInLog()* is the method that retrieve the data from the Accelerometer, process it by applying *PCA*, *FFT*, *Summary* and then write it in two files called *LogSimple.txt* (with all the data processed) and *LogSummary.txt* (with only a summary of the data). For more information, please refer to the section 2.4.

- **FFT.java**, **PCA.java**, **Summary.java** implement statistic and analytic tools to process the data.

2.4 Processing data

2.4.1 Acquiring data

In our project, we had to determine if a user was still, on foot, biking or in a vehicle. To do so, we chose to retrieve data from the Accelerometer. However, it is possible to get access to a lot of other sensors such as Gyroscope, Luxmeter, Battery, Barometer etc. AWARE provides us an easy access to these sensors. You need to :

- Add this code in **Plugin.java** :

```
public void onCreate() {
...
        Aware.setSetting(getApplicationContext(),
        Aware_Preferences.STATUS_SENSOR, true);
...
}

public void onDestroy() {
...
        Aware.setSetting(getApplicationContext(),
        Aware_Preferences.STATUS_SENSOR, false);
...
}
```

- Add this code when you want to get the data (SQL query) :

```
Cursor wanted_data = getContentResolver().query(
    SENSOR_Data.CONTENT_URI,
    tableColumns, whereCondition, whereArguments, orderBy);
```

NB : All the constant values you have to provide in the last method can be found on the official AWARE website in Documentation/Sensors.

2.4.2 Preprocessing the data

The data returned by the query cannot be used "as is". Indeed one must first make sure that random phenomenon do not make the data unusable. In our case, there are two such random phenomenon.

Straightening the phone

The orientation of the phone inside the pocket of the user can be considered random. This makes it impossible to use accelerations without treating them. Let us say that if the phone were straight the X acceleration would be high. The fact that the phone can be randomly oriented will randomly project that acceleration on other axis or randomly give it a negative sign. Now the way the regression works is it detects that a certain type of activity is charachterized by a high value on a certain feature, and rewards that by giving that feature a high coefficient. But if the orientation of the phone randomly projects this high value on other features or give it a negative signe, the mean over a large set of examples for this feature will we zero, and therefore the logistic regression will never be able to use the information.
To solve that problem, we perform a principle component analysis on the data. Corresponding code can be found in file *PCA.java*. PCA uses the fact that covariance is a scalar product to build a new Base for a vectorial space (here, our 3D world), in which the new vectors in the base have no covariance (are orthogonal to each other for the covariance). This is possible thanks to the spectral theorem. Intuitively, it will determine the three direction along which movement is the most important. The important thing is that they will be the same no matter the orientation of the cell phone. The three directions are orthogonal to each other, and they are ordered by decreasing order of variance. The signature of the function is the following :

```
public double[][] determine_PCA(){
```

It returns a matrix which we can use to change base, and consider that we have straightened the phone. The file is abundantly commented for more detail.

then the following line computes the new coordinates:

```
new_coord = pca.changeBase(newBase,x_series1,y_series1,z_series1);
```

Making a summary of the data

We first apply *Fast Fourier transform* on the data. The data produced by the accelerometer is temporal, over an interval of ten seconds. We use the fft because we intuitively feel that walking will create some frequencies in the accelerometer, for instance. What's more data should not depend on time, because this includes a random component (just as the random orientation of the phone does). For a person walking, for instance, values of features would depend on wether the person is finishing or beginning a step at the start of the interval : for a person beginning a step, Y acceleration would be high, and low for a person finnishing a step. Just like the orientation of the, this adds a random sign that makes the data unusable. FFT takes care of all this.
Once the fft has been perfomed we make a summary of the arrays : the corresponding code can be found in *Summary.java*. It will simply add statistical measures to the feature vector : mean, standard deviation, skewness, kurtosis, max and min.

During our first tests we found out that our model was performing very well, and therefore we created an alternative version where feature vectors are smaller : we grouped frequencies together by groups of 20 (making a sum for negative numbers and one for positive numbers for each of these groups). We found out that this was sufficient data for the classifier, but we kept the first version of getData because more features might be necessary to detect complicated activities in the future.

Possible new sources of data

Other sources of data could have been used and would be used in future devellopement of this project. We would have begun with already existing plugins such as the gyroscope.

2.5 Reliability of the Google Activity Recognition Plugin

2.5.1 Protocol

In the test we made, we wanted to **determine the probability of succes in different situations**. Those situations were : Being still, being on foot, riding a bike, being in a vehicle. To do it properly, we defined a test protocol :

- Choose an activity to test.
- Choose an interval of time for the test during which we *strictly* do the activity tested. We cannot use the phone for another purpose when we are testing it.
- Write logs thanks to the graphic interface *every two minutes*. (ref. **??**).
- Compute the empirical probability of success thanks to : $p = \frac{\#Success}{\#Logs}$.

Why is this approach valid ? Because activities are separated, the experiments can be considered as *Bernouilli variables*. The result of an experiment can be Right with a probability p that we are trying to determine, or wrong with a probability $1-p$. That being said, we are trying to use our data to estimate p. It turns out that the most classical parameter estimators is statistics, maximum likelihood and moment estimators, are both equal to $\frac{\#Success}{\#Logs}$. As is usual in parameter estimation, we would like to know how many measures must be made to have accurate estimations. To answer that question , we must determine a confidence interval.
If all experiments are considered identical and independant one can apply the *Central limit Theorem*, which states that : $S_n = \frac{X_1+...+X_n}{n}$ is distributed as

$\mathcal{N}(m, \sigma^2)$ when $n \to \infty$ and where m is the mean and σ the standard deviation. This can also be written

$$\sqrt{n}((\frac{1}{n}\sum_{i=1}^{n} X_i) - m) \to \mathcal{N}(0, \sigma^2) \tag{2.1}$$

So we can use this formula to have approximated intervals for any precision wished (that would determine the number of measures that has to be done) or we can choose to have a fixed confidence (let us pick 0.95 and then the interval will tighten with n.

2.5.2 Results

Using the protocol described above, we got those results :

- **Activity** : Still / **NumberOfLogs** : 500 / **Accuracy** : 99%
- **Activity** : Walking / **NumberOfLogs** : 300 / **Accuracy** : 98%
- **Activity** : Vehicle / **NumberOfLogs** : 200 / **Accuracy** : 95%
- **Activity** : Biking / **NumberOfLogs** : NonTested / **Accuracy** : NonTested

2.5.3 Observations

Given these results, we can say that the Google Activity Recognition Plugin is accurate in our experiences. Nevertheless, these test were made in *'ideal'* conditions. Indeed, we were doing *only one activity*, during a *long period*, with the phone *in our pocket*. But, we noticed that when we changed the location of the phone during the experience (for instance from the pocket to hands) the plugin started to detect absurd things or not detect anything at all.

Chapter 3

Alternative methods for activity recognition

3.1 Introduction

Our goal here is to imagine our own plugin for activity recognition. To that end, we will rely on the phone sensors to acquire data and on classical machine learning methods to create the statistical model. Our goal is to be capable of predicting the current activity of the smartphone user given some data that is acquired by sensors in the smartphone. In machine learning, activity recognition is a particular case of classification: there is a finite number of activities, which will be our classes (there can be an "unknown activity" class if unknown activities must be classified), and data is used to pick the most likely class.

3.2 General mindset of classification problems

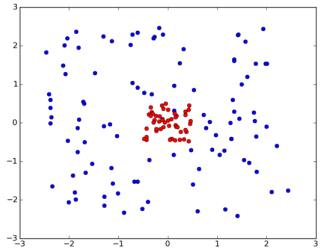
Let us first present the general mindset of classification problems, as well as how our activity recognition problem relates to it. The presentation of machine learning will be succinct and can be either a reminder for those who have already studied machine it, or a way to get the most important concepts for those who have not.

3.2.1 Learning from a training set

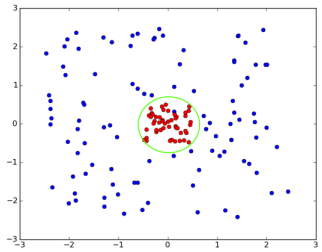
Fomalizing the problem

In classification problems a Data set is given. It is composed of a succession of vectors that each represent one experiment. The class to which each experiment belongs can be given (in which case it is a supervised classification problem) or unknown (unsupervised classification problem). Machine learning algorithms then have to learn from this Data set, so that it can make predictions given

new data. For example imagine that we have two classes, and we are given two features to describe each experiment as well as the class to which they belong. Our data could be represented like that :



Then this data would be used to create a statistical model, which would allow us to determine whether a new point should be colored in red or in blue (that is to say, does it belong to the first or the second class?). In two dimensions, such a model could look like the following decision boundary (many different models exist):



In that case a new example would be classified as red if it was inside the green circle. Such a model can make mistakes: it does some even on the training set (which is not separated by the model).

Later on, we will present our actual work and results. We now know that our first task will be *producing a Data Set*, by performing a certain activity and recording data produced by chosen sensors during the activity. That data can easily be labeled because we know what we are doing while we are doing it, so it is a supervised classification problem. We have created a plugin which produces that data.

Training a model

Once the data set is acquired, the goal is creating a statistical model which will be trained on it. First a type of model has to be chosen. It will be a function which, given features, will output a class or a vector of probabilities (each probability being associated with a class). This function will depend on

parameters, therefore the goal is to find the parameters that produce the most accurate guess. Once this is done, the model is capable of making a prediction given so far unseen data. Measurements of how well the model performs then need to be done. A first step is seeing how well the model is doing on the training set. Typically this involves computing a "Cost function" that is a measure of the amount of mistakes a given model is making on the training set.

Cost function: The cost-function has to measure an "average distance" between predictions made by the model and reality. Suppose we have the following :
- An array of parameters θ
- m examples in our data set : the i-th example is noted $x^{(i)}$ and $y^{(i)}$ is a vector where $y_j^{(i)}$ is equal to 1 if exemple i belongs to class j and 0 oterwise.
Our model $h_\theta(x^{(i)})$ which outputs a vector of probabilities, $h_\theta(x^{(i)})_j$ being the probability of belonging to class j.
- The cost function we use to measure our performance is then the following (keep in mind that y and $h_\theta(x^{(i)})$ are vectors):

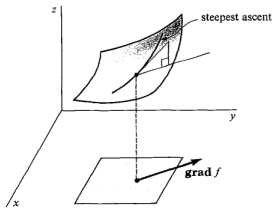
$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))].$$

The idea is simple : when $y_j^{(i)}$ is equal to 1, $h_\theta(x^{(i)})_j$ must be close to 1 (and therefore $log(h_\theta(x^{(i)})_j$ close to 0) for the sum to be low, and conversly if $y_j^{(i)}$ is equal to 0 then $1 - y_j^{(i)}$ is 1 and $(1 - h_\theta(x^{(i)}))_j$ must be close to 1 for the sum to be low.

Fitting the parameters: The cost function gives us the intuition of how parameters will be chosen: once a model is determined we will chose the parameters which minimize the cost function over the training set using numerical optimization techniques, and we will keep the corresponding parameters. The model and the parameters will then be re-used to classify new, so far unseen examples. The question of how to chose the models remains: typically numerous models are tried and the best one is kept. We used very classical models for our classifier.

Gradient Descent: Different techniques exist to minimize the cost function. There are exact techniques, but they typically become computationnally very expensive when the number of features increases. Approximation methods are often much more efficient, and some can be made as precise as wished (but cost increases with complexity). We will use a technique called gradient descent, the most classical one to perform this task in numerical optimization. This technique is adequate because our cost function is a multivariate differentiable function. The intuitive idea of gradient descent is that in a given a spot, the

gradient (computed in that spot) points in the direction in which the slope is greatest. Here is an exemple of this phenomenon :



This is only local information (the direction can change very quickly) but this still gives a way to decrease the value of the function: going in the opposite direction. This is the basic principle of Gradient descent algorithms. Another problem is not going "too far" in the opposite direction, because as we said gradient offers only local information. Gradient descent algorithms therefore have to take care of that. To perform this task we will use optimized functions given in Octave (this is the software we use for our regressions).
To minimize our cost function, these optimizers require us to compute the cost function and the gradient in given spots. We will later present how to compute the gradient. Different functions for optimization can be found, for example *fminunc* is already implemented in octave.

Mathematically, the gradient is determined using partial derivatives :

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)})$$

3.3 The problem of over-fitting

3.3.1 Intuition of the problem

Although the presented cost function is a very useful tool, it poses a great and subtle problem: overfitting. If a very complex model can be chosen, it will be capable of fitting the data very closely. Sticking to the training set very closely will make for a very low cost-function, but the model will often generalize very poorly.

Indeed, it will typically be making huge efforts to explain an irreducible variance (which comes from the random nature of the training set) and will "explode" on new examples, often to the point of performing much more poorly than more simple exemples, which is at first very counter-intuitive. This problem arises if the training set is "too small" compared to the complexity of the model we are trying to train. Paradoxically, if the training set is too small a complex model will perform much worse than a simpler one. Luckily, this phenomenon is easy to spot, if not easy to solve: the model can be trained with a learning set and tested on data that has not been used to train it. A model that suffers from over-fitting will perform much worse on new data. Comparing measures

of accuracy on a training set and on a test set is therefore all the indication we need.

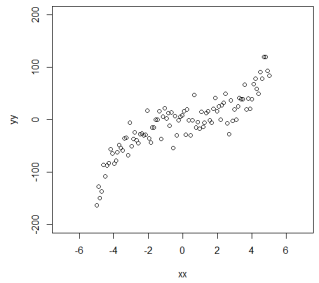
3.3.2 Example of over-fitting

Let us give a simple example of when over-fitting may arise -This exemple is taken from Ensimag's Data-Mining course allthough we made our own R code-. We will assume that linear regression is known by the reader. If that is not the case refer to the part on logistic regression, where linear regression is briefly explained. We will create a set of points distributed as a third degree polynomial, to which we will add a random gaussian term symbolizing the irreducible error that should not be explained by the model.

The code (in R) is the following :

```
> xx<-seq(-5,5,length=100);
> yy<-xx-xx^2+xx^3+rnorm(100,mean=0,sd=20); %Third degree polynomial
> plot(xx,yy,xlim=c(-7,7),ylim=c(-200,200))
```

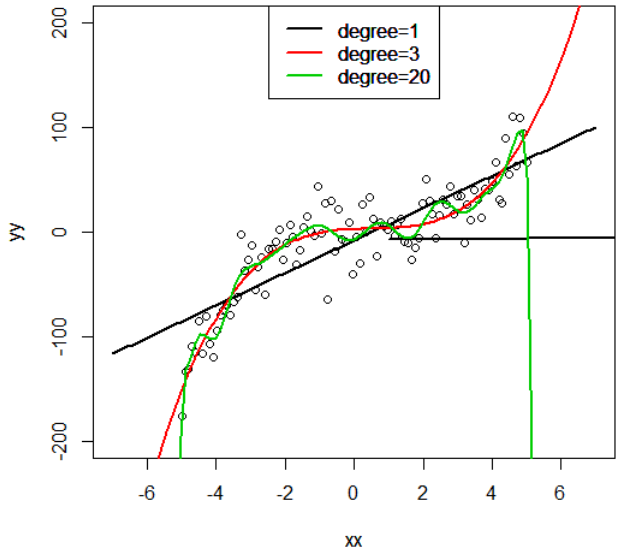
This is the result we can get :



Now we will create statistical models that learns from the dataset and will enable us to predict new values given a new x. We know how the exemple was built: it is the sum of the polynomial term, which we want to explain, and a random error which should not be explained by the model (irreducible variance). Let us regress using three different polynomial models, with degrees 1,3 and 20. We know that the second one is the adequate model. (Note that in R, linear regression is called with lm)

```
zz<-seq(-7,7,length=100)
degree<-c(1,3,20)
for(i in 1:3){
   fm <- lm(yy ~ poly(xx, degree=degree[i],raw=T))
   lines(zz, predict(fm, data.frame(xx=zz)), col = i,lwd=2)
}
legend("top",col=1:3,legend=c("degree=1","degree=3","degree=20"),lwd=c(2,2,2))
```

gives the following result :



Here the model with one parameter is clearly unsufficient. This is because it is not complex enough and incapable of explaining enough variance. The model with three degrees fits the data and generalizes well, which is reassuring regarding the way we built the data set. The model with 20 parameters makes huge efforts to explain variance that should not be explained (due to the random term we added, and therefore impossible to predict) and will therefore have a low cost-function on the training set. It Completely fails to generalize however, as the complexity built to fit the parameters explodes outside the training set. Therefore, the cost-function computed on a test-set would be very high.

3.3.3 Solutions

In the presence of over-fitting, numerous things can be done. If it is an option, increasing the size of the data set will allow more complex models to perform better. If it is not, a simpler model can be chosen, or the cost function can be modified to penalise parameters that make huge efforts to explain the irreducible variance of the data (this is the variance that should not be predicted by the model because it is completely random). To penalise complex model a new term can be added to the cost function : assume array θ is an array containing our parameters, $\lambda \sum_{i=0}^{n} \theta_i^2$ can be added to the cost function. It is called a regularization term. This will prevent a parameter from having high values : the gradient will only allow a parameter to have a high value if it is very powerfull to explain the model (in that case θ_i^2 will be high but it will be compensated by a decrease in the rest of the cost function. The greater the need for regularization, the bigger λ should be. The regularized cost-function is, n being the numer of

parameters:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)})) + \frac{1}{2m}\sum_{i=1}^{n}(\theta_i^2)].$$

And regularized gradient is :

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}) + \frac{1}{m}\sum_{i=1}^{n}(\theta_i)$$

3.4 Statistical models used in this project

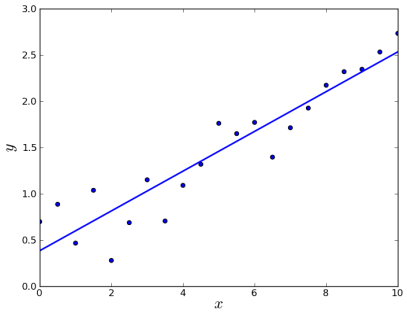
Now that we have presented the general mindset of Machine Learning, as well as the main problems that can arrise, let us present the techniques we used in our work.

3.4.1 Bits of theory for logistic regression

The first technique we will use can be considered the most classical and intuitive one in Machine-learning : Logistic regression. It is based on linear regression, which we will quickly present :

Linear regression:

Suppose we are given a set of two-dimensional points. The goal of linear regression (in one dimension) is finding the best-fitting straight line through the points. The best-fitting line is called a regression line.



At first, this model may seem very weak. Indeed most phenomenons in real life are not linear, and those who are often are not the most interesting ones to observe. If a phenomenon is quadratic for instance, even the best-fitting straight line will fit very poorly except in a few spots.
This is not, however, so much of a problem : new variables can be built using the given ones. If the set of points seems to be quadratic then $u_i = x_i^2$ can

be computed. Then a one variable linear regression can be done using the set u_i,$i \in I$. The result will be a line when u_is form the horizontal axis but a quadratic curve if we go back to using the x_is as horizontal axis. If we suspect that the result depends on both x and x^2, we can keep the two features. We then have two coefficents to find, which brings us to multidimensional regression.

Multidmensional linear regression:

For multidimensional linear regression, we use the following model :

$$Y_i = \beta_0 + \sum_{j=1}^{p}(x_{ij}\beta_j) + \epsilon_i$$

Here again new features can be computed, for exemple if x_p is given but the result is likely to depend quadraticaly on X_p a new feature $X_{p+1} = X_p^2$ can be computed and the model will be

$$Y_i = \beta_0 + \sum_{j=1}^{p}(x_{ij}\beta_j) + x_{ip}^2\beta_{p+1}\epsilon_i$$

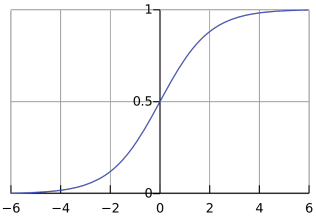
So the new model now depends quadratically on X_p. As was explained about the cost-function (3.2.1, page 21), the model is fitted using a cost function. In the case of linear regression we use the residual sum of squares, wich measures the squared distance between predicted points and observed data :

$$RSS(\beta) = \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2$$

This can be minimized using gradient descent or with the exact solution.

Logistic Regression:

We now have presented all the concepts required to build a Logistic Regression. Linear Regression allows us to approximate a real-valued function. To build a two-class classifier we will simply map the result of a linear regression to $[0; 1]$ and interpret it as the probability of belonging to a class. We well then decide that a point belongs to the class if the probability is higher than $\frac{1}{2}$.
- To map the result to a probability we use the "sigmoid function" : $h(z) = \frac{1}{1+e^{-z}}$. This function strictly increases on $\mathbb{R}$, maps $-\infty$ to 0 and $+\infty$ to 1. It looks like this :



So given θ an array of parameters, our model is :

$$h_\theta(x) = \frac{1}{1 + e^{-(\beta_0 + \sum_{j=1}^{p}(x_{ij}\beta_j) + \epsilon_i)}}$$

Of course, the cost function no longer is the Residual sum of square, but rather the cost function introduced in 3.2.1, at page 21. The parameters are once again chosen by minimizing the cost function using gradient descent.

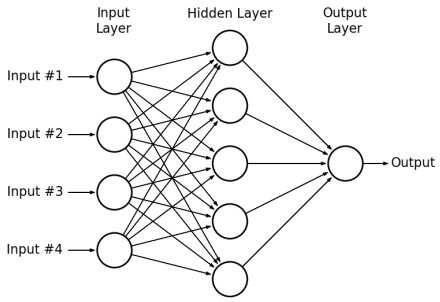
Limitations of logistic regression:

Logistic regression can be very powerfull if the features that should be built are known before hand
For instance let us imagine that we are trying to find the coefficients of air friction and that some experiments have already shown that friction depends on polynomials of the speed, then given the speed we will simply compute the polynomials and perform the regression.
However there are cases where the right features that should be computed are not easy to guess. One could take the features given and compute a bunch of polynomials, logs, exponentials with them, but if there are many variables already this will increase their number greatly, especially if products between given features are made (length time width to compute area). This will make the model very expensive to train, and give no garantee that good features have been computed. It may also foster overfitting if too many features are computed : indeed, increasing the number of features increases the complexity of the model, and the size of the data-set must then be increased.
Neural networks offer a solution to this problem.

3.4.2 Bits of theory for Neural networks

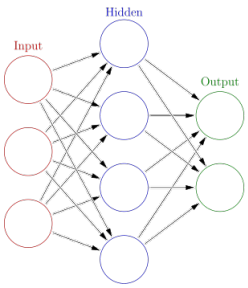
Introduction:

One of the great strength of neural networks is that they build their own features. Neural Networks can be seen as combinations of different logistic regression, that can use each other. Each logistic regression will correspond to a neuron. Those neurons will not be computing our final answer but rather intermediate features, that will be in turn used to compute new features or, at the end of the process, outputs. A neural network can be represented thus :



Each neuron of the hidden layer computes features using the input layer (our features), then the neuron computing the output uses the results of the hidden layer to perform it's logistic regression.

If more than one class have to be detected, the output layer has more neurons :



Now, our model is more complex. With two layers and N neurons on the hidden layer the equation for one output looks like this (this is painfull to read, but find solace in the fact that is was more painfull to write):

$$h_\theta(x) = \frac{1}{1 + e^{\beta_0^{N+1} + \sum_{k=0}^{N} (\beta_k^{N+1}) \left(\frac{1}{1 + e^{\beta_0^k + \sum_{j=1}^{p} (x_{ij} \beta_j^k) + \epsilon_i}} \right)}}$$

.

This model is complex, but the cost function has not changed, and the method for optimizing it also remains the same. The computation of the gradient is where the difficulty lies, but an efficient algorithm exists : Backpropagation.

3.5 Our implementation of these methods:

The octave files are abundantly commented and should be refered too for a detailed understanding of how things work. Here we will only present the key

aspects of the implementation. Implementation has been done in Octave, the files should be usable with Matlab with little to no modifications.

3.5.1 Implementation of logistic regression

The implementation is divided into the following files :

```
logisticReg.m  % Main file
lrCostFunction.m % Compute cost and gradient given a training set and parameters
oneVsAll.m % Finds the best parameters for each classifier
predictOneVsAll.m % Uses given parameters to classify a given entry.
```

The main frame is the following:

Parameters have to be set up at the beginning :

```
%% Setup of the parameters that we will use for one-vs-all classification
input_layer_size  = 768;  % Number of features
num_labels = 2;          % number of classes
```

Then Data is loaded and divided into training and test set:

```
data1 = load('data_sstill.txt'); % training data stored in arrays X
[...]
y = data(:, end); %Getting the labels
```

Parameters are then trained using *oneVsAll.m*, our constants and a regularization parameter :

```
[all_theta] = oneVsAll(X, y, num_labels, lambda);
```

Then we use the parameters learned to test the model on our training set.

```
pred = predictOneVsAll(all_theta, X);
```

Finally accuracy on the train and test set are displayed :

```
fprintf('\n Our training Set Accuracy is: %f\n', mean(double(pred == y)) * 100);
pred_test = predictOneVsAll(all_theta, X_test);
fprintf('\n Our test Set Accuracy is: %f\n', mean(double(pred_test == y_test)) * 100);
```

The most important snippets of code are the following :

Learning the parameters: (in *oneVsAll.m*)

```
initial_theta = zeros(n + 1, 1);
options = optimset('GradObj', 'on', 'MaxIter', 100);
for i=1:CLASS_NUMBER
[theta] = ...
fmincg (@(t)(lrCostFunction(t, X, (y == i), lambda)), ...
initial_theta, options);
all_theta(i,:) = theta;
endfor
```

This snippet calls fmincg to optimize the cost fonction over each class, giving some parameters such as the Miximum number of iterations allong the way. It then returns the parameters learned in all_theta. It uses *lrCostFunction.m0* which computes the cost function and its gradient :

```
J = (1/m)*sum( -y .* log(sigmoid(X*theta) ) -(1 .- y) .*log(1 - sigmoid(X*theta) )) +
(lambda/(2*m))*(theta'*theta - theta(1)^2);

grad = ((1/m)*( ( sigmoid(X*theta) .- y )' * X)') .+
(lambda/m).*([0;theta(2:(length(theta)))]);
```

Note that the computation of the cost function and gradients is vectorized. This is much more efficient than using loops and makes for shorter code, but it also makes it harder to write. Let us recall the formulas :

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)})) + \frac{1}{2m}\sum_{i=1}^{n}(\theta_i^2)].$$

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}((h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}) + \frac{1}{m}\sum_{i=1}^{n}(\theta_i)$$

The code is easy to compare to the equation, but one must be very carefull while writing it : for exemple X is a matrix, theta a vector etc.. it is easy to make a mistake in the order in which things are written, placing the transpose well, etc.

3.5.2 Implementation of Neural Networks

The main frame of the implementation for neural networks is the same as for logistic regression, and will therefore not be detailed again. We believe the code to be well commented for deeper understanding of it. The main difference in the code is that there are more parameters and a clever way of computing the gradient is required : the backpropagation algorithm is the right way to compute it. The code is hard to justify mathematically, and harder still to explain intuitively : we tried and gave up. We will simply give a mathematical result: Let us note θ_{ij}^k the weight associated to the j^{th} feature of the i^{th} neuron on the k^{th} layer. We will also note E the energy that we are trying to minimize, then it can be shown that : $\frac{\partial E}{\partial \theta_{ij}^k} = x_j^{(k-1)}e_i^{(k)}$ where $e_i^{(k)} = g'(h_j^{(n)})$ and the algorithm to which this formula leads : (we do not believe that this can be understood without further research on the internet, wikipedia explains it well but it is not concise).

Here is our implementation of Backpropagation:

```
 %BackwardsPropagation
DELTA2 =zeros(num_labels,hidden_layer_size +1);
DELTA =zeros(hidden_layer_size,input_layer_size+1);
```

```
for t=1:m %We iterate on all the training set.
  a = X(t,:);
  a = a';
  z2 = Theta1 * a;
  a2 = [1;sigmoid(z2)];
  z3 = Theta2 * a2;
  a3 = sigmoid(z3);

  delta3 = a3 - Y2(t,:)';
  delta2 = (Theta2(:,2:end)'*delta3) .* sigmoidGradient(z2);

  DELTA = DELTA + delta2*(a');
  DELTA2 = DELTA2 + delta3*(a2');
endfor
```

3.5.3 The results obtained

To first test our octave scripts, we will first our data set in two sets : a training test and a test set. The complete procedure would require us to divide our set into three : one for learning , one for selecting a model and one for testing. We will skip the set used for selecting the model because we will not perform this task : we will simply test 4 models. The 4 models will be : logistic regression with all the processed data, logistic regression with the summarized feature vector, and one Neural Network with 2 layers (25 neurons on the hidden layer) with the same data. With more time we would have compared models and looked for the best ones, it would not be that long but we managed to get everything working smoothly too late.

- If the only class that we try out are Walking and Still, we achieve a 100 % accuracy (only once did our Neural Network missclassify one single example) on the test tests, no matter what method we try out. This can look suspicious but is not acctually that incredible : it is very easy to separate still (and fidgeting slightly) from walking even by simply looking at the data.
- If we add "running" to the linear regression, the results drop to 94 %accuracy with summarized data, and 95.5% accuracy.
- We have not tested the accuracy of the neural network (which is a shame, but life can be tough).

3.6 Training our classifier to detect new activities

The great thing about our two alternative models is that they can be fairly easily extended to detect new classes. The octave scripts would not be modified except for constants parameters.

3.6.1 Detecting a new activity:

New activities can be almost automatically added to the classifier. That does not mean that this activity will be accurately detected of course, but for the time being let us assume that the data we already acquire (the data produces by the accelerometer, or rather the data which is produces after processing the output of the accelerometer) is good information to discriminate the new activity. To train the classifier, one would simply need to register data using the *getData* plugin in a log file, add a new class number at the end of each line and add the new lines to the training set.
(Here is how to add a number at the end of each line using Vim macros :

```
ECHAP , qa , A, [enter your new number], ECHAP, j, q
```

How to use the macro :

```
[number of lines]@a
```

Once this is done, constants have to be modified in both files.

Modifying constants for Logistic Regression one must modify line 41−43 of file *logisticReg.m* :

```
%% ====== Setup of the parameters =======
input_layer_size  = 768;  % Number of features we will use for the regression
num_labels = 2;           % number of labels/classes
```

File *oneVsAll.m* must also be modified at line 20:

```
CLASS_NUMBER = 2;
```

Other files remain the same.

Modifying constants for Neural Networks:

3.6.2 Fitting the parameters:

There are parameters that can be tweaked if, after adding new classes , the classifier no longer performs well. The problem can come from a number of causes :

Irrelevant data:

It is possible that the data used is not relevent too discriminate the new activity, in which case new data should be added. Nothing needs to be modified in Octave scripts to add new data except the number of examples in the training set (see how to set up the parameters : 3.5.1 at page 29).

Over-fitting:

Your model might be suffering from over-fitting. As explained in 3.3 at page 22 this is easy to spot : compute an error on your training set, then an error on a set that was not used to train the model (test set) , and compare the two. If the second averaged error is higher, then your model suffers from overfitting. In that case :

- You can increase the size of the training set.
- You can increase the regularization parameter. Tyically you will want to try out different values of λ, plot both the training error and the test error against λ and choose the parameter that gives the best result. There is a subtelty there : the set must be divided into 3, a training set to fit the parameters of the regression, a set to determine which λ yields the best results, and a set to test the final result. Reusing one of the two first set to judge the model would be optimistic as the model will be biased towards those sets.
- Finally you could try simplifying the model. In the case of linear Regression the only option you have to do this is decreasing the number of features, and that must be done in the plugin that is acquiring the data. In the case of Neural Network, you can also decrease the number of features in the input layer, or you can try modifying the number of neurons on the hidden layer. In that case the best number of neurons can be determined with the same method as the best λ, with three sets.

The model may not be complex enough:

Your model may simply not be good enough. This is even easier to spot : the error is high even on the training set. In that case you must increase the complexity of the model. Decrease λ,increase the number of features or the number of neurons, and use the same methods to judge your progress (divide the set in three).

3.7 What could be done next

3.7.1 Adding classes

Our intention was that our classifier would detect many more activities that it now does, for instance driving a bicycle or a car. As we have explained, it is not hard to add a new class, but building a training set is long an we lacked the time to do it for additionnal classes. What we would therefore do if we were to continue this project is keep adding classes until the classifier showed its limit using the accelerometer then we would add new sources for data

3.7.2 Adding data sources

As explained in chapter 2 of this document, adding new sensors can be done quickly. We had implemented a plugin to get data from the Gyroscope, but we did not use it to train our classifier. Some sources would be preprocessed with techniques similar to those we used for the accelerometer (that would be the case for the gyroscope), but other very different sources would have to be preprocessed differently. Looking at what can be done with the GPS signal or the audio-recorder seems like interesting subjects.

3.7.3 Making the model more frugal

Our models work very well as is on classifying our activities : almost too well. Very big feature vectors are used (allthough we did create a "summerized" version of the vectors, which also performs well). Since smartphone power has to be saved, making the model as frugal as possible would be interesting.

Chapter 4

Conclusions about Aware

4.1 Judging the possibilities offered by AWARE

AWARE is a framework that is intended to simplify context acquisition, and we believe it is interesting for the following reasons :

- AWARE provides us with a usefull **level of abstraction** as concerns the sensors. That is to say that we don't have to use the *complicated* android classes to use sensors. The only thing we need to access data is running a query on the sensor database provided by AWARE. All constant keys to do so are listed on the AWARE website. (in Documentation/Sensors/). This saves a lot of time and effort allthough manipulating the sensors could be done from scratch.
- AWARE works on a **modular** principle. That is to say that every plugin may communicate with other plugins but also with other applications on the phone. They can *broadcast* data, context, information, everything they want to share with others. This means context can be refined by successive plugins that would be more and more specific. For instance, you could have a little plugin that determines if you are inside or outside, and once it has been assessed that you are indeed outdoors, an other little plugin that determines if you are walking or biking would be ran. Both of them communicate their findings (contexts) and these can in turn be used by other plugins.
- AWARE is an **open source** framework. That is to say that you might find a wide range of different plugins through the web. You can also be inspired by the code of other plugins in order to understand and create your own ones. You can easily find the author of a plugin by using GitHub and then ask them questions if you have any. It is open-source, but not under a constraining license (Apache Software License 2.0.) which allows commercial purposes.

There are also limitations brought by AWARE :
The most important thing to note is the fact that we develop *plugins*. In other words, we don't develop an application so we are not totally free with what we may want to do. For example, we don't have access to the *onCreate()* method of the application so we can't run a specific code when AWARE is starting. To sum up, we are forced to develop in the environment provided by AWARE, even if it doesn't suit our needs. Plugins can send Broadcasts however, so an application could be ran in parallel to an AWARE plugin, and listen to it's broadcast. That way the programmer is free to build his own app and still benefit from AWARE (But it means that both must be installed and running).

4.2 Recommandations

To sum up, we believe that AWARE's main interest is in saving time and effort : dealing with the sensors could be done from scratch but would require ressources, and AWARE does it reliably and is simple to use. If using other people's plugin is an option then this also saves efforts : no need to reinvent the wheel.

What we would do from here :

- Try to build an external application which uses data broadcasted by a plugin. Indeed AWARE is usefull to acquire context but not to use that data (it simply offers a "stream" wich is an interface that displays what we want it to display, but we do not find this particularily usefull.
- Try to build a stream of numerous plugins that use each other to refine a context efficiently.
- Work with other sensors, find out what can be done with them.

4.3 Good bye

We hope this report was sufficiently clear, detailed, and not too laborious. This project taught us much about Android, offered an opportunity to try out what we had learned about Machine learning, and to top it all we enjoyed working together a great deal. Thank you for reading !