

Categorizing and adjusting Data Quality Indicators for Android Motion and Environment Sensors

Project Report

By

Ankan Mookherjee

axm3244@g.rit.edu

Supervisor: **Dr Leon Reznik**

lrvcs@rit.edu

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology Rochester, New York

December 2014

Acknowledgment

I would like to take this opportunity to express my heartfelt gratitude to Professor Leon Reznik who gave me the chance to work on this project under his supervision. It would not have been possible to pursue this work without his continuous guidance, support and motivation. I would also like to thank Professor Justin Cappos and the team behind Sensibility Testbed app - which inspired me to develop my own app to carry out customized data collection and analysis. I would like to further thank all my project group mates for their valuable inputs and encouragement. My sincere thanks to my colleagues at High5Games who helped me in collecting data and supported me when I needed it the most. I would also like to thank my wife Shraddha for inspiring me to work hard and motivating me throughout. Finally, I would like to thank my parents and my sister Arunima for their constant support and guidance.

Abstract

Smartphones today have built in sensors including accelerometers, compasses, GPS and sensors for pressure, relative humidity, temperature and light. These sensors are used in a wide variety of applications ranging from weather, travel, interactive games, displaying relevant adverts based on user location and preferences. This project focuses on 2 test cases to identify ways to track and measure data quality of the information obtained from android smartphone sensors. In the first test case, we look at the android motion sensor that measures gravity, accelerometer and orientation of the device. While in the second case, we look at the weather sensor that has barometer (pressure) data and temperature data (finding the correlation between external temperature retrieved from weather API and battery temperature retrieved from battery sensors). I started with gathering data from different android devices and studying ways to explore alternatives to generate attributes for creating Data Quality (DQ) metric to measure the accuracy and quality of phone sensors. To track and measure data obtained from motion sensors, I primarily focused on finding out DQ metric to help determine calibration errors in the sensors. In case of weather sensors (barometer) I looked at comparing the real time data obtained from phones with data from weather API's. I believe that this particular application research can be very valuable in developing crowdsourcing applications to securely capture smartphone data without compromising user privacy. This can also act as a precursor to large-scale data mining to help interpret different trends of smartphone usage. Moreover, this study can also be very helpful in developing an android sensor diagnostic application to diagnose any possible physical damage to a device and monitor its condition. As part of this project, I have developed a distributed application that focuses on continuous integration of device data to cloud-based server, which helps in generating rapid and readjusted DQ indicators during data collection. This can also act as a continuous monitoring system for motion and environment sensors of android devices.

Table of Contents

1 Introduction	4
1.1 Problem	
1.2 Motivation	
1.3 Related Work	
2 Choosing Data Quality Metrics	8
2.1 Android Sensors	
2.2 Motion Sensors	
2.3 Accelerometer and Gravity Sensor	
2.4 Environment Sensors	
2.5 Open Weather API: Web Client for extracting weather information	
3 Methodology	13
3.1 High Level View	
3.2 Sensor Test App	
3.3 Bundling and Sending the Data to Azure Queue	
3.4 Azure SQL DB	
3.5 DQ Engine	
3.6 Device ID and Device Info	
3.7 Calculation of DQ Indicators	
3.8 MVC Reporting Dashboard	
4 Results	30
4.1 Quality Analysis with Device Type	
4.2 Sensors DQ with Device Model Types	
4.3 Gravity Sensor Quality Metric for Device Model	
4.4 Gravity vs. Accelerometer Sensor Quality Metric for Device Model	
4.5 Battery Temperature Sensor, External Temperature Quality Metric for Device Model	
4.6 Pressure Sensor Quality Metric for Device Model	
4.7 Score and Percentile for Device Mode	
4.8 Score and Percentile for Device Model - Complete Sensor Quality Data Table	
4.9 Device ID (IMEI) Vs. Sensor Data Quality Indicators	
4.10 Sensor Monitoring: Continuous Readjustment of DQ Indicators	
4.11 Key Results	
5 Future Work	40
6 Conclusions	41
7 References	42

1 Introduction

Today, most smartphones have built-in sensors such as accelerometers, compasses, GPS and sensors for temperature and light. These sensors are used in a wide variety of applications ranging from weather, travel, interactive games, displaying relevant adverts based on user location and preferences. The built-in sensors in these devices can measure various environmental conditions in addition to motion and orientation. These sensors provide raw data with high precision and accuracy, and can help in monitoring changes in the ambient environment near a device.

For instance, the weather application uses a device's accelerometer, pressure and temperature sensor to measure environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. In the same way, Accelerometer sensor allows us to determine the current orientation of a device, monitor and track changes in orientation, monitor gravity and acceleration. These sensors can also be used to track user information including GPS location.

If we record data from a sensor that is sending out continuous data - we are not only monitoring sensor diagnostics but also ensuring that the sensor data fluctuations are within known range. Further, we also ensure that the system carries out data cleansing and preparation while it collects data.

In my project, the data quality analysis specifically focuses on calculating accuracy for weather based and motion sensors. I have primarily looked at generating calibration errors for sensors as this data can be used as a tool to help diagnose the change in sensor information over a given period of time. Also, this particular data can help negate calibration errors and can be used to calculate accurate orientation data of the phone using sensor fusion.

I have sourced and collated data from different android devices including Samsung Galaxy S5, Galaxy S4, Galaxy S3, Note 3 and HTC. I have checked the Data Quality of the extracted sensor data and assigned scores, based on percentage of the magnitude of the specific tuple over difference between its minimum and maximum deviation points.

1.1 Problem

We can securely capture and share sensor data on Android devices using the collaborative sensor platform called the Sensibility Testbed Framework. However, we can't always guarantee the reliability and quality of the data captured. My project looks at carrying out Data Quality Analysis on the data captured from different devices, in order to verify its trustworthiness.

Data accuracy can be calculated by testing the security features of a sensor for malicious data in addition to carrying out checks for firewalls, detecting intrusions and finding out the gaps between patches and installation data.

In order to fulfil this objective I have created an android app, which securely captures sensor information and generates sensor data quality metric from the device data. I have done a continuous integration of the collected sensor data to a back end cron job. This would ensure readjustment of the DQ indicators and metric on an hourly scale with all the data collected from various devices. I have created a data quality metric to measure the accuracy of sensor data captured in order to categorize the device's data into levels of trustworthiness. This process has enabled me to further compare the results of the data to carry out a continuous re-adjustment of the data quality.

1.2 Motivation

Smartphones produce immensely valuable data for critical scientific pursuits across a wide variety of disciplines. Data from environment sensors including Barometer, Relative Humidity and Ambient air temperature, if securely collected can be used to create meteorological data models which have higher precision than the open weather API's currently available. This is because the information from these sensors tends to focus on major area zone and has a time lag instead of real time updates that one can get from smart phones. If properly harnessed and leveraged the information sensing capabilities of android devices can bring us substantial benefits.

Also, Motion sensor information can be subjected to calibration error, as phones are prone to falling down or breaking, which can result in incorrect sensor feeds. In addition to calculating the frame of reference of a device, motion sensors also relay a lot of other valuable information which can be very useful to develop gaming applications such as Temple Run and Google Navigation.

Further, this project looks at the captured sensor data from a privacy and security point of view.
^{[20][15][16]} It has been reported that some apps tend to record data without prior permission of the smartphone user. This study can further lead to detecting and preventing hardware Trojan ^[21] by mapping the quality of motion sensor data. It can also give us a better idea about the ways in which we can introduce a detectability metric to analyze the real strengths and weaknesses of smartphone sensors. All of which will be very helpful in verifying the trustworthiness of a device before collecting data for research or commercial purposes.

1.3 Related Work

This project combines two key areas of study. The first one relates to studying and collecting android smartphone sensor data and the second one relates to Data Quality evaluation. The overall project deals with Data Quality evaluations carried out on sensor data available from android smartphone devices. In his research papers,^{[1][2][3][4]} Dr Reznik has defined several new approaches to integrate data quality indicators available from a various studies to measure accuracy, safety and security engineering, which can act a facilitator in building better and secure products. My work is based on some of the outlined approaches for data quality evaluation for android sensor data.

Currently there are many applications in the Google Play app store such as Phone Tester^[9], Sensor List^[10] and Sensibility Testbed^{[5][6][7][8]}, which provide sensor information for the phone. Sensibility Testbed application is a sensor Testbed application which captures sensor data in addition to allowing a remote researcher to access the device sensor data using a client side code. This data is always accessed with user's permission and is only meant for research purposes. Further, linking an install build with an encrypted public key and private key ensures the confidentiality of the data access. These keys are generated on a one-time basis, which is, completely unique to a new installation build. Therefore, a researcher can only access a build with a key pair if the people installing the app on their phones have agreed to share their phone sensor data with that particular build.

This framework also allows us to share this sensor data on the collaborative platform with other researchers. This project is a part of a major collaborative effort that involves Data collection and Evaluation on the collaborative sensors platform. My project is making use of the Sensibility Testbed framework that allows secure capturing and sharing of sensor data. There are several modules included in this open source project. In this project, I have implemented the following stages^[1] for Data Quality (DQ) evaluation:

1. Assignment of initial Data Quality Metric
2. Calculation of Data Quality indicator
3. Performing self-adjustment of the DQ indicators using collaborative sensors group

2 Choosing Attributes for Data Quality Metrics

2.1 Android Sensors

Sensors can be broadly divided into 3 categories ^[9]:

- 1) Motion Sensors (accelerometer, gravity, gyroscope, linear accelerometer and rotation vector)
- 2) Position Sensors (orientation, geomagnetic field and proximity)
- 3) Environment Sensors (light, pressure, humidity and temperature)

2.2 Motion Sensors

The motion sensor information can help determine if the device is stable or in motion at a given time at which the accelerometer reading has been taken. In order to avoid incorrect gravitational sensor reading (generated by sideways oscillation), it is important to ensure that the device is stable/still before taking the accelerometer reading. In order to catch the error we need to get a g reading, which is consistent around 9.8 m/s^2 .

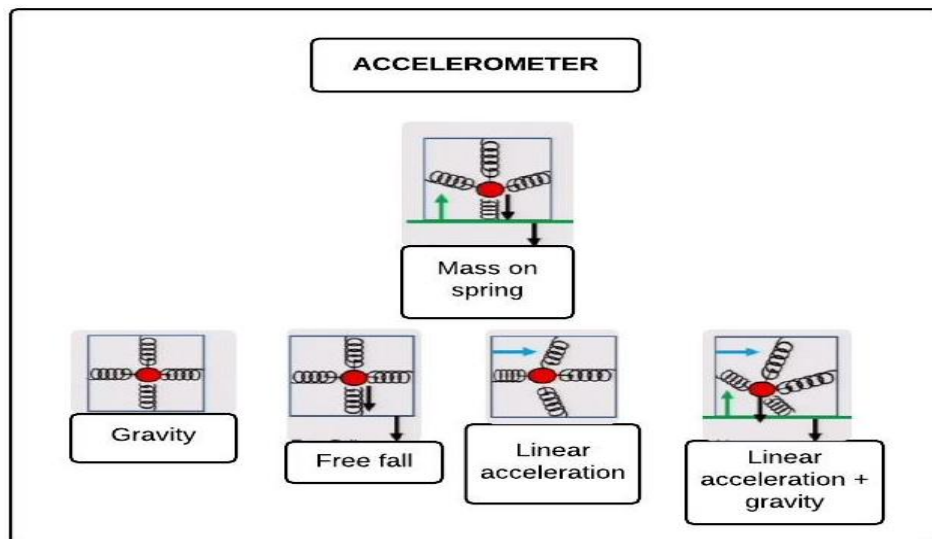


Fig 2.0 Gravity and Accelerometer Sensor

This diagram shows how accelerometers work in general and how they calculate the reading. It is important to note that the accelerometers used in android devices are not mechanical in nature (unlike the one shown in the diagram above) and are primarily electronic and magnetic based. However, the basic logic behind the data capturing technique still remains the same.

If the system is still/stationary, we would get an acceleration according to this formula:

$$g = (x^2 + y^2 + z^2)^{0.5} \approx \mathbf{9.8 \text{ m/s}^2}$$

While, for free fall:

$$g = 0$$

However the acceleration slightly differs for different latitudes^[22]

$$^{[22]} gn = 978031.85 (1.0 + 0.005278895 \sin^2(lat) + 0.000023462 \sin^4(lat)) (mgal)$$

Where *lat* is latitude in degrees

Taking the derivative w.r.t. *latitude* and converting to (m) from (rad) gives a correction of:

$$\Delta g_l = 0.000812 * \sin(2 * lat) \text{ mgal/m (N-S)}$$

Linear acceleration applies when the system is in a state of motion. This system when combined with gravity is unstable and can make it very difficult to spot the difference between correct and incorrect reading. In this particular case, we have to treat this data as an outlier.

In order to ensure that we separate the outlier data for Accelerometer, I added *sensor fusion*^{[23][13][14]}, which would combine data from gyroscope and compass sensors to give us an idea about the relative stability of the system at a given time at which the accelerometer data is collected.

2.3 Accelerometer and Gravity Sensor

Metadata includes: Gyroscope, Compass Data, Geo Location and Time-Stamp

Gravity sensor can use Gyroscope data to determine whether or not the device was still/stable at the time at which the data was recorded. However, in order to ensure that the sensor readings are accurate and include only the gravity sensor data without any vibrations or oscillations - we need additional data from compass and gyroscope. A steady gyroscope reading and change in compass data over a small fraction of time can help determine the correct gravity reading. Gyroscope and compass data can also help us calculate the phone orientation at the time at which the accelerometer reading is taken. Geolocation and timestamp information is also key in finding the actual direction of magnetic field from the test location.

2.4 Environment Sensors

Android smartphones have several environment sensors such as Barometric Sensor, Temperature Sensor, Relative Humidity Sensor, Light Sensor. These sensors collect real time ambient data. Due to the newly introduced waterproofing feature in android phones, some of the devices such as Samsung Galaxy S5 don't have relative humidity sensor anymore. However, I have collected the data for relative humidity on the backend for future reference. Light and proximity sensors are difficult to compare and measure for DQ metric analysis on crowd-sourced app.

- **Barometer Sensor**

Metadata includes: Geo Location, Time-Stamp, Ambient Barometric Pressure

After carefully visualizing all environment sensor data, I chose **(pressure/barometer)** sensor for this project. I integrated the system with a web API to compare the data received from both sources.

- **Battery Temperature Sensor**

Metadata: Geolocation, Time-Stamp, Ambient Air Temperature

I wanted to include **temperature** into my DQ metric calculations and while carrying out the literature review I came across several articles ^{[25][24]}, which discussed ways in which crowdsourcing battery temperature can be used to calculate outside ambient temperature. Although battery temperature depends on several factors including the phone's location (indoor or outdoor) or if it is in one's pocket or charging – However, including the noise there is some kind of correlation to ambient air temperature. This inspired me to collect the battery data which can be harnessed to create another external data quality metric.

2.5 OPEN WEATHER API: Web Client for extracting weather information

The API that I selected was **openweathermap**^[26] as it is convenient to parse and it has access to historic data. Using this API, the data collected in a previous timestamp can be also compared.

The API Call that I used to send parameterized input of Latitude (lat) and Longitude (lon) looked like this:

HTTP REQUEST: <http://api.openweathermap.org/data/2.5/weather?lat=40.73&lon=-74.08>

HTTP RESPONSE: 200 OK

JAVASCRIPT OBJECT NOTATION (JSON) OUTPUT:

```
{ "coord": { "lon": -73.08, "lat": 39.73 }, "sys": { "message": 0.0243, "country": "US", "sunrise": 1418299417, "sunset": 1418333285 }, "weather": [ { "id": 803, "main": "Clouds", "description": "broken clouds", "icon": "04n" } ], "base": "cmc stations", "main": { "temp": 278.975, "temp_min": 278.975, "temp_max": 278.975, "pressure": 1019.44, "sea_level": 1019.45, "grnd_level": 1019.44, "humidity": 82 }, "wind": { "speed": 12.42, "deg": 286.505 }, "clouds": { "all": 76 }, "dt": 1418261529, "id": 4504014, "name": "Seaside Park", "cod": 200 }
```

Parsing out the useful information

Temperature: 278.975 Kelvin = 5.825 Celsius = 42.485 Fahrenheit

Pressure: 1019.44 hPa (Hexa Pascal)

Humidity: 82%

```

var client = new RestClient("http://api.openweathermap.org/data/2.5/weather");
var request = new RestRequest ("",Method.GET);
request.AddParameter("lat", _currentLocation.Latitude);
request.AddParameter("lon", _currentLocation.Longitude);

var response = client.Execute(request);

RestSharp.Deserializers.JsonDeserializer deserial= new JsonDeserializer();
var JSONObj = deserial.Deserialize<Dictionary<string,string>>(response);

string main = JSONObj["main"] ;

// this how main looks like: {"temp":293.61,"pressure":1019,"humidity":17,"temp_min":291.15,"temp_max":295.15}
string[] splitSrt = main.Split(',');
string[] pStr = splitSrt[1].Split(':');

string[] ptemp = splitSrt[0].Split(':');

gsData.pressure_service = Convert.ToDouble( pStr[1] );
double temp = Convert.ToDouble(ptemp[1]);

//kelvin to celcius
gsData.temperature = temp - 273.15;

```

Fig 2.2 Code to collect Weather Data from a location

3 Methodology

3.1 High Level View

The Sensor Test app records android sensor data from various devices. It uses the geo-location sensor to put a REST request to open weather API in order to get the weather information for a particular location. The complete information including information from sensors and weather API is added with timestamp in a bundle and sent to an Azure Queue, which is a secure endpoint on cloud. This Azure Queue then polls the data and inserts it in Azure SQL database. There is DQ Engine, which is a cron job (a continuously running web job) on cloud, which extracts DQ indicators from the database and puts the newly processed data into another table within the database. This web job is set at an hourly pace. Finally, there is MVC view, which shows us a UI of DQ Metrics generated.

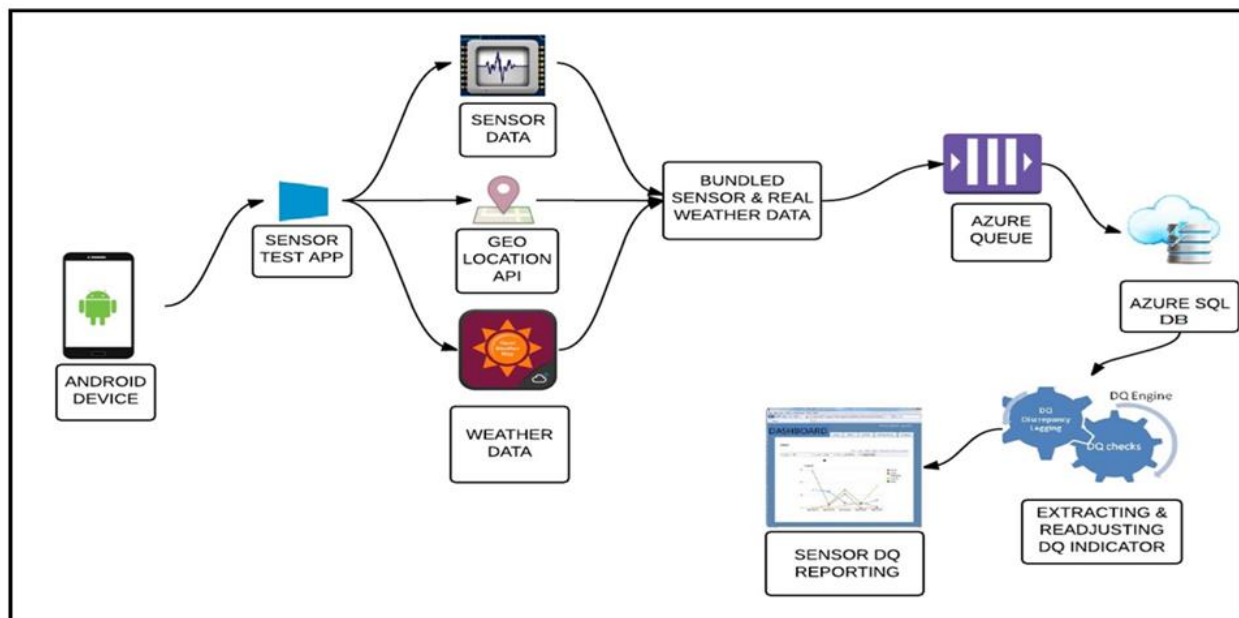


Fig 3.0 High Level Architecture Diagram

The next section describes the following key areas:

- 1) Building the App
- 2) Building the data repository for data collected from different smartphones
- 3) Generating DQ indicators from the data
- 4) Readjusting DQ indicators based on the relative error to the new data
- 5) MVC razor for GUI reporting of DQ indicators

After working on the Sensibility Testbed project for some time, I realized that it would be best if I created my own system and worked on it. Therefore, I created my own app to collect the information needed for this project. In addition to this, I also created a cloud-based repository to collate and store the data collected. This repository also housed a web form application, which performed readjustment of the DQ indicators.

3.2 Sensor Test App

My Application is **Sensor-Test** (version 5.0). It collects motion sensor, GPS, environment sensor, battery and telephony data.

App Code:

<https://github.com/Ankanmook/Sensor-Test/tree/master/Sensor-Test-Client-master>

My app is created using Xamarin Studio ^{[27][28][29]} which is a C# based android as well as iOS application development framework. I rolled out Version 5.0 of this app after testing it on multiple android devices and fixing bugs. As of now, I have made provisions within the app to manually collect data using a record button – However, automatic collection of sensor data is also feasible. This app runs on Honeycomb and higher android version (Android API 11+, Version 3.0 and higher).

My Sensor-Test app collects Accelerometer, Gravity, Gyroscope, Magnetic Field, Orientation, Barometer, Relative Humidity, and Ambient Temperature sensor data and sends it to the cloud DB. This device also makes calls to openweathermap API and attaches Weather data along with the sensor data every time data is recorded.

Major Functionalities of the app include:

- 1) **Data Filter:** To ensure that data is captured only when the phone is in a stationary position.

In my code I have a class **GravitySensorData.cs** that performs the function of filtering the noise and data obtained from the sensor. For this purpose, it ensures that the device is placed on a stable surface. The raw data is not ready to use until it is calibrated. In order to calibrate the data, scale and bias must be taken into account. The bias represents how far the centre of data is from the zero, while the scale reflects how large the range of data from the sensor is as compared to real meaningful data.

The data filter collects a lot of data and processes it, to allow storage of only that part of the data, which has been collected when the device has been continuously stable for a certain number of instances. So far, I have tried to record and filter queues of 10, 20 and 30 instances. However after experimenting, I decided to use 20 instances to be filtered for my final value, to be checked and processed for accuracy and sent back to the backend.

```
/*
 * Filtering noise from barometer and pressure data
 */
public void calculateBarometerGravityData(){

    //Values are stable
    if ( ( (xGMax - xGMin) <= 0.05) && ((yGMax - yGMin) <= 0.05)
        && ((zGMax - zGMin) <= 0.05)) && ((pMax - pMin) <= 0.05)
        && (xList.Count >= arrSize) && (yList.Count >= arrSize) && (zList.Count >= arrSize) && (pressure.Count >= arrSize) ) {

        //Will stop counting
        recount = false;

        //Record the gravity data
        //Record the pressure data
        getGravityPressure ();

    } else {
        //Will start counting again
        recount = true;
    }
}
```

Code: Filter used to ensure that the device is stable in Queue

- 2) **Kalman Filter**^[13]: To reduce the white noise. Noise generated from gyroscope sensor is of exceedingly high order, which is caused as a result of drift in rotational angle. Gyroscope uses Coriolis acceleration effect^[13] on a vibrating mass to detect angular rotation. The measure angular velocity by gyroscope is linear which is proportional to the rate of rotation. The gyroscope sensor result is computed on a real time basis but the drift over time alters really quickly. For example in a stable system after roughly 30 second's drifts slows down by nearly 50 degrees. The reasoning behind this lies in the fact that the accumulated integration increases the noise over the time. This integration of disturbance is converted into the drift which leads to erroneous (noisy) results. Moreover, the integration result is less noisy as compared to the gyroscope sensor but tends to have more drift.

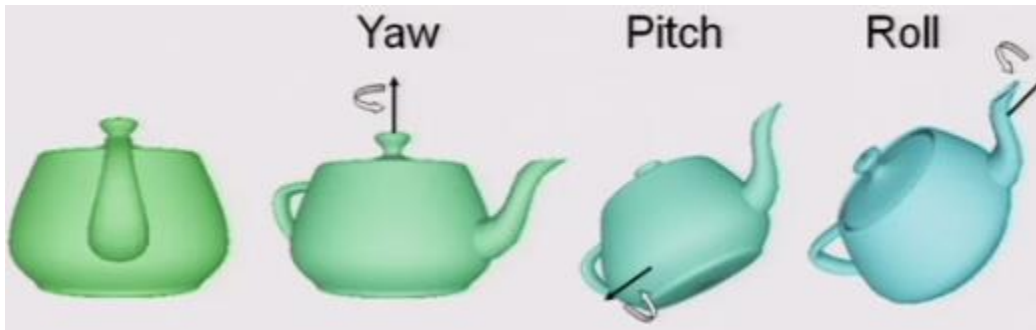


Fig 3.1 Euler Angles (Orientation Sensor Data)

One of the key advantages of the gyroscope is that earth's gravity has not effect on its reading. However Accelerometers measure acceleration from linear movement in two dimensions combing with gravity of the earth. Since Accelerometers are unable to separate these two accelerations, for measuring orientation filtering out gravity is redundant. Filtering has an added cost (time) which can make the response slow and that's why gyroscopes mostly use accelerometers in order to provide drift. The accelerometer output can help calculate the rotation around the X- axis (roll) and that around the Y-axis (pitch). If X_accelerometer, Y_accelerometer, and Z_accelerometer are accelerometer measurements in the X, Y and Z axes respectively, then we use that to calculate the pitch and roll angles (Euler Angle).

Neither Accelerometer nor Gyroscope can provide the most accurate rotation measurement on its own. Therefore, we implement the sensor fusion algorithm^[13] to

compensate for the weakness of accelerometer and gyroscope sensors by combining them together.

First I read the roll and pitch from accelerometer and then combined them to gyroscope sensor reading through a Kalman filter to acquire a clean non-drift roll and pitch angle. Further, I applied tilt compensation^[16], which uses magnetometer in combination to roll, and pitch to calculate challenging yaw rotation. This is the final output obtained using the most reduced noise.

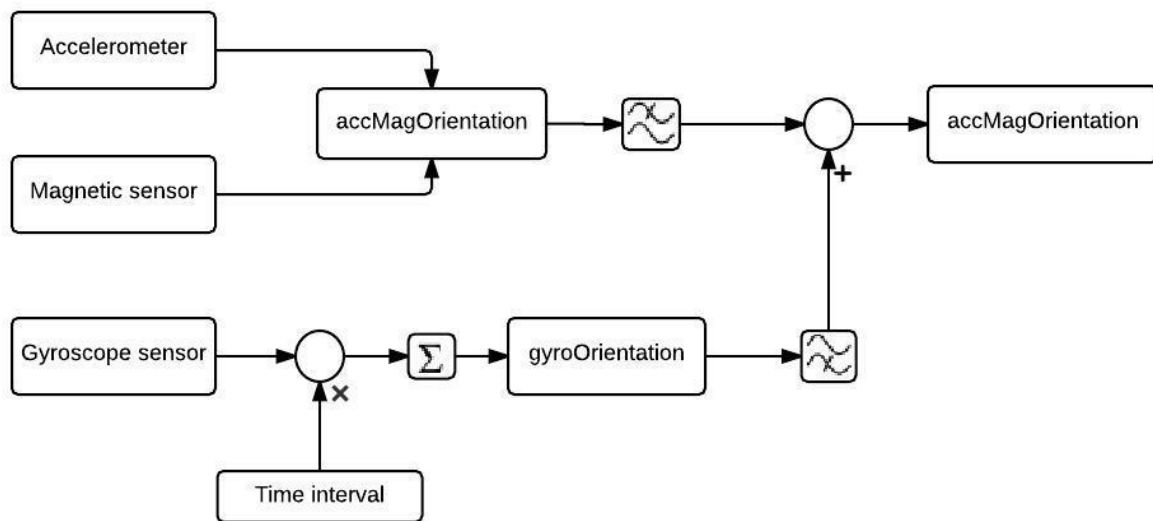


Fig 3.2 Sensor Fusion: Use of Kalman filters to remove the noise from the Accelerometer and Gravity Sensor Data

- 3) **Only sending complete data:** In case of GPS information (location) or information from weather API being unavailable, this functionality acts as a safeguard that stops from sending the data back. This also ensures that the data sent is clean and can be easily used to generate metric or carry out analysis.
- 4) **Generating notifications:** when successful batches of data are sent as well as when there is a case of single failure in sending data.

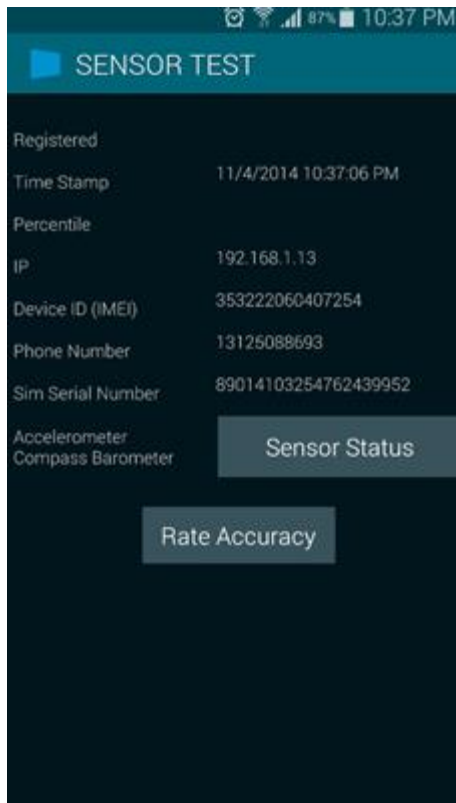


Fig 3.3 Main Screen

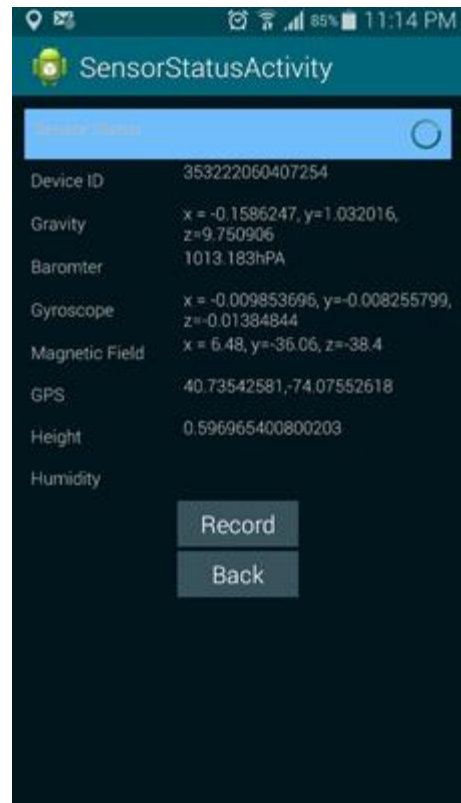


Fig 3.4 Record Data Screen

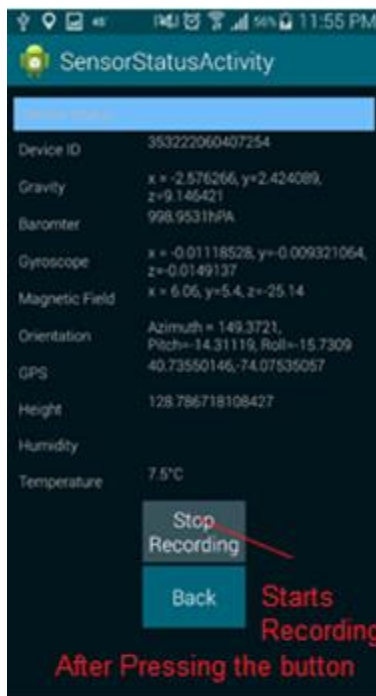


Fig 3.5 Manual Stop Recording

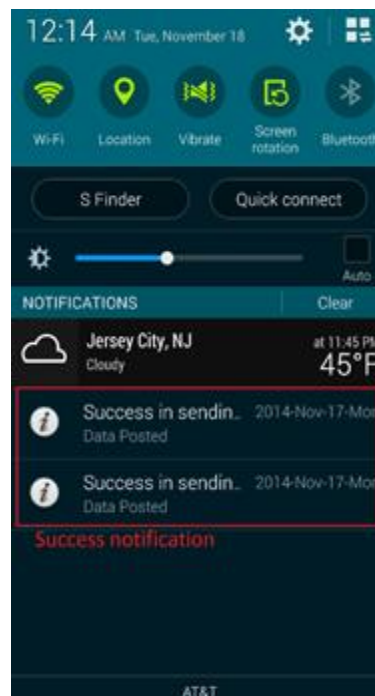


Fig 3.6 Success Notification

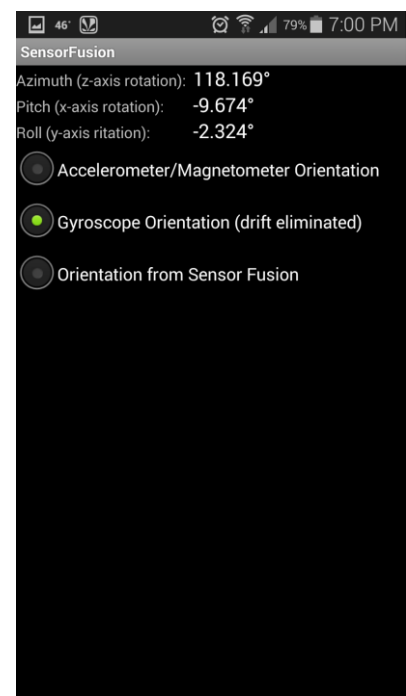


Fig 3.7 Sensor Fusion

3.3 Bundling and sending the data to the Azure Queue

After the data is collected and filtered for one instance, it is bundled and sent to Azure Queue, Which puts the data into Azure Sql Database.

```
[Java.Interop.Export()]
public async void sendData(){

try{

    sensorTable = client.GetTable<DeviceData>();

    // Create json request to be sent to azure web service
    var item = new DeviceData() {
        complete= true, deviceid = device_Id, deviceinfo = device_Info, latitude = _currentLocation.Latitude,
        longitude = _currentLocation.Longitude, altitude = _currentLocation.Altitude,
        gravity_cumulative = gsData.getGravity(), gravity_x = gsData.xComponent, gravity_y = gsData.yComponent,
        gravity_z = gsData.zComponent, error_gravity = gsData.error_gravity,
        accelerometer_x = gsData.accelerometer_x , accelerometer_y = gsData.accelerometer_y , accelerometer_z = gsData.accelerometer_z,
        pressure_device = gsData.pressureComponent, pressure_service = gsData.pressure_service , error_pressure = gsData.error_pressure,
        magneticfeild_x = gsData.magneticfeild_x , magneticfeild_y = gsData.magneticfeild_y , magneticfeild_z= gsData.magneticfeild_z,
        gyroscope_x = gsData.gyroscope_x , gyroscope_y = gsData.gyroscope_y , gyroscope_z = gsData.gyroscope_z,
        height = gsData.height, temperature = gsData.temperature, azimuth_angle = gsData.azimuth_angle, roll_angle = gsData.roll_angle, pitch_angle = gsData.pitch_angle,
        battery_status = gsData.battery_status , battery_level = gsData.battery_level , battery_voltage = gsData.battery_voltage ,
        battery_temperature = gsData.battery_temperature , battery_technology = gsData.battery_technology ,
        timestamp = DateTime.Now
    };

    //sensorTable = client.GetTable("DeviceData",sendData);
    // Insert the new item
    await sensorTable.InsertAsync(item);

    if (item.complete){
        sensorList.Add(item);
        //NotifyDataSetChanged();
        //adapter.Add(item);
    }
}
```

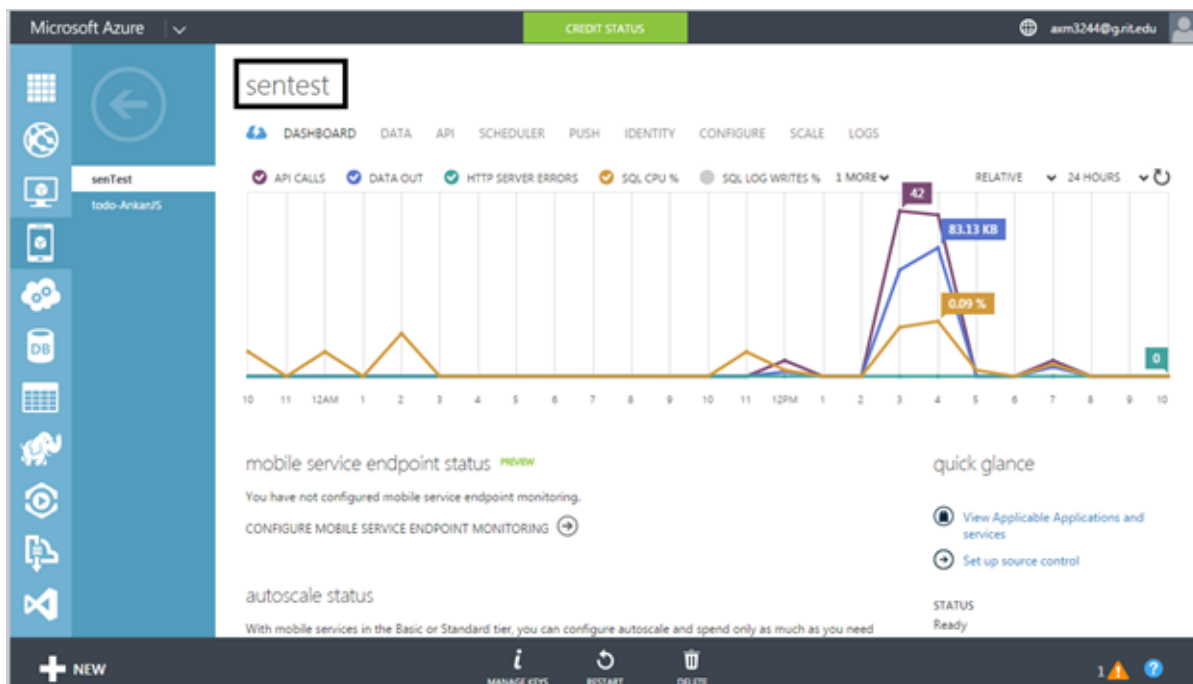
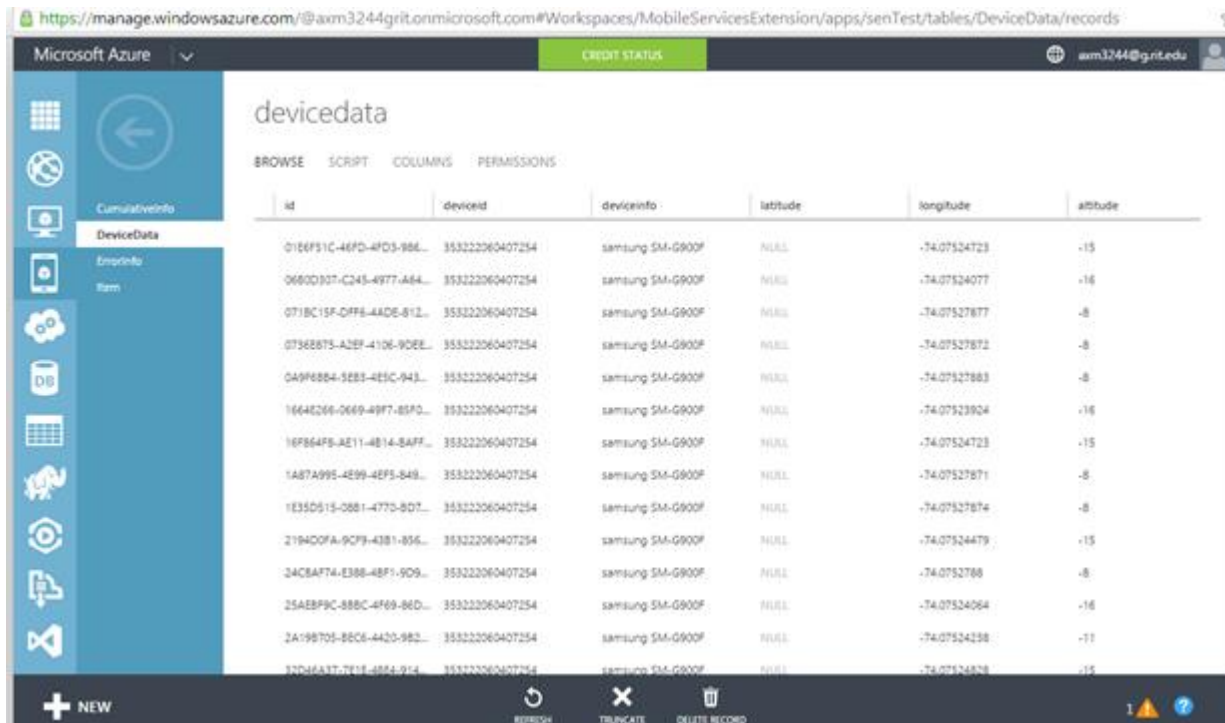


Fig 3.8 Web service calls from Device to Azure Cloud and Device Traffic

3.4 Azure SQL DB

Azure SQL Database has 2 tables in **SenTest** Database. First one is **DeviceData** Table, which stores the raw device data. Second one is **DQIndicator** where calculations of the recent DQ indicators are inserted and updated from time to time.



Microsoft Azure

devicedata

BROWSE SCRIPT COLUMNS PERMISSIONS

id	deviceid	deviceinfo	latitude	longitude	altitude
01E6F51C-46FD-4FD3-986...	353222060407254	samsung SM-G900F	NULL	-74.07524723	-15
0660D307-C245-4977-A64...	353222060407254	samsung SM-G900F	NULL	-74.07524077	-16
0718C15F-0FF8-44D6-812...	353222060407254	samsung SM-G900F	NULL	-74.07527877	-8
07362875-A2E9-41D6-9D6E...	353222060407254	samsung SM-G900F	NULL	-74.07527872	-8
049F6864-5E83-4E3C-943...	353222060407254	samsung SM-G900F	NULL	-74.07527883	-8
1644E266-0669-49F7-85F0...	353222060407254	samsung SM-G900F	NULL	-74.07523924	-16
16F864F8-AE11-4B14-8AFF...	353222060407254	samsung SM-G900F	NULL	-74.07524723	-15
1A87A995-4E99-4E95-849...	353222060407254	samsung SM-G900F	NULL	-74.07527871	-8
1E35D615-08B1-477D-8D7...	353222060407254	samsung SM-G900F	NULL	-74.07527874	-8
2194C0FA-9CF9-43B1-856...	353222060407254	samsung SM-G900F	NULL	-74.07524479	-15
24C8A774-E3B8-48F1-9D9...	353222060407254	samsung SM-G900F	NULL	-74.0752788	-8
25AEBF9C-888C-4F69-8ED...	353222060407254	samsung SM-G900F	NULL	-74.07524064	-16
2A1987D5-86C6-442D-982...	353222060407254	samsung SM-G900F	NULL	-74.07524298	-11
37D6A837-7E18-4854-914...	353222060407254	samsung SM-G900F	NULL	-74.07524878	-15

+ NEW

REFRESH TRUNCATE DELETE RECORD

Fig 3.9 SQL Azure Table Recording Sensor Data



Fig 3.10 Data Model: DeviceData and DQIndicator Tables in Cloud Database

3.5 DQ Engine

Code:

<https://github.com/Ankanmook/Sensor-Test/tree/master/DQEngineFinal>

As the data was collecting in the DeviceData table from different android smartphone clients, I built a synchronous web job, which ran at an hourly interval, parsing out data from DeviceData table and updating calculated Data Quality indicators for devices already present and inserting new Data Quality indicators for new devices. This allowed the newly generated data quality indicators to be continuously readjusted as fresh data was added into the device data table.

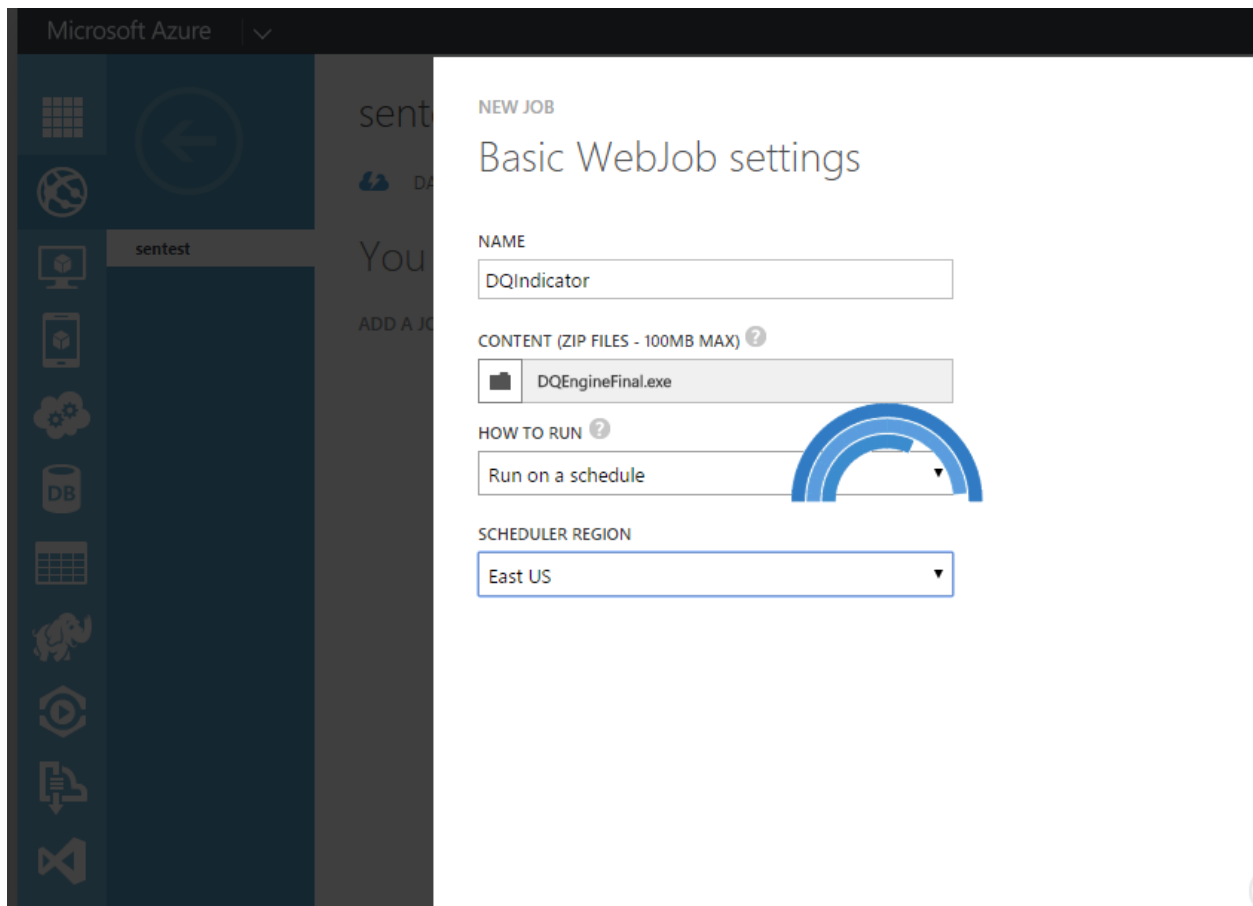


Fig 3.11 Scheduling DQ Indicator Web Job in Azure

3.6 Device ID and Device Info

In order to understand the sampling and calculation of DQ indicators from the DQ Metric, it's important to understand the difference between device id and device info. Device info is the build model, while the device id is the unique IMEI number. Therefore, as I had 4 Verizon Samsung Galaxy S4 devices, they will have separate device id but same device info. For example my phone Samsung Galaxy S5 has the device info Samsung G900F and device ID is 353222060407254.

```
Device_id = telephonyManager.Deviceld.ToString ();
```

```
Device_Info = Build.Manufacturer.ToString ()+ " " + Build.Model.ToString ();
```

Out of the 11 android phones that I have used to collect data, I can primarily categorized them into 7 main categories: Samsung GTI9300 (AT&T Galaxy S3), Samsung GT I9500 (AT&T Galaxy S4), ATT Samsung GT I9505 (AT&T Galaxy S4), Samsung SCH-I535 (Verizon Galaxy S3), Samsung SCH-I545 (Verizon S4), Samsung SCH-I605 (Galaxy Note4) and Samsung G900F (Galaxy S5). My major analysis deals with the device's behaviour on my DQ Metric parameters, which we will find out in the result section of this report.

3.7 Calculation of DQ Indicators

Using the sensor data from a device I chose the following metrics:

- 1) Gravity sensor vs. Gravity at that latitude
- 2) Accelerometer sensor vs. Gravity sensor
- 3) Orientation sensor vs. orientation filtered from sensor fusion
- 4) Barometric Sensor vs. Ambient Air Pressure
- 5) Battery Temperature vs. Ambient Temperature

Firstly, all the information from the **DeviceData** table is fetched using Entity framework and LINQ (language Integrated query) in the **initializeContext** method.


```

public void initializeContext()
{
    double actualGravity;
    //This will initialize data context
    using (context = new SensorTest_dbEntities())
    {
        var tuple1 = from c in context.DeviceDatas
                     select c;

        //Loading the complete list
        foreach (var d in tuple1)
        {
            idListDeviceData.Add(d.id);
            deviceIdList.Add(d.deviceid);
            deviceinfoList.Add(d.deviceinfo);

            magneticField_XList.Add((double)d.magneticfeild_x);
            magneticField_YList.Add((double)d.magneticfeild_y);
            magneticField_ZList.Add((double)d.magneticfeild_z);
            gyroscope_XList.Add((double)d.gyroscope_x);
            gyroscope_YList.Add((double)d.gyroscope_y);
            gyroscope_ZList.Add((double)d.gyroscope_z);

            actualGravity = getActualGravity((double)d.latitutde);
            actualGravity_List.Add(actualGravity);

            double sensorgravity = gravityCalculation((double)d.gravity_x, (double)d.gravity_y, (double)d.gravity_z);
            double accelerometerGravity = gravityCalculation((double)d.accelerometer_x, (double)d.accelerometer_y, (double)d.accelerometer_z);
            tempList.Add((double)(d.battery_temperature - d.temperature));
            accelerometer_eList.Add(sensorgravity - accelerometerGravity);
            pressureDiff.Add((double)(d.error_pressure));
            gravity_eList.Add(sensorgravity - actualGravity);
        }
    }
}

```

Using all the data collected in DeviceData table, the DQ Engine code loads all the data from into collections for a particular device id or device info. For example, within one collection you have one kind of data such as public List<Double> accelerometer_eList, which records the difference between accelerometer and gravity data for that tuple. Further, we will look at the inner working of DQ indicator calculation.

```

using (var context = new SensorTest_dbEntities())
{
    var tuple1 = from dd in context.DeviceDatas
                 select dd;

    foreach (var d in tuple1.ToList())
    {
        DQIndicator dqindicator = new DQIndicator();
        double actualG = getActualGravity((double)d.latitutde);
        double sensorgravity = gravityCalculation((double)d.gravity_x, (double)d.gravity_y, (double)d.gravity_z);
        double accelerometerGravity = gravityCalculation((double)d.accelerometer_x, (double)d.accelerometer_y, (double)d.accelerometer_z);

        dqindicator.id = d.id;
        dqindicator.deviceid = d.deviceid;
        dqindicator.deviceinfo = d.deviceinfo;

        dqindicator.gravity = getActualGravity(actualG);
        dqindicator.gravity_e = sensorgravity - actualG;
        dqindicator.percentile_g = Percentiler.percentile(gravity_eList, (double)dqindicator.gravity_e);
        dqindicator.score_g = Percentiler.giveRelativeError(gravity_eList, (double)dqindicator.gravity_e);

        dqindicator.percentile_accg = Percentiler.percentile(accelerometer_eList, (double)(sensorgravity - accelerometerGravity));
        dqindicator.accel_gravity_e = sensorgravity - accelerometerGravity;
        dqindicator.score_ga = Percentiler.giveRelativeError(accelerometer_eList, (double)(sensorgravity - accelerometerGravity));

        dqindicator.percentile_p = Percentiler.percentile(pressureDiff, (double)(d.error_pressure));
        dqindicator.score_p = Percentiler.giveRelativeError(pressureDiff, (double)(d.error_pressure));
    }
}

```

My ultimate aim was to calculate the DQ indicator for one device (unique device id) on a continuous basis while sensor information kept entering the cloud database. I calculated 2 DQ indicators for an attribute, a **Percentile** and a **Score** for average data collected. For instance, I calculated the difference between gravity data and integrated gravity sensor record from x, y, z coordinate. Further I used $g = (x^2 + y^2 + z^2)^{0.5}$ formula to calculate gravity and the formula mention for calculating gravity at the latitude ^[18] to calculate the two gravity information and measure the difference between them.

```
4 references
public double gravityCalculation(double x, double y, double z)
{
    return Math.Sqrt(Math.Pow(x, 2.0) + Math.Pow(y, 2.0) + Math.Pow(z, 2.0));
}

/*
 * Calculating gravity at exact latitude from this formula
 * For ref: http://geophysics.ou.edu/solid\_earth/notes/potential/igf.htm
 */
3 references
public double getActualGravity(double latitude)
{
    return 9.7803267714 * (1 + 0.00193185138639 * Math.Pow(Math.Sin(latitude), 2.0))
        / (Math.Sqrt(1 - 0.00669437999013 * Math.Pow(Math.Sin(latitude), 2.0)));
}
```

I then inserted the difference into a field **gravity_e**, which is a variable of **dqindicator** object, which calculates entire DQIndicator for one tuple of DeviceData.

```
double actualG = getActualGravity((double)d.latitutde);
dqindicator.gravity = getActualGravity(actualG);
dqindicator.gravity_e = sensorgravity - actualG;
dqindicator.percentile_g = Percentiler.percentile(gravity_eList, (double)dqindicator.gravity_e);
dqindicator.score_g = Percentiler.giveRelativeError(gravity_eList, (double)dqindicator.gravity_e);
```

This information is added to a collection of dqindicators

```
public List<Double> accelerometer_eList;
public List<double> gravity_eList;
public List<double> tempList;
public List<double> pressureDiff;
public List<double> orientationDiff;
```

Further, I calculated the percentile and score of this gravity_e in the entire row of gravity_eList

```
/*
 * This method calculates the percentile for a sequence
 * and the value you wish to calculate the percentile for
 */
6 references
public static double percentile(List<double> sequence, double excelPercentile)
{
    int n = findRankofElement(sequence, excelPercentile);

    return ((double)n * 100 / sequence.Count);
}
```

```
1 reference
public static int findRankofElement(List<double> sequence, double excelPercentile)
{
    int index = 0;
    sequence.Sort();

    while (index < sequence.Count)
    {
        if (sequence[index] == excelPercentile)
        {
            return index;
        }
        else
        {
            index++;
        }
    }

    return 0;
}
```

Score Calculation occurs by using the specific magnitude of **gravity_e** and calculating the **score = value - min / (max - min)**

```

5 references
public static double giveRelativeError(List<double> sequence, double value)
{
    return ((value - sequence.Min()) / getDistance(sequence)) * 100;
}

2 references
public static double getDistance(List<double> sequence)
{
    return Math.Abs(sequence.Max() - sequence.Min());
}

```

Each percentile and score is then added into cumulative score and percentile, which is further divided by the number of dimensions present - as shown in the code below:

```

dqindicator.cumulative_percentile = (double) ((dqindicator.percentile_accg + dqindicator.percentile_g +
    dqindicator.percentile_p + dqindicator.percentile_t + dqindicator.percentile_o) / 5;
dqindicator.score = (double)((dqindicator.score_g + dqindicator.score_p + dqindicator.score_ga
    + dqindicator.score_t + dqindicator.score_f)) / 5;

```

The final metric average of all the scores and percentile:

```

/*
 * Computes the average of a sequence
 */
10 references
public double getAvg(List<double> sequence)
{
    return sequence.Average();
}

```

However the cumulative score is calculated using 2 types of groups: by calculating individual device id (unique IMEI number) and by calculating unique model (device info). This information is then stored into DQIndicator table.

```

1 reference
public void getDeviceInfo(String deviceInformation)
{
    //This will initialize data context for a model
    using (context = new SensorTest_dbEntities())
    {
        var tuple1 = from c in context.DQIndicators
                     where c.deviceinfo == deviceInformation
                     select c;
    }
}

```

Average Calculated by Device Info (Model Number)

```

1 reference
public void getDeviceInfo(String deviceId)
{
    //This will initialize data context for a device (unique IMEI number)
    using (context = new SensorTest_dbEntities())
    {
        var tuple1 = from c in context.DQIndicators
                     where c.deviceid == deviceId
                     select c;
    }
}

```

Average Calculated by Device Id (IMEI number)

We can then leverage this information to publish the average score and percentile of a device within different categories. Further, we compare this result with that of all the devices in addition to comparing within same devices – such as best performing individual device out of all the Samsung Galaxy S3s used. We can also publish information and rating for an individual device, best performing device, worst device, and median information processing.

3.8 MVC Reporting Dashboard

Code:

<https://github.com/Ankanmook/Sensor-Test/tree/master/MvcIntensityChart>

I have built a dashboard, which displayed cumulative data and cumulative data quality metric evaluation for an individual device (unique IMEI number). This houses an in-house HTML5 view where users can the device information for their device id. The line graph shows the change in the instantaneous DQ indicators received from the device over the course of time. This code is written as an MVC app on Azure Cloud service.

My MVC GUI code plots the analysis for an individual device. However, my overall data analysis presents the results based on the device/build behaviour when tested against the same DQ Metric parameters. The job was divided into two parts. The first part was taken care of in the model involved in separating Data Quality metric for individual devices, which had a unique device id, followed by inserting/updating their average Data Quality metric into DQIndicator table. The second part of the job was the controller collating the data for line graph. Finally I used the collated data to plot in the view for graph.

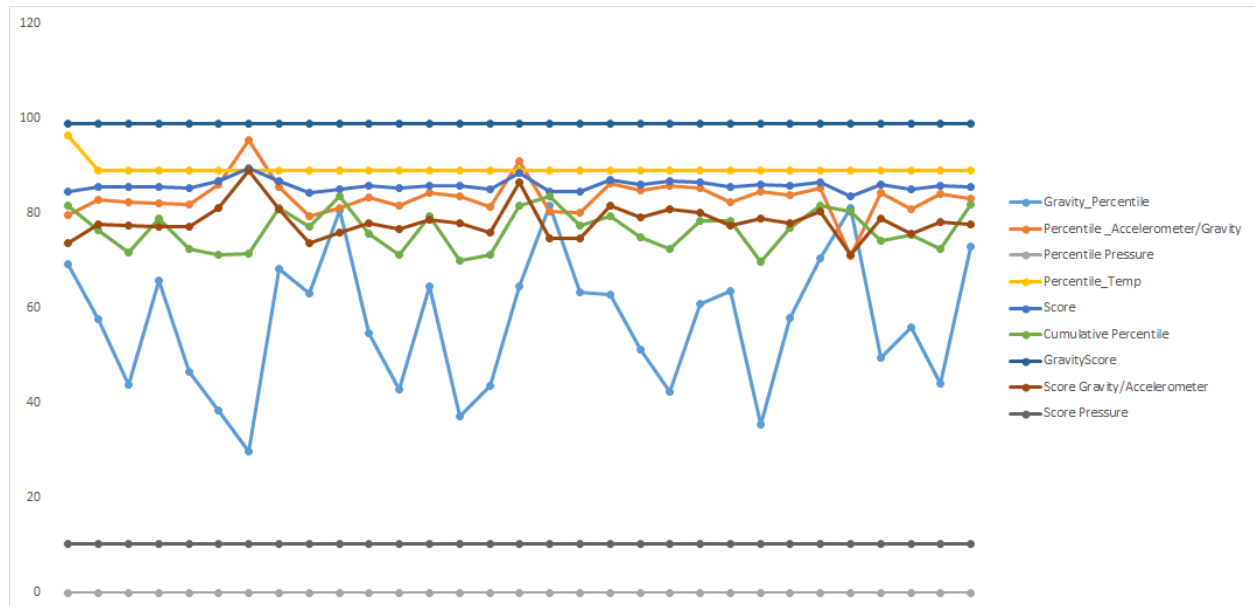


Fig 2.12 MVC Reporting View: Samsung Galaxy S5 Data Quality Indicators variation computed over a period of time

4 Results

4.1 Quality Analysis with Device Type

Out of the 11 android phones that I have used to collect data, I can primarily categorized them into 7 main categories: Samsung GTI9300 (Galaxy S3), Samsung GT - I9500 (Galaxy S4), Samsung GT - I9505 (Galaxy S5), Samsung SCH-I535, Samsung SCH-I545 (Note 2) and Samsung SCH-I605 (Note 4).

Each Parameter carries 20 points in the overall DQ Metric for score and Percentile calculated from all data. When I add the $20 * 5$, I get the cumulative score out of 100.

- 1) *Pressure* - Ambient Real Time Air pressure vs. Air Pressure from Open Weather API
- 2) *Gravity* - Gravity Calculated From Phones vs. Actual Gravity at the latitude
- 3) *Gravity vs. Accelerometer*
- 4) *Magnetic Orientation vs. Sensor Fusion Orientation*
- 5) *Battery Temperature vs. Ambient Air Temperature*

First part of the analysis focuses on information obtained after comparing device types (same model number - for e.g.: comparing cumulative data from Samsung Galaxy S5 with Samsung Galaxy S3 GTI9300)

4.2 Sensors DQ with Device Model Types

This part of result aims at combining all the results achieved from different devices that have the same device model. Further, we check the models against each other in terms of DQ Indicator results.

4.3 Gravity Sensor Quality Metric for Device Model

Since the filters within my app ensured that the gravity was recorded while the phone was stationary, I succeeded in getting a high score when calculating average gravity. The score was a percentage and percentile of average difference of the gravity of device model measured minus the gravity at the specific latitude against the maximum deviation. The graph below reflects the gravity measured across all devices:

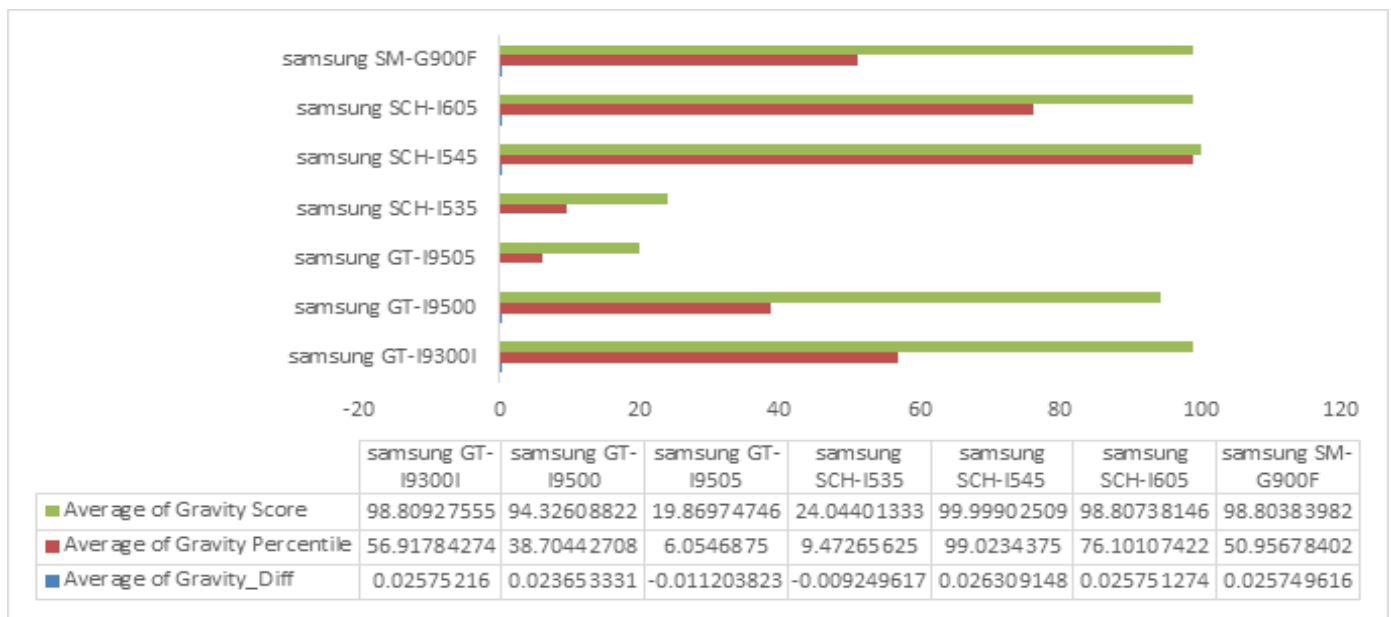


Fig 4.0 Device Model vs. Gravity DQ

4.4 Gravity vs. Accelerometer Sensor Quality Metric for Device Model

The reason behind choosing a DQ metric that selects difference between the average gravity sensor reading and accelerometer sensor is that accelerometer sensor reading are filtered to give you the gravity sensor reading. So hardware filtering make gravity sensor more accurate because of the reduced noise of accelerometer sensor. I wanted to see the deviation in processing at the phone hardware level. Except Samsung SCH – 1535 (Verizon Samsung Galaxy S3) all the others showed a reasonably high score and percentile.

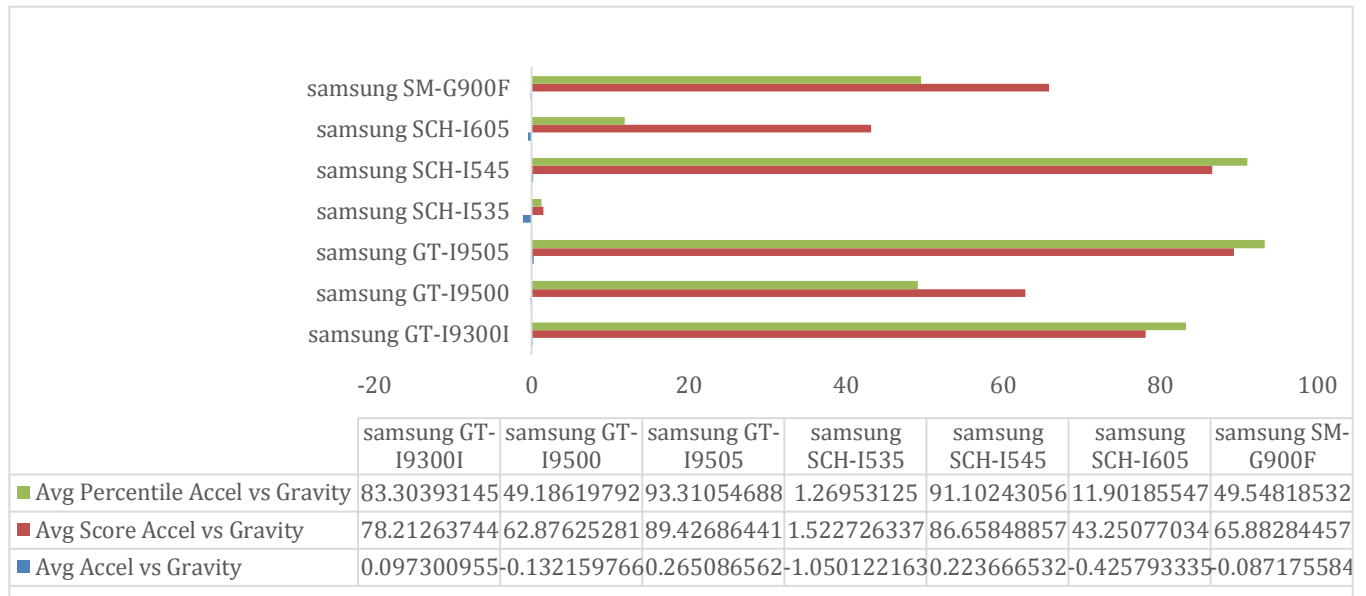


Fig 4.1 Device Model vs. Accelerometer/Gravity DQ

4.5 Battery Temperature Sensor and External Temperature Quality Metric for Device Model

This was an interesting experiment, which pits battery temperature against the external temperature (Received from Openweathermap API). The noise didn't allow any interesting observations from this metric. However, if a specific noise filter that can recognize whether the user is indoor or outdoor or if the phone is in the user's pocket is available – it will be very valuable in predicting ambient air temperature by using the battery sensor information. Another ingredient that can be used as a training set for such a machine learning system will be an external sensor that can sense the ambient air temperature as well outdoor temperature.

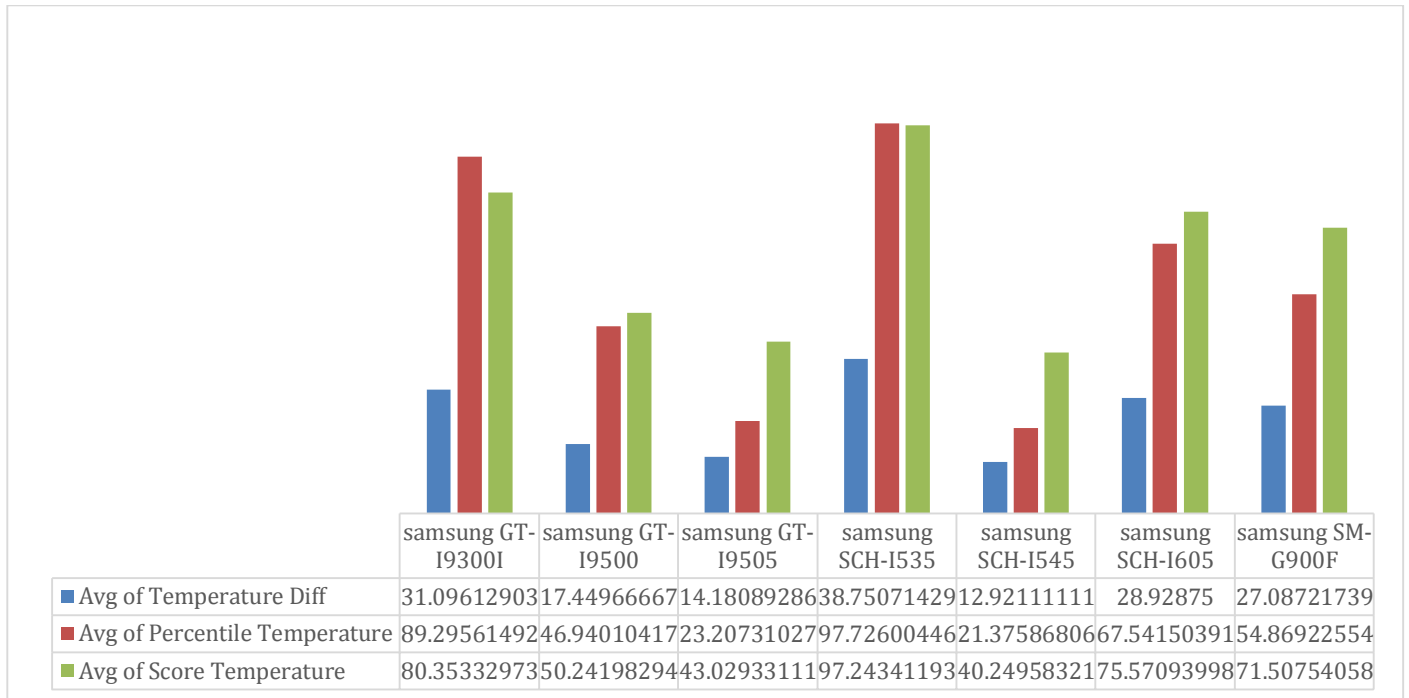


Fig 4.2 Device Model vs. Battery Temp/ External Temp DQ

4.6 Pressure Sensor Quality Metric for Device Model

The Pressure Sensor quality metric was one of the most interesting and successful DQ metrics chosen as it helps obtain very useful information including whether or not the phone's barometric sensors are working properly. We used openweathermap API to get the air pressure. As we know, the ambient air pressure and external air pressure is quite similar in most cases. I gathered that the Samsung GT -19500I (Galaxy S4) devices give a very high percentile of overall accuracy and score accuracy as compared to other devices. If securely leveraged for crowdsourcing data collection, the pressure sensor information from android smart phones can also be used to create micro meteorological weather models.

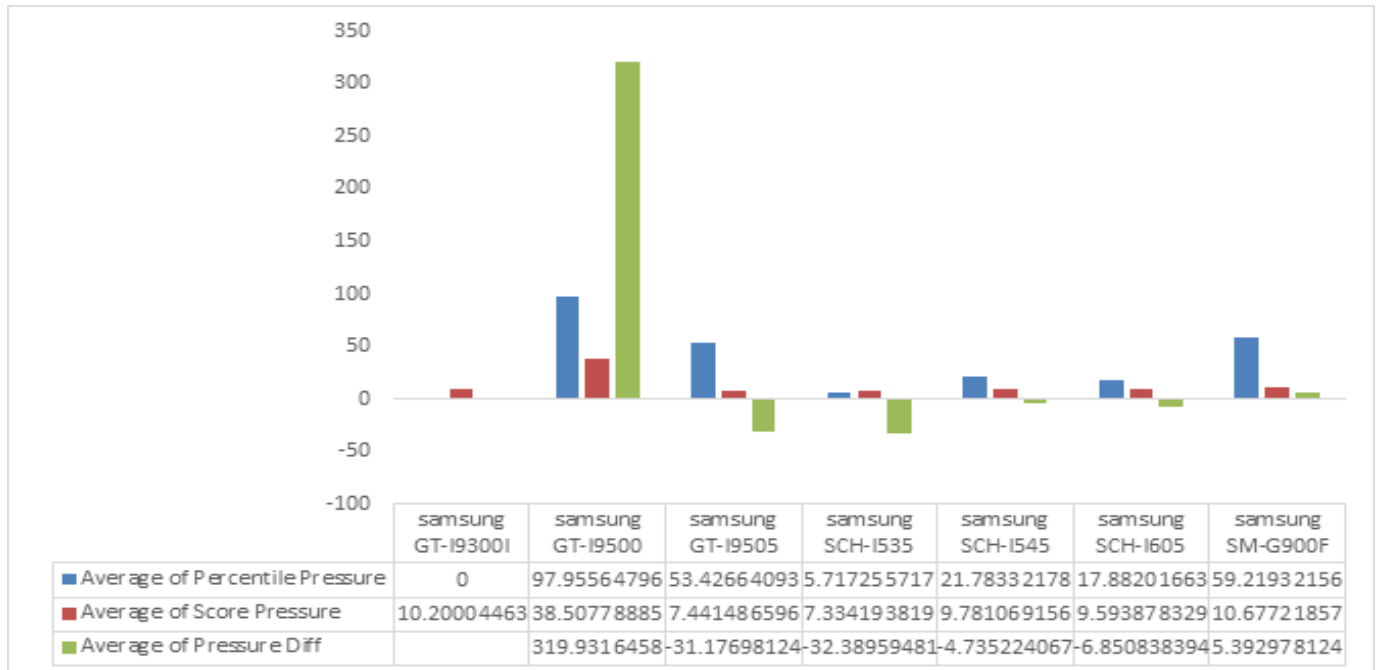


Fig 4.3 Device Model vs. Pressure DQ

4.7 Score and Percentile for Device Model

Finally the graph below shows the cumulative percentile and score of all device models by adding all the DQ Metrics together. My observation from the graph was that the Samsung SCH-1535 (Verizon Samsung Galaxy S3) has the least trustworthy sensor and sensor in Samsung GT-19300I (Samsun Galaxy Neo S3) had the highest level of accuracy.

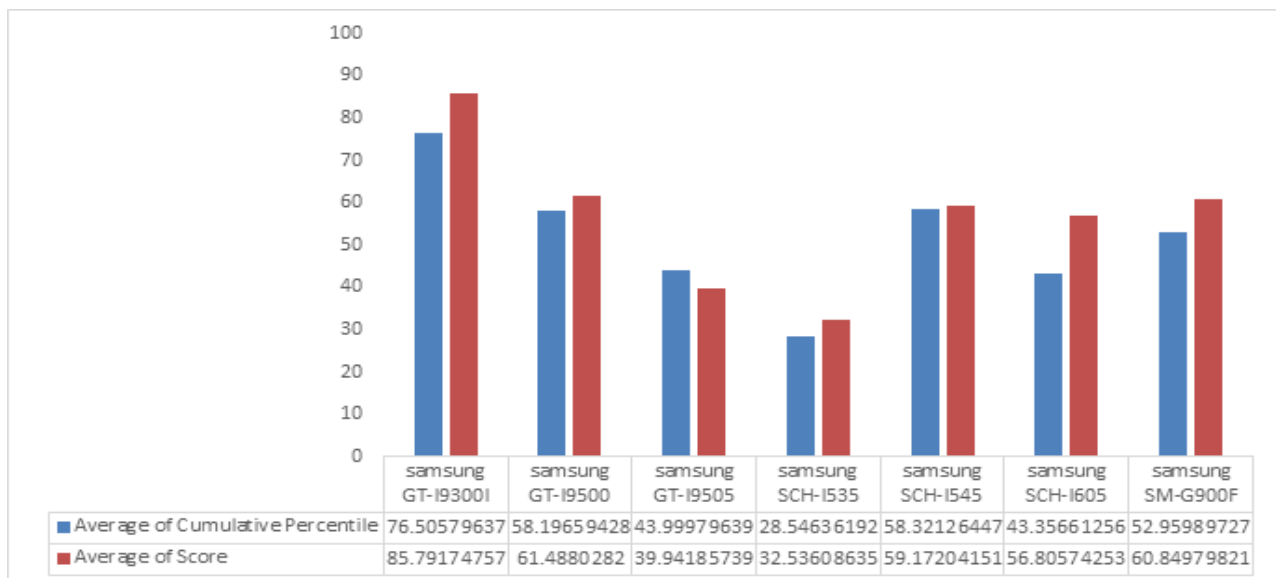


Fig 4.4 Device Model vs. Cumulative DQ

4.8 Score and Percentile for Device Model

Complete Sensor Quality Data Combined in One Table

	samsung GT-I9300I	samsung GT-I9500	samsung GT-I9505	samsung SCH-I535	samsung SCH-I545	samsung SCH-I605	samsung SM-G900F
■ Average of Percentile_Accelerometer/Gravity	83.30393145	49.18619792	93.31054688	1.26953125	91.10243056	11.90185547	49.54818532
■ Average of Percentile Pressure	0	97.95564796	53.42664093	5.717255717	21.78332178	17.88201663	59.21932156
■ Average of Score	85.79174757	61.4880282	39.94185739	32.53608635	59.17204151	56.80574253	60.84979821
■ Average of Cumulative Percentile	76.50579637	58.19659428	43.99979639	28.54636192	58.32126447	43.35661256	52.95989727
■ Average of Score Pressure	10.20004463	38.50778885	7.441486596	7.334193819	9.781069156	9.593878329	10.67721857
■ Average of Score Gravity/Accelerometer	78.21263744	62.87625281	89.42686441	1.522726337	86.65848857	43.25077034	65.88284457
■ Average of Accelerator_Gravity Diff	0.097300955	-0.132159766	0.265086562	-1.050122163	0.223666532	-0.425793335	-0.087175584
■ Average of Gravity Score	98.80927555	94.32608822	19.86974746	24.04401333	99.99902509	98.80738146	98.80383982
■ Average of Gravity Percentile	56.91784274	38.70442708	6.0546875	9.47265625	99.0234375	76.10107422	50.95678402
■ Average of Percentile_Temp	89.29561492	46.94010417	23.20731027	97.72600446	21.37586806	67.54150391	54.86922554
■ Average of Temp Diff	31.09612903	17.44966667	14.18089286	38.75071429	12.92111111	28.92875	27.08721739
■ Average of Pressure_Diff	0	319.9316458	-31.17698124	-32.38959481	-4.735224067	-6.850838394	5.392978124
■ Average of Gravity_Diff	0.02575216	0.023653331	-0.011203823	-0.009249617	0.026309148	0.025751274	0.025749616

Fig 4.5 Device Model vs. all DQ

4.9 Device ID (IMEI) vs. Sensor Data Quality Indicators

Note: The device Id shown in the report below has been shaded to avoid privacy infringement. This part of the result includes data collected from individual devices with unique IMEI number against one another. The interesting observation here is that the samsung SCH-I605 (Samsung Galaxy Note 4) device shows best results in most of the DQ Metric criterions.

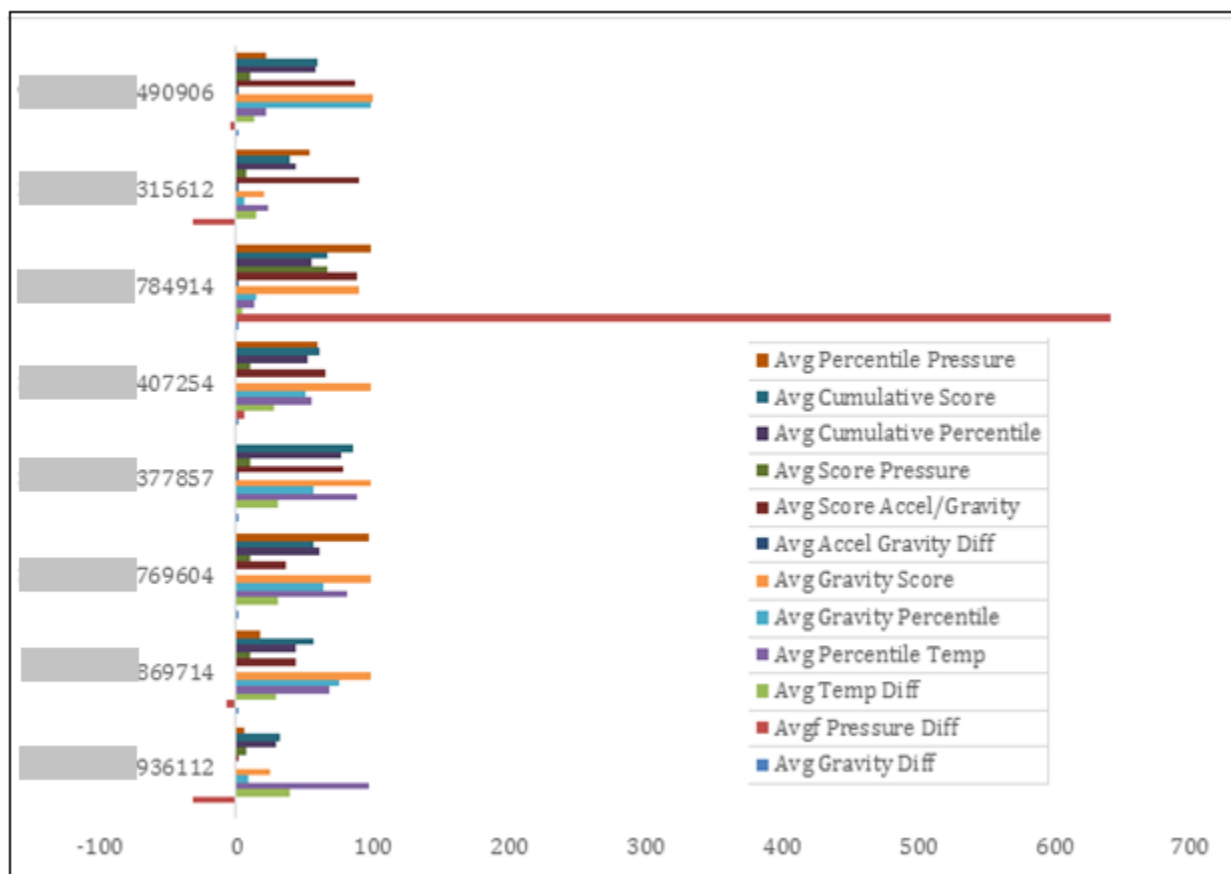


Fig 4.6 Device ID (IMEI) vs. all DQ Graph

Device ID(IMEI)	36112	69714	769604	377857
■ Avg Percentile Pressure	5.717255717	17.88201663	96.95079695	0
■ Avg Cumulative Score	32.53608635	56.80574253	55.97477799	85.79174757
■ Avg Cumulative Percentile	28.54636192	43.35661256	61.4121784	76.50579637
■ Avg Score Pressure	7.334193819	9.593878329	10.03039596	10.20004463
■ Avg Score Accel/Gravity	1.522726337	43.25077034	36.55053604	78.21263744
■ Avg Accel Gravity Diff	-1.050122163	-0.425793335	-0.52604125	0.097300955
■ Avg Gravity Score	24.04401333	98.80738146	98.80979513	98.80927555
■ Avg Gravity Percentile	9.47265625	76.10107422	63.54166667	56.91784274
■ Avg Percentile Temp	97.72600446	67.54150391	81.0546875	89.29561492
■ Avg Temp Diff	38.75071429	28.92875	30.26	31.09612903
■ Avgf Pressure Diff	-32.38959481	-6.850838394	-1.917354329	
■ Avg Gravity Diff	-0.009249617	0.025751274	0.025752404	0.02575216

Fig 4.71 Device ID (IMEI) vs. DQ Table

Device ID(IMEI)	407254	784914	315612	490906
■ Avg Percentile Pressure	59.21932156	98.96049896	53.42664093	21.78332178
■ Avg Cumulative Score	60.84979821	67.00127842	39.94185739	59.17204151
■ Avg Cumulative Percentile	52.95989727	54.98101016	43.99979639	58.32126447
■ Avg Score Pressure	10.67721857	66.98518173	7.441486596	9.781069156
■ Avg Score Accel/Gravity	65.88284457	89.20196959	89.42686441	86.65848857
■ Avg Accel Gravity Diff	-0.087175584	0.261721719	0.265086562	0.223666532
■ Avg Gravity Score	98.80383982	89.84238131	19.86974746	99.99902509
■ Avg Gravity Percentile	50.95678402	13.8671875	6.0546875	99.0234375
■ Avg Percentile Temp	54.86922554	12.82552083	23.20731027	21.37586806
■ Avg Temp Diff	27.08721739	4.639333333	14.18089286	12.92111111
■ Avgf Pressure Diff	5.392978124	641.780646	-31.17698124	-4.735224067
■ Avg Gravity Diff	0.025749616	0.021554259	-0.011203823	0.026309148

Fig 4.72 Device ID (IMEI) vs. DQ Table

4.10 Sensor Monitoring: Continuous Readjustment in DQ Indicators over a period of time

Final part of the result shows continuous integration of DQIndicator. Since the DQEngine was a web job that ran at an hour's interval, for a device the DQ indicators kept changing as seen by the fluctuations in the plots (on next page). This plot enables us to measure sensor-monitoring capability over a long period of time while data pouring in. Below are 2 example of the combined change in DQ Metric for 2 different devices.

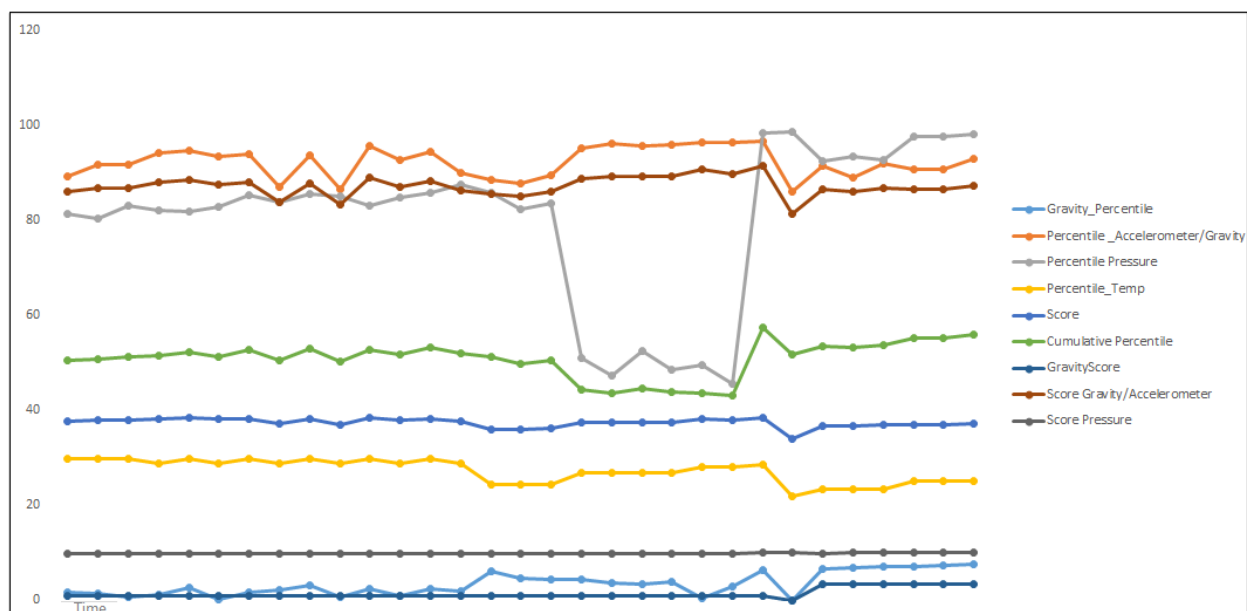


Fig 4.81 DEVICE A: Change in the DQ Indicators over period of continuous integration for an individual device (unique IMEI)

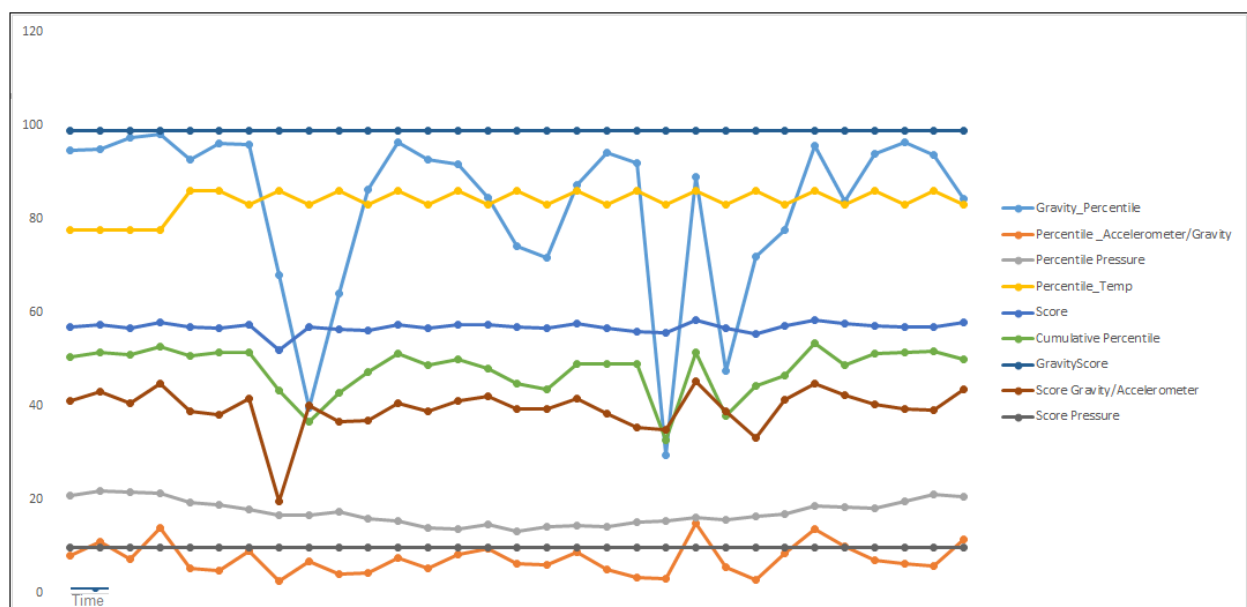


Fig 4.82 DEVICE B: Change in the DQ Indicators over period of continuous integration for an individual device (unique IMEI)

4.11 Key Results

1. Device-to-cloud integration.
2. Filtering and cleansing data within the app as well as at the DQ Engine. App level filtering helps avoid incomplete data with a very low chance of any outlier. This also ensured that the data at DQ Engine was good enough to be parsed and processed. After further customization, more noise reductions can be made to the data received into the cloud.
3. Notification system on successful completion or failure of data collection within the app.
4. More than 1000 instances of collected sensor data from over 10 devices within 4-5 weeks.
5. Data correlation for different devices in different locations.
6. DQ Metric for an instance of data sent towards the overall data collected for the device and that collected from various sources.
7. Sensor fusion helps ease out noise from gyroscope sensor and accelerometer data.
8. One of the device which used Sensor Test app to record data had an abnormally high pressure difference compared to other devices. Unless the data is spoofed, I concluded that the pressure sensor in this device was not working accurately as the difference amounted to 36 HPA. Moreover, if it had been only for a single instance - then it could have been considered an outlier but in this case the average reading had a high offset.

5.0 Future Work

1. I would like to release the iPhone and Windows Phone version of the SensorTest App. I have designed the android app using Xamarin with high code reusability factor.
2. This could be further built into a diagnostic tool within the app to get sensor calibration and accuracy information.
3. This data with prior permission from the app users can be exposed to a data broker to share high quality trustworthy sensor information.
4. I would also like to carry out a study on Battery Temperature to predict ambient air temperature ^{[20][21]}. I believe there is further scope in filtering out noisy data and generating ambient air temperature, if we are able to generate sufficient training set for our machine learning system.
5. Measuring **Timeliness** of the Data. This is DQ Indicator that I would like to calculate to check how much time it takes overall for the data to get populated from the app to the Database. There was a column `_createdAt` in the table, which gave date time offset of time when the record was added to the table. We can also add the timestamp to reflect the time at which the data was bundled in and sent to the API. The difference between the two values (in seconds) should give me a fair idea about the timeliness of the data. Timeliness can also be correlated with the latitude and longitude of the location from which the data was sent. While setting up the cloud, the major region where my cloud was hosted was east US. However I tried to keep the service accessible from all geolocations. This new metric of data can be achieved by correlating timeliness and distance (latitude, longitude) from central location in East US for further analysis.

6.0 Conclusions

1. The Gravity Sensor was unable to supply a lot of very useful information. The filters in the phone ensured that raw data was only sent while the sensors were absolutely stable on a flat surface.
2. Comparing the ambient air temperature is not very helpful as room temperature is different from external temperature (room heating/cooling). However I went with it, as it can potentially be good source to determine ambient air temperature in future.
3. No particular correlation was found between phone security and sensor data calibration error due to lack of data from real untrustworthy source.
4. Privacy information can be compromised with unauthorized use, such as by secretly collecting user information using an app. Intrusion detection and smartphone Trojan systems can acquire data using this principle.
5. The precision in terms of accuracy of data collected from 11 smartphones and 7 models is very high so far.
6. In theory, over a period of time sensors should degrade and relay less accurate information. This tool can be used to track degradation and to carry out a diagnostic device sensor check and calibration of the sensor devices.
7. Data from pressure sensor is extremely useful for real time weather predictions as compared to general geolocation based weather prediction systems.

7.0 References

- [1] L. Reznik, "Integral Instrumentation Data Quality Evaluation: the Way to Enhance Safety, Security, and Environment Impact," presented at the 2012 IEEE International Instrumentation and Measurement Technology Conference, Graz, Austria, May 13-16, 2012, 2012.
- [2] S. E. Lyshevski and L. Reznik, "Processing of extremely-large-data and high-performance computing," in International Conference on High Performance Computing, Kyiv, Ukraine, 2012, pp. 41-44.
- [3] G. P. Timms, P. A. J. de Souza, L. Reznik, and D. V. Smith, "Automated Data Quality Assessment of Marine Sensors," *Sensors*, vol. 11, pp. 9589-9602, 2011.
- [4] G. P. Timms, P. A. de Souza, and L. Reznik, "Automated assessment of data quality in marine sensor networks," in OCEANS 2010 IEEE - Sydney, 2010, pp. 1-5.
- [5] J. Cappos, L. Wang, R. Weiss, Y. Yang, and Y. Zhuang. BlurSense: Dynamic fine-grained access control for smartphone privacy. *Sensors Applications Symposium (SAS)*, 2014.
- [6] A. Rafetseder, F. Metzger, L. Puhlinger, K. Tutschku, Y. Zhuang, and J. Cappos. Sensorium—a generic sensor framework. *Praxis der Informationsverarbeitung und Kommunikation*, 36(1):46–46, 2013.
- [7] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, T. Anderson. Retaining Sandbox Containment Despite Bugs in Privileged Memory-Safe Code. *ACM Conference on Computer and Communications Security (CCS'10)*
- [8] Android Developer Sensor Overview Reference http://developer.android.com/guide/topics/sensors/sensors_overview.html
- [9] C. Perera, A. Zaslavsky, P. Christen, A. Salehi, and D. Geor-gakopoulos, "Capturing sensor data from mobile phones using global sensor network middleware," in IEEE International Workshop on Internet-of-Things Communications and Networking 2012 (PIMRC 2012-Workshop-IoT-CN12), Sydney, Australia, September 2012.
- [10] Telerik Developer Network Reference <http://developer.telerik.com/products/building-an-android-app-that-displays-live-accelerometer-data/>
- [11] Phone Tester App by Miguel Torres
- [12] Sensor List App by Idea Matters
- [13] Abyarjoo F., Barreto A., Cofino J., Ortega F., "Implementing a Sensor Fusion Algorithm for 3D Orientation Detection with Inertial/Magenetic Sensors", CISSE 2012.
- [14] Quadri, S. A.; Sidek, Othman and Abdullah, Azizul bin. "A Study of State Estimation Algorithms in an OktoKopter", *International Journal of U- & E-Service, Science & Technology*, 2014.
- [15] Clint Gibler, Jonathan Crussell, Jeremy Erickson, Hao Chen, AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale, *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, June 13-15, 2012, Vienna, Austria
- [16] William Enck , Peter Gilbert , Byung-Gon Chun , Landon P. Cox , Jaeyeon Jung , Patrick McDaniel , Anmol N. Sheth, Taint Droid: an information-flow tracking system for real-time privacy monitoring on smartphones, *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, p.1-6, October 04-06, 2010, Vancouver, BC, Canada
- [17] Android developer reference, <http://d.android.com/>
- [18] Google.Google play, <http://market.android.com>
- [19] Bitblaze, <http://bitblaze.cs.berkeley.edu/>
- [20] <http://yro.slashdot.org/story/14/11/27/1451203/ubers-android-app-caught-reporting-data-back-without-permission>

- [21] M. Li, A. Davoodi, and M. Tehranipoor, "A sensor-assisted self-authentication framework for hardware Trojan detection," in Proc. of the IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE12), pp. 1331-1336, 2012.
- [22] Latitude to gravity Correlation <http://epsc.wustl.edu/~epsc454/grav-corrections.html>
- [23] Sachs, David "Sensor fusion on android devices: A revolution in motion processing", Google Tech Talks, 2010
- [24] <http://opensignal.com/reports/battery-temperature-weather/>
- [25] <http://www.battcon.com/PapersFinal2003/McCluerPaperFINAL2003.pdf>
- [26] <http://openweathermap.org/api>
- [27] <https://components.xamarin.com/view/azure-mobile-services>
- [28] <http://developer.xamarin.com/>
- [29] <http://androidapi.xamarin.com/>
- [30] Milestone 1, Project Milestone, Ankan Mookherjee and Leon Reznik
- [31] Milestone 2, Project Milestone, Ankan Mookherjee and Leon Reznik
- [32] Milestone 3, Project Milestone, Ankan Mookherjee and Leon Reznik
- [33] Project Proposal, Project Milestone, Ankan Mookherjee and Leon Reznik
- [34] Poster, Project Poster, Ankan Mookherjee and Leon Reznik