# S2AS : Sensor as a Service

**Harshil Gandhi**
Masters Candidate
Columbia University
503-West, 122nd Street
hmg2138@columbia.edu

**Kuber Kaul**
Masters Candidate
Columbia University
503-West, 122nd Street
kk2872@columbia.edu

**Digvijay Singh**
Masters Candidate
Columbia University
503-West, 122nd Street
nr2445@columbia.edu

**Prof. Roxana Geambasu**
Adjunct Professor
Columbia University
116th Street
roxanna@cs.columbia.edu

## 1. Abstract:

With the launch of Apple's IOS and Google's android operating systems the sale of smartphones has increased exponentially. Android emerges at the top in this market since it is open source with permissive licensing which allows the software to be freely modified by enthusiastic developers. Because of this a lot of interesting apps find their way into the Android app store which is currently estimated to hold around 800,000 apps. A lot of these apps make use of the data from the sensors present in device to find out some information about related to the physical environment that the user of the device is in. For instance, GPS data from the device would tell us the exact location of the device and hence of the person using it or a very low light sensor reading would indicate that the user is in relatively less lit place which could mean that he is sleeping or maybe at a presentation. Our application caters to applications of this kind by providing all the sensor information from the user's device on the Cloud, as a service. We have implemented two use cases to show how this service might be utilized. The first use case is called "wreckwatch". Here data from the appropriate sensors is continuously monitored and analysed to detect if the phone and hence its user has been in an accident. If such values are detected an alert is sent to the people registered as his emergency contacts via call and SMS requesting them to check on him. The second use case is called "battery alert". Here whenever the battery level in the device goes below a certain value an alert is sent to people registered as his emergency contacts on the app via call as well as SMS, asking them to get in touch with him soon if they had anything of importance to convey to him. Since the app sends out alerts every time it thinks the user might be in trouble it has been rightfully named "BatSignal".

## 2. Introduction

### 2.1 Motivation

Sensors as a service, as the name suggests, is mostly about the sensing ability of the smart phones. The phones in today's time now have become more capable than just a traditional phone back in the day. The cellphones now are loaded with tons of features, including all the sensors that are newly introduced a few years back. It's just a matter of time before these sensors can now be used in an open-source manner for monitoring and acquiring mass data for various use cases, analyses and researches.
The main motivations for the idea of Sensors as a service are:

**1.** On a collective scale, involving analysis of data from a large group of subscribers, this service can be used very effectively to draw out certain

conclusions about various test cases based on the aforesaid collective data set that be generated and observed.

**2.** On an individual scale, the service can be used to monitor particular subscribers by authorized clients who can locate and monitor the subscriber's activities and can be set as a potential emergency contact by the subscriber to be notified in case of some events that can trigger the sensors in a particular way that can be monitored.

**2.2 Services Provided**

A lot of the services provided by the Sensors-as-a-service idea are based on the general sensors that a smartphone are equipped with these days. We, in our dummy app have concentrated on Android phones. However, the entire framework of Sensors-as-a-service can be easily extended to support other mass-produced cell-phones like the iPhone, Windows mobiles etc.

The services provided mainly include any sensors and the service as a whole can be extended to support newer sensors or data incoming from the phones. Some services which are easily accessible with today's phones are:

**Hardware installed sensors:**

1. Accelerometer

2. Gyroscope

3. Thermometer

4. Barometer

5. Linear Accelerometer

6. Photometer

7. Magnetometer or Compass

8. Proximity sensor

9. Rotational Vector

10. Humidity sensor

**Indirect sensors:**

1. Battery

2. Cellular Network

3. Wifi connectivity

4. Mobile internet connectivity

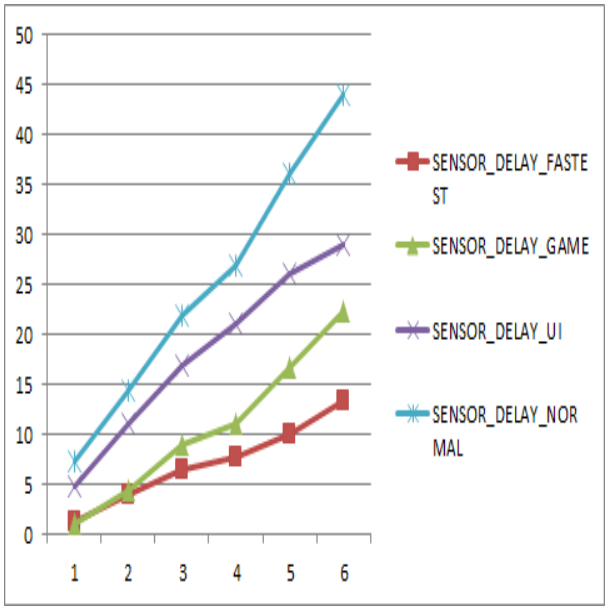
5. GPS

6. Microphone

7. Camera

Besides the sensors, the end user or the client that is accessing the services also gets a unique feature from the service. The manner in which the user wants to access the sensor data can be decided by the user himself. The user may decide to directly access the raw sensor data so that he can apply his own operations on them to get a certain set of expected results for him, or she may decide to query the entire database of the sensor information that is collected by our service. This querying can be done in a high level sentence or a collection of verbs, based on which the service decided to act in a way that optimizes the battery usage of client end devices for sensing and the accuracy of the results obtained based on the accessed and processed data.

The sensors (Android) themselves provide the flow of data from all these sensors in 4 basic types of speed flow which we can modify according to the type of the query which we would explain in much detail later.



The 4 types of speed as explained in the above are the four different speed Android offers for sensor information.

## 3. Challenges associated

There are many challenges associated with implementation of this idea in the form of a running service. Some of them that we faced are listed below. Some others that may be potentially cumbersome challenges while implementing apps or end-user product that uses Sensor-as-a-service feature are also listed below.

**Challenges during design:**
The main challenge during the architecture design phase was to weigh the two brightest options that seemed logically the best ones. They were whether to follow the pull based approach or the push based approach. This is the main crux of the architecture. Referring the WreckWatch paper which is the inspiration behind building our test dummy app, we figured that through their testing in the paper, the authors found out that for simpler upload tasks, it is better to have a pull based approach rather than the push based one. This was based on the inference that in case of push based approach (where the app on constantly pushed sensor data on the cloud database), the client side cell phone battery usage and power consumption would be futile when there is no need for the data to be posted on the database as there are no jobs requested by a user that needs the sensors. To avoid this, we decided to follow the pull based approach (where the backend server on the cloud requests particular sensors info for a particular time period at a particular sampling rate, all computed on the backend) which saves all the unnecessary power consumption on the sensing devices.

**Challenges during discussion about use cases:**
We believe this was the most important challenge in our process where in we had to consider the possibilities of apps that can be built and the range of use cases that can be covered by the architecture of the design. In order to consider the coverage of the broad span of use cases that we had figured, it was a huge hurdle to finalize on the right methods to implement the architecture.

Firstly, due to the variance in the separate modules of the architecture poises problems for generalizing a method that several different use-cases can share and re-use. This led to separate tracks of varied versions of the same module being formulated for taking care of different possible sample spaces in that model. For example, in case of the sensor for accelerometer to be accessed, the sampling space was supposed to be quick for achieving reasonable accuracy, however, for battery, there was a separate heuristic to sample the monitoring events. There had to be a separate listener for example in order to listen for updates in battery status.

Secondly, the data format for different use cases that we formulated, had a potential of coming in different shapes and sizes, meaning a lot of knowledge can be conveyed by very little information at some times, whereas draining the entire battery to get the appropriate data was sometimes insufficient. These inconsistencies led to a wide range of case handling mechanisms in terms of separate sensing modules having their own separate data upload modules. In short, the main challenge is generalizing the entire single process of say sensing or uploading or processing. These cannot be generalized due to the different and extremely wide nature of the use cases that are made possible by offering sensors of the smartphones as a service.

**Challenges during implementation:**
As stated above, the most important challenges were figuring out the exact architecture and module structure of the framework that we would implement so as to satisfy most possible test cases for achieving a wide coverage of different possible sensors domain.

The approach that we pursued is that of not having any computation to be done on the phone at all. Computation as in the processing of which sensors are required, how much sampling of that sensor data, for how long and in what order of upload to the database on the cloud was all decided on the server backend, which is again on the cloud. By simple message passing mechanism and better performance monitoring and optimizing

this primitive idea of light computation at the client end and quicker data upload and processing at the backend, the system can be optimized even further, which of course, can be further improved which is the next big challenge

**Challenges during testing:**

This was quite a unique challenge requiring a separate set of mechanisms for testing. The main testing that needed to be done was of the following characteristics of the dummy app that we built:

**Connecting Android phones to Amazon's RDS:**

This was a major hurdle in the project as uploading the data in the fastest possible manner had to be of utmost priority. Initially, the approach that we followed was to connect the RDS directly to the Android device through SQL connectors for Java. On experimentation, it turned out that the connection and the drivers that run the connection for Java to AWS RDS are too heavy for the light devices like a smartphone with limited computing resources. Because of this, the data upload rates were far less than satisfactory to judge based on them whether or not an accident has occurred or any other even for that matter.

The solution to this we figured out was to have an intermediary servlet connector, hosted on the cloud web server on an AWS EC2 machine, which directly entered data that it receives to the RDS. Hence, the connectivity overhead is now that of the high computing resourceful virtual machine on the cloud. As for transferring the sensor data from the phone to this sort of intermediary web service, a simple lightweight HTTP request was enough. This request had a JSON object embedded within it which had all the necessary data. This resolved the major challenge that we faced and we achieved far better (almost 20 times faster) data transfer rates at around 5 records of accelerometer (the costliest sensor upload unit) per second, which was down to 0.2 records per second in case of a direct RDS connection.

**Django backend server on Amazon's EC2:**

Separate from the EC2 web service that had servlets for handling data uploads to the RDS database, we had to run an EC2 instance that hosts the web-app that the end user would use to access the data in many possible ways. This web-app was the UI for the end user. This was hosted on an EC2 server on the cloud which makes it highly scalable and we implemented it in Django.

**Accelerometer threshold:**

Setting the threshold value right for detecting the jerks and the G-forces on the phone when being operated normally and the sensors being activated was another testing issue that we faced. According to the Wreckwatch paper, the authors calculated that a G-Force value observed to be 3.5 was necessary to conclude that the observation was from a crash or accident as it was a serious jerk which we were unable to replicate. As a result, we lowered the threshold and had to perform several experiments with the phone being tossed around for getting the data good enough for the triggering of the event of an accident. For our dummy test app and demonstration settings of our prototype of the backend, we set the threshold to be lesser, at around 1.75 which is good enough to detect moderate to heavy jerks (simulated by throwing the cellphone). Getting this parameter right was one of the main challenges as well.

**Microphone and GPS data:**

In general, getting the data from all the sensors (both hardware and indirect) mentioned in section 2 above was relatively more structured and concrete. However, the same could not be said for the data that we record from Microphone and GPS. In case of the microphone data that we record, firstly, it starts the Voice Recorder app in the android phone, which means there is no simpler or less power consuming way to record surrounding sound. Further, the data that we received had huge analytical challenges in that analyzing the frequency or amplitude, syncing the data in terms of exact time to the accelerometer data to eliminate false positives and false negatives as well, so as to confirm that a crash has occurred. In case of the GPS, it was the most power consuming operation, for which the main challenge was to devise a plan to pull the data from GPS sensors in the phone as scarcely as possible. To do this, we thought of using the accelerometer data and check if there is a constant

movement in the "x" direction (x represents horizontal movement parallel to the ground). If there is a constant movement, then there is a high chance that the GPS location of the device has shifted. So depending on this, we propose to fetch data from the GPS sensor. We did not implement this as t was out of the scope of our dummy app, but we propose this so that this can be one of the very good use cases that can be implemented using the Sensor-as-a-service platform as the data source.

# 4. Technologies

This project uses various technologies that are integrated and made to work in an efficient way to reduce load on the mobile application and shift the computation to the Server..

Among these technologies are:

**a) Amazon Web Services:** Amazon Web Services such as Relational Database Service (RDS) and Elastic Cloud Compute (EC2) were used.

RDS is a service offered by Amazon to makes scaling a relational database on the cloud easy. It was used in this project to store the User table , which contained the names and numbers of all the users that subscribed to our S2AS application. It also stores the values of the different sensors like Accelerometer, Gyroscope etc. In their respective formats.

Two Amazon EC2 machines were used. Tomcat Servlet was deployed on windows EC2 machine, which was used for the communication between Android client and RDS. The other one was a Linux EC2 that was used for deploying the Django application.

**b) Django:** The Django web server, written in python, handles the computation of the readings from the

phone. Django was used in the design to reduce the computational complexity overhead on the client and therefore reduce battery consumption. Another advantage with this design is that it is off-site, hence monitoring of data can be done though the phone is slightly damaged.

**c) Android client:** The app is built for the android platform. The app basically scraps through a message and then sends the request to the sensors which sends back the sensor data that is parsed and sent to the RDS

**d) Twilio Cloud Communications:** The python API of Twilio was used to send SMS messages to the users/subscribers depending upon what use case of our S2AAS we are talking about.Every subscriber whose sensor data is required by the client, is sent an sms.This sms activates the sensors of the subscriber , the android app talked above starts recording its data and stores it in Amazon RDS. This API integrates well with the backend Django framework to provide this functionality.

**e) MySQL** : This is used on the backend of Amazon RDS to manipulate , update and store relational tables containing information Users and Sensors.

# 5. Approaches ( Solutions )

### 5.1 Architecture

One of the most important challenges for us was to create an architecture that would support S2AS well enough to provide a general platform for the user to access all the sensors through an interactive website and then be responded back with the data returned back from the sensor itself. We wanted to offload all the processing from the android app itself to our django server which as we had seen in our experiments had a detrimental effect on the power consumption of the mobile device. The architecture for S2AS was clearly thought over and with enough deliberations we came up with a model that satisfied all the requirements that we had and had minimum redundancy in the architecture.

The control flow is smooth and as follows, the user, which can be any random person requests scenario of

1. Analysing and processing sensor data in the Android app

2. Analysing and processing sensor data in the server

The first set of experiment that we used for our analysis was using the power tutor app in the android phone for testing as to how much power was consumed as we worked our way through single queries on S2AS using firstly just the android app and then secondly offloading the processing on the server itself. We picked four random locations of our choice and conducted this experiment 4 times.

| Sensor | Avg. Power(mW) (L) : Single Q | Avg. Power(mW) (S) : Single Q | Avg. Power(mW) (S) : Multiple Q | Avg. Power(mW) (L) : Multiple Q |
|---|---|---|---|---|
| Accelerometer | 82.68 | 76.21 | 93.74 | 108.46 |
| GPS | 308.43 | 304.54 | 46.89 | 49.88 |
| Battery | 35.68 | 30.56 | 42.56 | 45.78 |

PowerTutor is an application for Google phones that displays the power consumed by major system components such as CPU, network interface, display, and GPS receiver and different applications. The application allows through our website the number of sensors and the person that is registered with our app and the number of samples for the sensors to be asked for. Once this information is attained, the django server sends a request message containing the metadata from the website to the Twilio API. Twilio API routs the message to the android app which then requests the mentioned sensors n the message and gets the sensor data from them. Android app then sends the information to the RDS ( MySQL ) database that we maintain in the cloud on Amazon EC2 machine . Finally our django server then extracts the specific data from the RDS database and analyses it in server to gain some meaning out of the data as per our use case. This information is finally displayed on our website for the user to check.

The time of the cycle above is 6-8 seconds depending mostly on the lag of the message from Twilio to our Android app.

We debated over offloading our processing of data for extracting relevant information from the Android app to the server. We followed our debate by two test cases as to how much the power is consumed when using both are software developers to see the impact of design changes on power efficiency. Application users can also use it to determine how their actions are impacting battery life. PowerTutor uses a power consumption model built by direct measurements during careful control of device power management states. This model generally provides power consumption estimates within 5% of actual values. A configurable display for power consumption history is provided. It also provides users with a text-file based output containing detailed results. You can use PowerTutor to monitor the power consumption of any application. PowerTutor's power model was built on HTC G1, HTC G2 and Nexus one.

Each hardware component in a smartphone has a couple of power states that influence the phone's power consumption. For example, CPU has CPU utilization and frequency level, OLED/LCD has brightness levels. The power model in PowerTutor is constructed by correlating the measured power consumption with these hardware power states. The power states we have considered in the power model includes:

- CPU: CPU utilization and frequency level.

- OLED/LCD: For hardware with LCD screen and OLED screen without root, we consider brightness level; For hardware with OLED screen with root access, we consider brightness together with pixel information displayed on the screen.

- Wifi: Uplink channel rate, uplink data rate and packets transmitted per second.

- 3G: Packets transmitted per second and power states.

- GPS: Number of satellites detected, power states of the GPS device (active, sleep, off).

- Audio: Power states of the Audio device (on, off).

The second set of experiment was physically keeping track our the battery consumption as we worked our way through running single query on our app and than multiple query on our app on both just the android app and then offloading the entire processing and analysis of data to the remote server that we maintained on the EC2 machine. We ran the query for an hour flat therefore total of two hours where each hour was spent on running the single query and then multiple query and keeping track of battery and performance.

The results that were returned to us using both these test cases for multiple locations echoed the same thought process that we had envisioned.

There was a very slight drop in terms of power consumption which was expected factoring in the power consumption on sending the sensor data from the device to the database in cloud over the network when we tested for single/multiple query over the local server i.e in the Android as compared to the remote server that we setup in the cloud using django. Even though the power consumption was far less than we had envisioned it was still more efficient model than processing information in the android app itself. We, therefore proceeded with our model of offloading all the processing to a remote server rather than doing it all in our local database.

**5.2 Thresholds**

We developed three use cases for our platform to provide with examples as to how our architecture can actually be used for a number of applications. We plan to develop an API for S2AS so that it can in future easily be integrated into any project thus abstracting them from the interior details. Some problems that we ended up facing were while processing data coming from the sensors we needed to extract information or meaning from the data and hence calculated thresholds for each sensor data to correspond toanactivity.

For example in case of an accelerometer we had a lot of raw data that was coming in terms of :

1. x : force acting on the x axis minus the gravity
2. y : force acting on the y axis minus the gravity
3. z : force acting on the z axis minus the gravity

These values are raw data and correspond to no unique activity. We proceeded to calculate the g_force on the object using the x, y and z value which basically corresponds to the force acting on an object in the x, y, z plane minus the gravity and reveals the push on it.

g_force = (x)^2 + (y)^2 + (z)^2 / (9.8)^2

The above is the actual true force acting on the body. Once we calculated the threshold we wanted to indentify specific incidents that correspond to certain thresholds of G force. Without any reference , we faced major testing cycles to calculate the most optimum value of g_force for specific incidents. We identified three scenarios for which we aimed to calculate the most optimal threshold. After extensive testing of about 30 hours we came down to the following values for our three scenarios.

1. Accident (car) x>= 3.0 (G_Force)
2. Jerk 1.2 <= y >= 2.1
3. Walking .8 < z > 1.1

where :

1. x – Accidents
2. y – Jerk
3. z – Walking.

Thus we resolved to some optimum values for three scenarios which our use cases utilized to reveal some information about the scenarios. Similarly we had to resolve most others sensors and attach specific incidents to them which our use cases utilized. We would like to clarify that even though we worked on the G_Force but that was to illuminate our use case and we do not aim to dwell our time extracting information from

these sensors but provide the sensors to all the users who might want to use it .

### 5.3 Query Optimizations

Our most important optimization to provide the sensor data in the most optimal way for the server to process it was to do query optimization to save on computational expense. We have our own four different optimization models which we devised to make the most optimum model. The models are as presented :

#### 5.3.1 'AND' - 'OR' Query Model

We have evaluated 'AND' – 'OR' models for analyzing queries of both types such as single query and multiple query over the remote server. Basically the thought process is that in single/multiple query scenario where we conjugate the inputs which in this case is the data coming from the sensors. We use Boolean operator 'AND' wherever necessary and 'OR' operator to evaluate the rest.

The logic for such an a analysis is that we calculate the first input of the multiple query and if :

1. In 'AND' query you calculate the first input and if it is false then we ignore the rest of the query no matter how long it is because now no matter what it is always going to be false. We tested this in comparison to our normal query evaluation where we calculate all the inputs one by one and then check the power consumption. Our analysis is based on both the test cases that we mentioned where we check using power tutor and physically checking the battery.
2. In 'Or' query you calculate the first term and in case that is true then we ignore the rest of the query because in all probabilities it is going to be true in any case and hence we dont need to calculate it. We compared this to our simple model where we compute all inputs .

| Avg. Power(mW) | Naive Query | 'AND' Query | 'OR' Query |
|---|---|---|---|
| Samples = 10 | 252.46 | 158.68 | 201.56 |
| Avg. Power(mW) | Naive | Optimized (independently) | Optimized (with MOD) |
| Samples = 10 | 245.68 | 105.56 | 218.56 |

Our analysis using power tutor and physically checking the batter proved that in case of :

1. 'AND' query : We achieved a result which was 60 % more computationally efficient than the normal evaluation using all inputs of the query.
2. 'OR' query : We achieved 12-15 % accuracy on combining the results from both the power tutor and physically checking the battery test. Our analysis is because OR is less restrictive than And query therefore the power consumption reduced by way less margin.

#### 5.3.2 'Most Economical Sensor First'

The next optimization that we build over our 'AND' and 'OR' query is that we use the same model but now while calculating the first input we use the most cheapest sensor in terms of power consumption. During our testing phase we tested all the sensors available exhaustively and found out that GPS and accelerometer have the most power consumption followed by our other sensors. We therefore use the same precedence model that we have recorded from our test to calculate the input from the most cheapest sensor in the query to calculate it and in case it is in the 'AND' or 'OR' model then we can safely ignore the rest thus giving us reduced power consumption using both our models.

E.g – Battery > 10 AND GPS < 32.5, 45.8 AND accelerometer > 12.3 , 3.4 ,7.8

We evaluate all using AND operator and then we calculate the first term initially , which in

this case is the battery and since it is the cheapest that will require the least computational expense. This can easily be followed by ignoring the rest of terms given that Battery is less than a threshold asked in the query itself. Therefore giving us a power efficient model .

#### 5.3.3 MOD Query

The third and the final optimization in our project was calculating the mod value of all the samples that we took for a specific sensor. In S2AS we provide the opportunity to the users to give in the number of samples they want to use to generalize there results or be very specific. We figured out that the accuracy of our use cases are going to vary as per the number of samples we look over. The specific scenarios for which we might need more samples is the case of a person walking where we would need more number of samples over a period of time to confirm his activity but in the other case of maybe a battery check or jerk we would need very specific samples as over a brief period the values might normalize themselves and we would actually be returned with nothing meaningful.

The logic that we employ is that we take up n number samples as requested by the user and then add their absolute values which gives us the numerator. We divide this by the number of samples that we have with us now. This gives us an average value over the number of samples or in a given amount of time thus giving us a more general setting than more specific setting.

Average Value = Sum of samples/ Number of samples

## 6. Results

The results that we were returned during and on concluding our project were very satisfactory and as per what we had in our mind as we set out working on our project.

The results can be compiled in serial order as :

1. We were successfully able to achieve a performance boost by offloading our analysis and processing of the data to our server, which was remote. Our performance increase as compared to the normal power consumption was minimal and not very huge but there still was some kind of performance increase.
2. Our optimization over the query analysis improved our performance by a huge margin. We were able to achieve sizeable performance margins using our models of query analysis, 'AND' and 'OR' queries gave us performance boost of up to 60 % and 12 % . We coupled that with our cheapest sensor first model to give us additional performance increase of 10 % .
3. We were able to provide three uses cases to justify the use of our platform for use in various projects. We have identified various other use cases that can actually use our project and API once it is built on top of our platform. These use case range from very simple applications to very broad sophisticated ones as described in our Use Cases section.
4. We were able to achieve an accuracy of about 77% over all three test cases we implemented using up to 40 samples per test cases and exhaustive testing.
5. We have used Amazon cloud services in terms of amazon S3, Cloud Front, Amazon EC2, Amazon RDS database for storing data in the MySQL database that we have in our instance. He most positive thing about this feature was that we could scale up and down almost at need with the infrastructure. We had unlimited amount of storage and infrastructure at our disposal as we needed it.
6. Though as we would like to make it clear we are not focusing at all on the security of the app and we assume that the user who has registered with our service is ready to provide his sensor information at all times to anyone who requests it, we also have provided the user with the option to block his sensor service so that in any case when he doesn't want to stream his sensors, he can stop it manually.

## 7. Testing

Various aspects of the application have been tested thoroughly. Testing could be broadly

divided into few categories. These include client-side testing, sending data to RDS, back-end processing, Twilio messages and calls.

One of the challenges we faced was that we could not check sensor and battery level readings on the emulator. The testing had to be done on a real android phone.

As and when data was being sent to the RDS, it had to be taken care that processing was performed on the latest data. For this, we passed the time-stamp of the readings along with the sensor reading values. This way analysis is accurate since alerts (calls and messages) are sent immediately.

The G-Force formula, which is used, is originally used to detect real-time accidents. Hence, the G-Force threshold to detect such impacts is very high. For our testing purposes, we had to reduce this threshold to an optimal value of 1.4 from 3.5, so that small jerks could be detected. Doing multiple experiments and trying with jerks of different magnitude led us to choose this value. We had to choose a threshold value such that detection of very small jerks could be avoided.

This optimal value was good enough for small jerks such as dropping the phone from a certain height. This greatly helped in testing because it would have otherwise been impractical to create a real accident.

We repeated the experiment about 50-60 times to test if an accident/jerk is being detected correctly. For this, we had to observe the RDS tables and check if the data is updated correctly and if the analysis on that data satisfied the necessary criteria of an accident/jerk.

The code for sending messages and calls using Twilio API was tested by sending SMS messages and calls to friends and team members. Friends of each teammate were registered as a contact for that teammate who was currently registered with the app. This way, we could make sure if alerts are going to the correct contacts and not to everyone, whose details exist in the database.

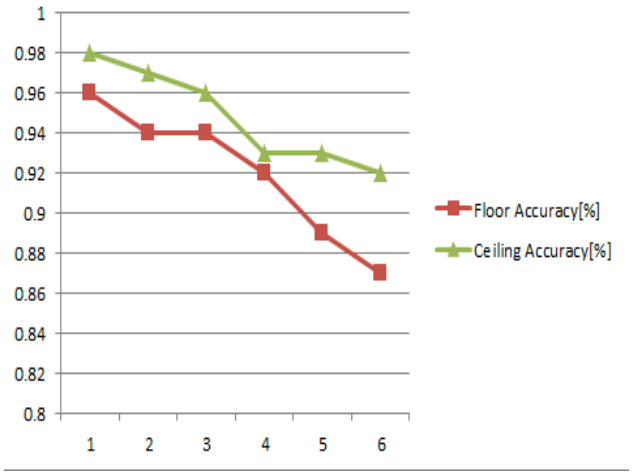
The user's contacts get notified when his phone discharges below a certain threshold. For this, the value chosen was 10%. This part of testing could not be done on the emulator, since the battery of the emulator is always set to 50%. We used a real android phone to do this testing and decreased its battery below 10% and observed the population of the data in the RDS tables. This threshold was changed to 20% and 30% and it was made sure that it works for any value.

**7.1 Accuracy**

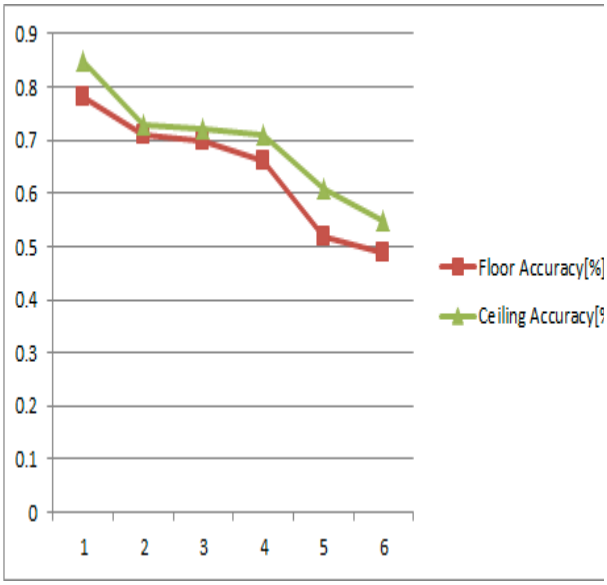
We evaluated our project for the accuracy of our test cases in terms of the number of samples that we took vs the particular use case that we tested for . We have provided our results in a graph structure that will clearly display what we tested for:

Below are the two diagrams representating the accuracy percentages of our two use cases. The first one is the detection of jerks and the second is the detection of low battery. These accuracy is the number of success ratio to the number of samples.

**Use Case 1 – Detection of Jerks :**



**Use Case 2 – Low Battery Alert**




**Android App Testing**:

We have tested the app with a number of samples and our own versions of testing which include both power tutor and battery testing. As and when data was being sent to the RDS, it had to be taken care that processing was performed on the latest data. For this, we passed the time-stamp of the readings along with the sensor reading values. This way analysis is accurate since alerts (calls and messages) are sent immediately.

The G-Force formula, which is used, is originally used to detect real-time accidents. Hence, the G-Force threshold to detect such impacts is very high. For our testing purposes, we had to reduce this threshold to an optimal value of 1.4 from 3.5, so that small jerks could be detected. Doing multiple experiments and trying with jerks of different magnitude led us to choose this value. We had to choose a threshold value such that detection of very small jerks could be avoided.

This optimal value was good enough for small jerks such as dropping the phone from a certain height. This greatly helped in testing because it would have otherwise been impractical to create a real accident.

We repeated the experiment about 50-60 times to test if an accident/jerk is being detected correctly. For this, we had to observe the RDS tables and check if the data is updated correctly and if the analysis on that data satisfied the necessary criteria of an accident/jerk.

The code for sending messages using Twilio API was tested by sending SMS messages to certain users/subscribers of the application.When people install the android app,their names and numbers get uploaded to Amazon RDS which is pulled by the Django Server to be shown to the client. Twilio messages are then sent to these numbers in case the clients want sensor information from them.

The user's contacts get notified when his phone discharges below a certain threshold. For this, the value chosen was 10%. This part of testing could not be done on the emulator, since the battery of the emulator is always set to 50%. We used a real android phone to do this testing and decreased its battery below 10% and observed the population of the data in the RDS tables. This threshold was changed to 20% and 30% and it was made sure that it works for any value.

## 8. Concluding Remarks/Future Work:

With the growing popularity of smartphones providing the sensors information as a service on the Cloud could provide app developers with endless opportunities to develop interesting and useful use cases.

A thing that needs to be kept in mind is although making data available on the Cloud is beneficial in so many ways it comes with the huge responsibility of keeping that data secure. The availability of the data should be controlled such that the privacy and the security of the user are in no way compromised.

Certain measures that Batsignal can take in this direction are as follows: Login feature should be included in the app so that only the user can select the alert services and his emergency contacts, and so that the supplied information cannot be

changed by anybody else. A must add feature is seeking permission from the person who has been registered as someone's emergency contact, to send alerts. This would make sure that the app is not misused to unnecessarily flood someone's phone with messages and calls.

Apart from security certain other features can be added to ensure that Batsignal is one of the sought after apps on the app store. For example in the wreckwatch alert apart from the accelerometer data we can also use microphone data to check for heightened noise levels after the high G force was detected by the server. If that were to be recorded then it could be told with higher certainty that the user might have met with an accident. Another feature that could be added would be to record the user's GPS data find out his exact location and along with a call to the emergency contacts also a make a call to the nearest medical help available.

Although it might seem like similar apps have come into the market before, Batsignal sets itself apart by keeping all of its processing at the server. So even if the device undergoes some damage, the alerts would still reach the enlisted contacts. For instance we can add a feature that if no readings are obtained from the user's phone for a certain amount of time then a different alert be issued. In all the available apps since the processing is all done on the phone the working of the app ends with the non-availability of the device over a network.

As mentioned earlier, a lot can be done with sensor data being made available as a service and as long as the necessary security precautions are considered very useful use cases can be implemented.

## 9. References:


[1]. WreckWatch: Automatic Traffic Accident Detection and Notification with Smartphones

http://www.dre.vanderbilt.edu/~jules/wreckwatchj.pdf

[2]. Stack Overflow – A lot of difficulties that we faced while programming in android were cleared by referring to this website.

It was also helpful while programming the backend where the framework Django needed to be used just to connect to the database and process its value.

www.stackoverflow.com

[3]. Sandy Walsh (Blog) – This was referred to understand how to use Django framework for a Stand-alone application.

http://www.sandywalsh.com/2010/05/packaging-django-application-as-stand.html

[4]. Django – The official website for the framework.

https://www.djangoproject.com/

[5]. Django Databases – The official website which contains the documentation on how to use the database part of Django

https://docs.djangoproject.com/en/dev/ref/databases/

[6]. Standalone Django scripts – Referred to study how stand-alone scripts can be run using the django framework

http://www.b-list.org/weblog/2007/sep/22/standalone-django-scripts/

[7]. Amazon Relational Database Service (RDS) Documentation – Referred to understand connections to RDS from an app

http://aws.amazon.com/documentation/rds/

[8]. Servlets forum at JavaRanch – Referred to understand how sensor data can be sent as JSON objects to a Java Servlet.

http://www.coderanch.com/t/366326/Servlets/java/Writing-JSON-data-client

[9]. Amazon Web Services Documentation – Referred to understand how to deploy the back end on Amazon EC2.

http://aws.amazon.com/documentation/

[10]. Sensors Overview ,Android Developers - Referred to understand the working of sensors in Android.

http://developer.android.com/guide/topics/sensors/sensors_overview.html

[11]. Twilio Docs – Referred to understand how the API can be used in BatSignal.

http://www.twilio.com/docs

[12] Power Tutor Documentation –

http://ziyang.eecs.umich.edu/projects/powertutor/documentation.html