

# SUTD 2021 50.003 ESC Final Report C3G7

Team Members:

Darren Loh Zhao Ying

Harshit Garg

Seah Zen Yi

Dong Jiajie

James Raphael Tiovalen

# Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>Requirements</b>	<b>3</b>
<b>Design</b>	<b>10</b>
Class Diagram	10
Use Case Diagram	11
Sequence Diagram	12
Overall Flow	12
Registration	13
Login	14
Mapping	15
<b>Implementation Challenges</b>	<b>16</b>
Database Challenges	16
Mapping Challenges	16
Algorithmic Challenges	17
Random Forest	17
KNN (K-Nearest-Neighbor)	18
Convolutional Neural Network	19
Engineering Challenges	20
Testing Challenges	21
<b>Testing</b>	<b>21</b>
Algorithmic Testing	22
Random Forest	22
KNN (K-Nearest-Neighbor)	22
Convolutional Neural Network	23
User Interface Testing	25
Login Feature	25
Image Upload Function	27
Image Upload Process	27
Firebase/Cloud and Authentication Testing	28
WiFiDataManager Testing	29
Integration/System Testing	30
Robustness Testing	30
<b>Lessons Learnt</b>	<b>31</b>
<b>Source Code Deliverables</b>	<b>32</b>

# Requirements

ID	FindMyTag_UC_01
Name	Area Mapping
Objectives	Accurately mapping the location on the phone and sending it to database to triangulate the location
Pre-conditions	<ol style="list-style-type: none"><li>1. Phone must accurately map the location.</li><li>2. Phone must be able to send data to the database.</li><li>3. The various Wi-Fi APs must be up and available to connect to.</li></ol>
Post-conditions	<p>Success:</p> <ol style="list-style-type: none"><li>1. Area mapping is successful, and the correct data is sent to the database.</li></ol> <p>Failure:</p> <ol style="list-style-type: none"><li>1. Area mapping has failed to identify the locations.</li><li>2. Area mapping has succeeded but failed to send the data to the database.</li></ol>
Actors	<p>Primary:</p> <ol style="list-style-type: none"><li>1. Wi-Fi APs</li><li>2. Mobile Application</li></ol> <p>Secondary:</p> <ol style="list-style-type: none"><li>1. Database</li><li>2. User</li></ol>
Trigger	User presses the 'Map' button.
Normal Flow	<ol style="list-style-type: none"><li>1. User uploads the floor plan onto the database.</li><li>2. Once uploaded, the user may proceed to the starting location.</li><li>3. When ready, press the 'Map' button and move in a line through the doors.</li><li>4. The devices then sense the Wi-Fi signal strength from the different APs connected and conclude the location of the user using triangulation method.</li><li>5. The user can pause, choose to re-map, or proceed to map it onto the floor plan.</li><li>6. A success message is shown to the user, prompting the user to continue.</li><li>7. The user may proceed to another area of the</li></ol>

	floor to continue mapping.
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>Mapping has failed and an error message is shown to the user on why it failed. <ol style="list-style-type: none"> <li>Wi-Fi has failed to detect the location; use case concludes with error message and prompt to re-map.</li> <li>Uploading to database has failed; use case concludes with error message for user to temporarily keep data locally and fix the database.</li> </ol> </li> </ol>
<b>Interacts With</b>	Notify User, Data Collection, Wi-Fi Information Collection use cases.
<b>Open Issues</b>	<ol style="list-style-type: none"> <li>How detailed should the error message be? <ol style="list-style-type: none"> <li>Inform user ways to debug.</li> <li>Simple troubleshooting guide.</li> </ol> </li> <li>Will there be an admin and common user separation? <ol style="list-style-type: none"> <li>Required if other users are expected to access the mapped data but cannot upload the data.</li> </ol> </li> </ol>

<b>ID</b>	<b>FindMyTag_UC_02</b>
<b>Name</b>	Data Collection for Database
<b>Objectives</b>	To collect and record the location data in a database.
<b>Pre-conditions</b>	<ol style="list-style-type: none"> <li>Internet connection must be sufficiently strong.</li> <li>The database must be connected to the internet and ready to receive the data.</li> <li>Data must have been collected through the application.</li> </ol>
<b>Post-conditions</b>	<ul style="list-style-type: none"> <li>Success <ol style="list-style-type: none"> <li>Places visited are recorded and can be checked any time.</li> </ol> </li> <li>Failure <ol style="list-style-type: none"> <li>Location data collection is a failure.</li> <li>Location data collection succeeded but information is not stored in the database.</li> </ol> </li> </ul>
<b>Actors</b>	Primary:

	<ol style="list-style-type: none"> <li>1. Database</li> <li>2. Mobile Application</li> </ol> <p>Secondary:</p> <ol style="list-style-type: none"> <li>1. User</li> <li>2. Wi-Fi APs</li> </ol>
<b>Trigger</b>	User presses 'Upload' to send the data to the database.
<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. After mapping has been completed in the app, User presses the 'Upload' button.</li> <li>2. The app does a primary check on if there is an existing local data before requesting a connection with the database.</li> <li>3. The database authenticates the app before receiving the data from the app.</li> <li>4. Data received is then prepared and put into the cloud function to determine the user location using the algorithm.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>1. User is unable to connect to the internet; use case concludes with error notification to the user.</li> <li>2. App is unable to contact the database; use case concludes with error notification to the user.</li> <li>3. Database fails to authenticate the app; use case concludes with error notification to the user.</li> <li>4. Database fails to receive the data from the app; use case concludes with error notification to the user.</li> </ol>
<b>Interacts With</b>	Notify User, Area Mapping use cases.
<b>Open Issues</b>	<ol style="list-style-type: none"> <li>1. How will the data be prepared for the cloud function after getting taken from the app?</li> <li>2. How will we fetch the data stored locally after prior failed attempts?</li> </ol>

<b>ID</b>	<b>FindMyTag_UC_03</b>
<b>Name</b>	Wi-Fi Information Collection
<b>Objectives</b>	Collecting location information using the Wi-Fi
<b>Pre-conditions</b>	<ol style="list-style-type: none"> <li>1. Wi-Fi must be enabled.</li> <li>2. Signal strength must be sufficiently strong.</li> </ol>

<b>Post-conditions</b>	Wi-Fi information collected upon request.
<b>Actors</b>	Primary: 1. Wi-Fi APs  Secondary: 1. Mobile Application
<b>Trigger</b>	User presses the 'Map' function on the application.
<b>Normal Flow</b>	1. The Wi-Fi emitter sends the signal to the device consistently. 2. The signal strength is then used by the device for mapping purposes.
<b>Alternative Flow</b>	1. One of the Wi-Fi emitters cannot be detected; use case concludes with slightly less accurate data collected.
<b>Interacts With</b>	Area Mapping use case.
<b>Open Issues</b>	1. (Alternative Flow 1) Might need an initial set-up to detect all possible Wi-Fi APs or beacons. 2. How will we troubleshoot the aforementioned issue?

<b>ID</b>	<b>FindMyTag_UC_04</b>
<b>Name</b>	Notify User
<b>Objectives</b>	From the collection of errors, pick out ones that suits the error that the user is facing
<b>Pre-conditions</b>	An error has occurred in one of the use cases.
<b>Post-conditions</b>	Error has been rectified or the user can troubleshoot the issues.
<b>Actors</b>	Primary: 1. Mobile Application  Secondary: 1. User 2. Database
<b>Trigger</b>	A connection error or data collection error in other activities has occurred.
<b>Normal Flow</b>	1. The activity that receives an error sends the

	<p>error code to the overseer.</p> <p>2. From the collection of error codes, a most suitable one is sent back to be displayed.</p>
<b>Alternative Flow</b>	NIL
<b>Interacts With</b>	Area Mapping, Data Collection and Wi-Fi Information Collection use cases.
<b>Open Issues</b>	1. How will we further develop this use case?

<b>ID</b>	<b>FindMyTag_UC_05</b>
<b>Name</b>	Data Testing
<b>Objectives</b>	From the data collected in the prior use cases, test the accuracy of the data.
<b>Pre-conditions</b>	<ol style="list-style-type: none"> <li>1. The collected data exists.</li> <li>2. The data has been mapped.</li> </ol>
<b>Post-conditions</b>	<ol style="list-style-type: none"> <li>1. The data collected during mapping is accurate and usable.</li> <li>2. Data accuracy has an error allowance of approximately 2~5m.</li> </ol>
<b>Actors</b>	<p>Primary:</p> <ol style="list-style-type: none"> <li>1. User</li> <li>2. Mobile Application</li> <li>3. Database</li> </ol> <p>Secondary:</p> <ol style="list-style-type: none"> <li>1. Wi-Fi APs</li> </ol>
<b>Trigger</b>	The user presses the 'Test' button on the mobile application.
<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. The user presses the 'Test' button on the app.</li> <li>2. Based on the Wi-Fi APs, the application fetches the floor plan from the database based on the initial estimated location.</li> <li>3. The floor plan is then displayed on the mobile app, with a dot to indicate the user location.</li> <li>4. The user can then choose to continue testing by pressing a 'Next' button to check his location at another spot.</li> <li>5. The data collected is accurate and the testing concludes.</li> </ol>

<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>1. There is no initial data collected from Area Mapping use case; use case concludes with an error notification to the user.</li> <li>2. The data collected is inaccurate; use case concludes, user can attempt to re-map the floor.</li> <li>3. There is trouble fetching data from the database; use case concludes with an error notification to the user.</li> </ol>
<b>Interacts With</b>	Notify User, Data Fetching, Wi-Fi Information Collection use cases.
<b>Open Issues</b>	<ol style="list-style-type: none"> <li>1. Should we make it more user friendly by implementing extra buttons such as 're-map' during the testing phase?</li> </ol>

<b>ID</b>	<b>FindMyTag_UC_06</b>
<b>Name</b>	Data Fetching
<b>Objectives</b>	Fetch the data from the database and display onto the application.
<b>Pre-conditions</b>	<ol style="list-style-type: none"> <li>1. There must be existing data in the database.</li> <li>2. The application must be connected to the database.</li> </ol>
<b>Post-conditions</b>	The data fetched from the database is accurately displayed on the application for the user to view.
<b>Actors</b>	<p>Primary:</p> <ol style="list-style-type: none"> <li>1. Mobile Application</li> <li>2. Database</li> </ol> <p>Secondary:</p> <ol style="list-style-type: none"> <li>1. User</li> </ol>
<b>Trigger</b>	<ol style="list-style-type: none"> <li>1. 'Test' button is pressed.</li> <li>2. 'Pull' button is pressed</li> </ol>
<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. The user presses a button to try and pull the data from the database.</li> <li>2. The application then collects the current location data and sends it to the database.</li> <li>3. The database then proceeds to authenticate and send the data requested by the application.</li> <li>4. Upon receiving the data, the application then displays the data onto the application for the</li> </ol>



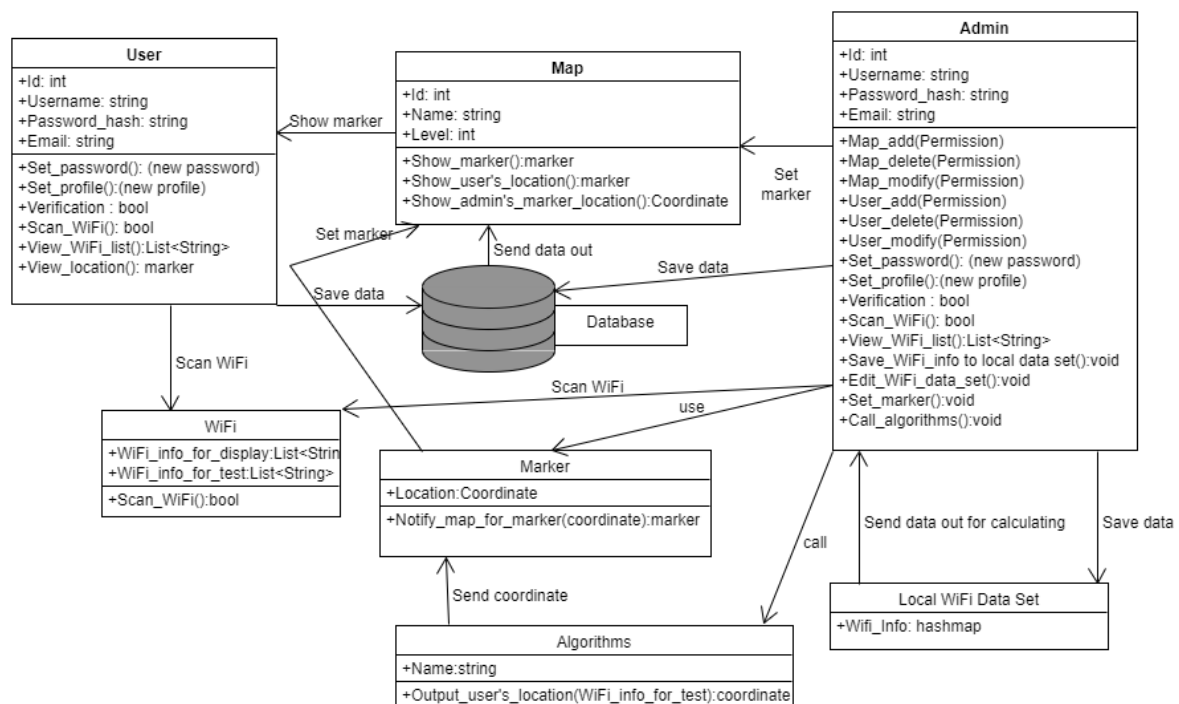
	user.
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>1. There is no initial data collected from Area Mapping use case; use case concludes with an error notification to the user.</li> <li>2. There is trouble fetching data from the database; use case concludes with an error notification to the user.</li> </ol>
<b>Interacts With</b>	Notify User, Data Testing use cases.
<b>Open Issues</b>	NIL

<b>ID</b>	<b>FindMyTag_UC_07</b>
<b>Name</b>	Hidden AP Search
<b>Objectives</b>	Search for hidden APs whose SSIDs cannot be searched by WifiManager scans.
<b>Pre-conditions</b>	<ol style="list-style-type: none"> <li>1. Location permissions must be allowed.</li> <li>2. Location must be turned on.</li> <li>3. Presence of Wi-Fi APs.</li> </ol>
<b>Post-conditions</b>	Hidden APs detected and also accounted for in our data collection and testing.
<b>Actors</b>	Primary: <ol style="list-style-type: none"> <li>1. Wi-Fi AP</li> <li>2. User</li> </ol> Secondary: <ol style="list-style-type: none"> <li>1. Database</li> </ol>
<b>Trigger</b>	When the 'Map' button is pressed, WifiManager starts to scan for Wi-Fi APs and collect the BSSID + level.
<b>Normal Flow</b>	<ol style="list-style-type: none"> <li>1. User presses the Map button.</li> <li>2. WifiManager starts scanning for Wi-Fi APs.</li> <li>3. Wi-Fi APs BSSID and level is collected and stored.</li> <li>4. Changes in level are stored and sent to the database for calculations.</li> </ol>
<b>Alternative Flow</b>	<ol style="list-style-type: none"> <li>1. User presses the map button.</li> <li>2. No Wi-Fi APs detected OR location settings are turned off.</li> <li>3. Notify users of failure.</li> </ol>

<b>Interacts With</b>	Notify User, Data Testing, Wi-Fi Information Collection use cases.
<b>Open Issues</b>	NIL

## Design

### Class Diagram

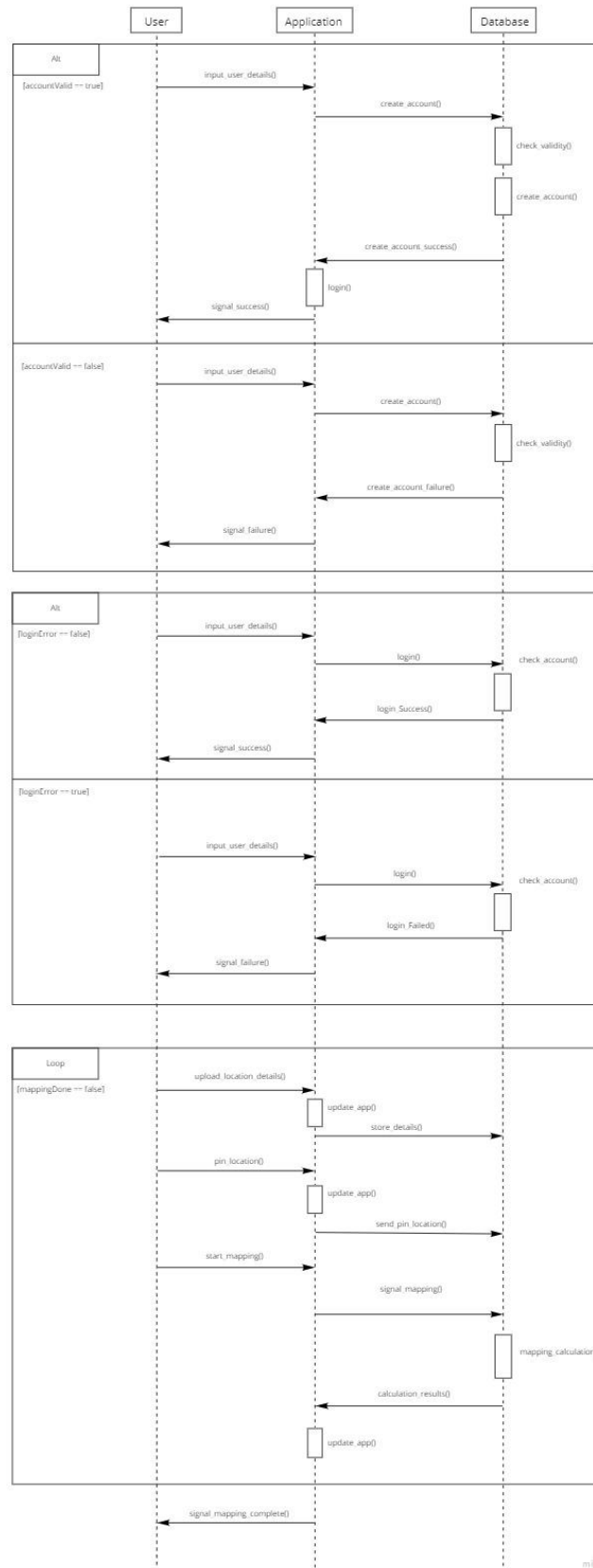


## Use Case Diagram

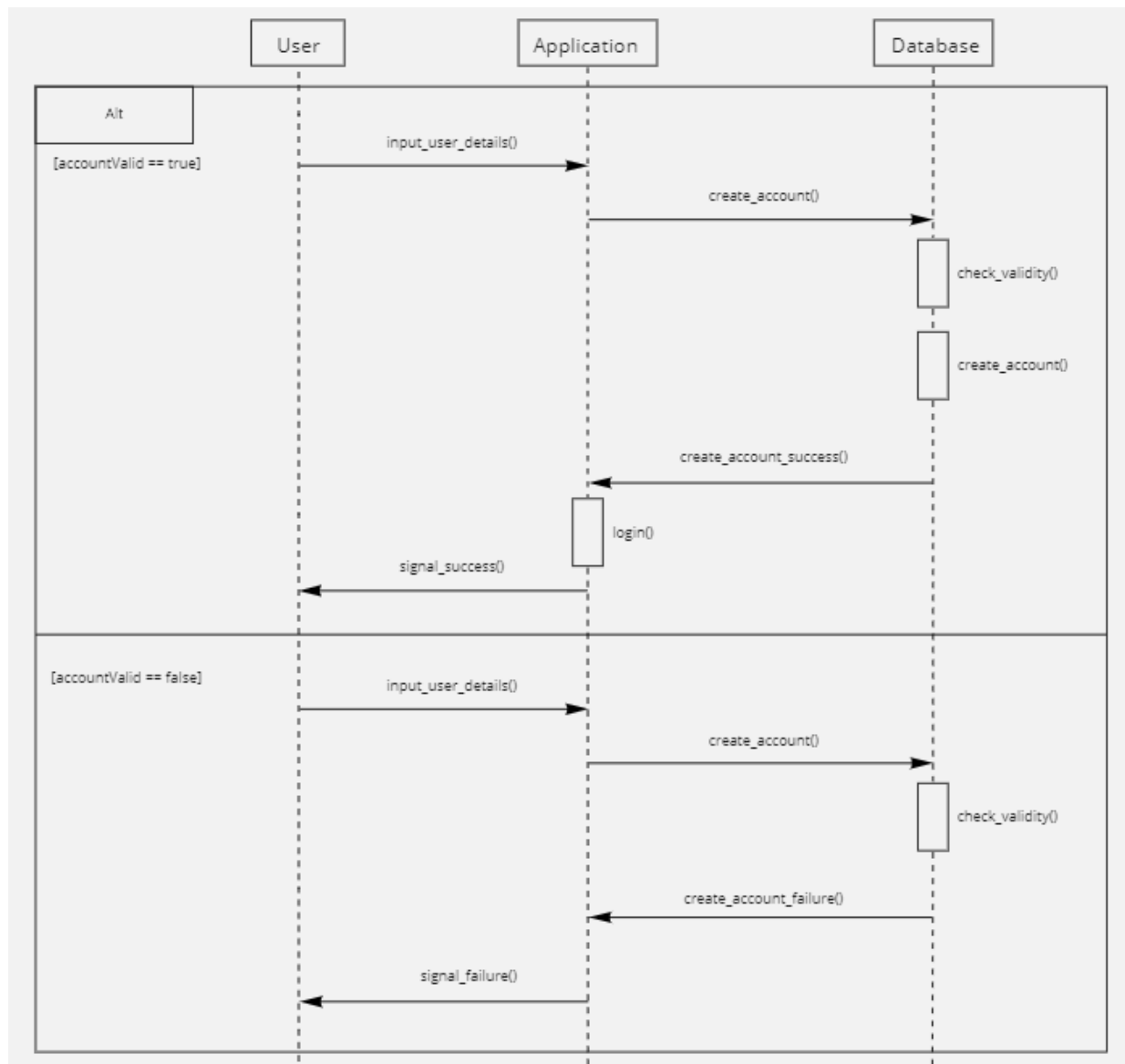


# Sequence Diagram

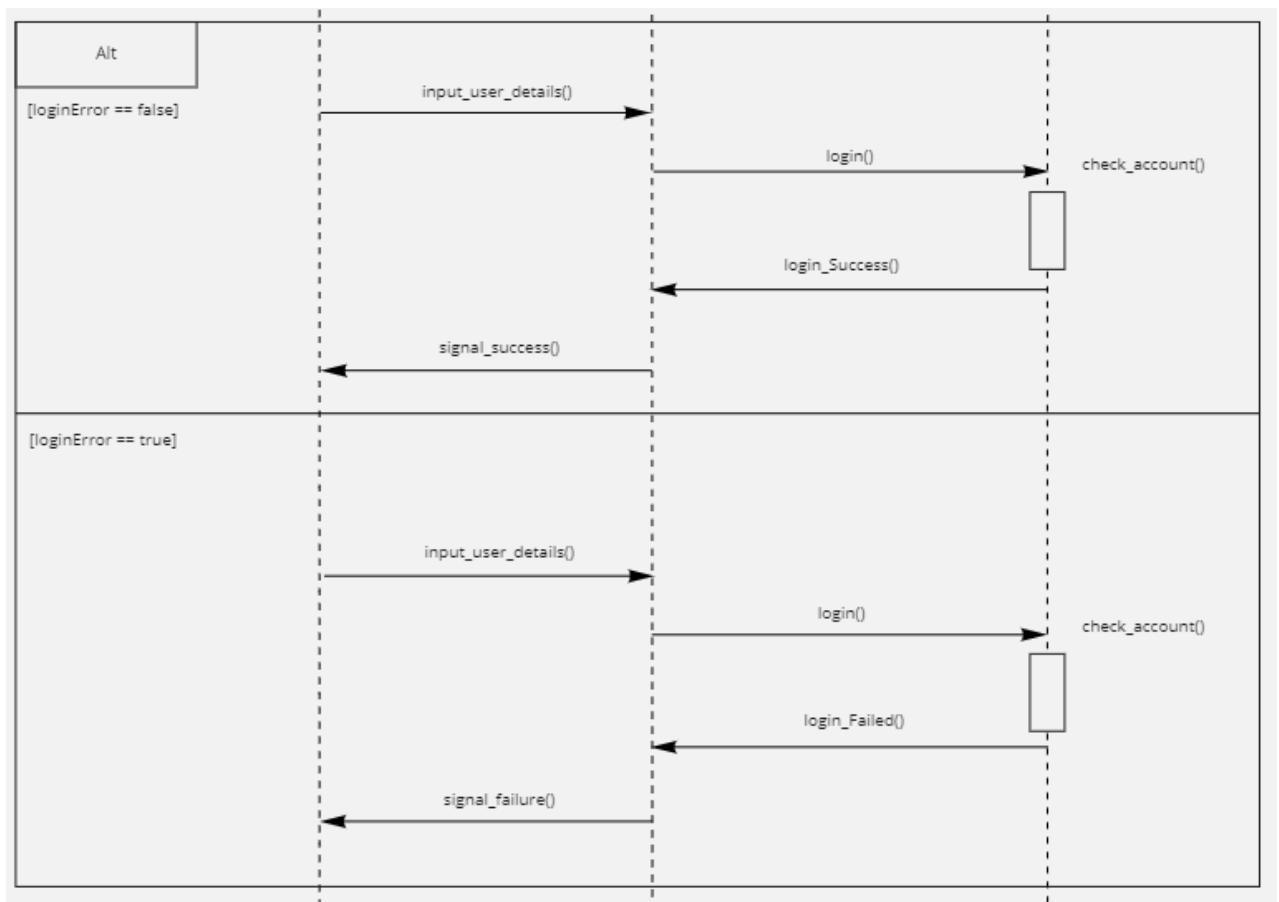
## Overall Flow



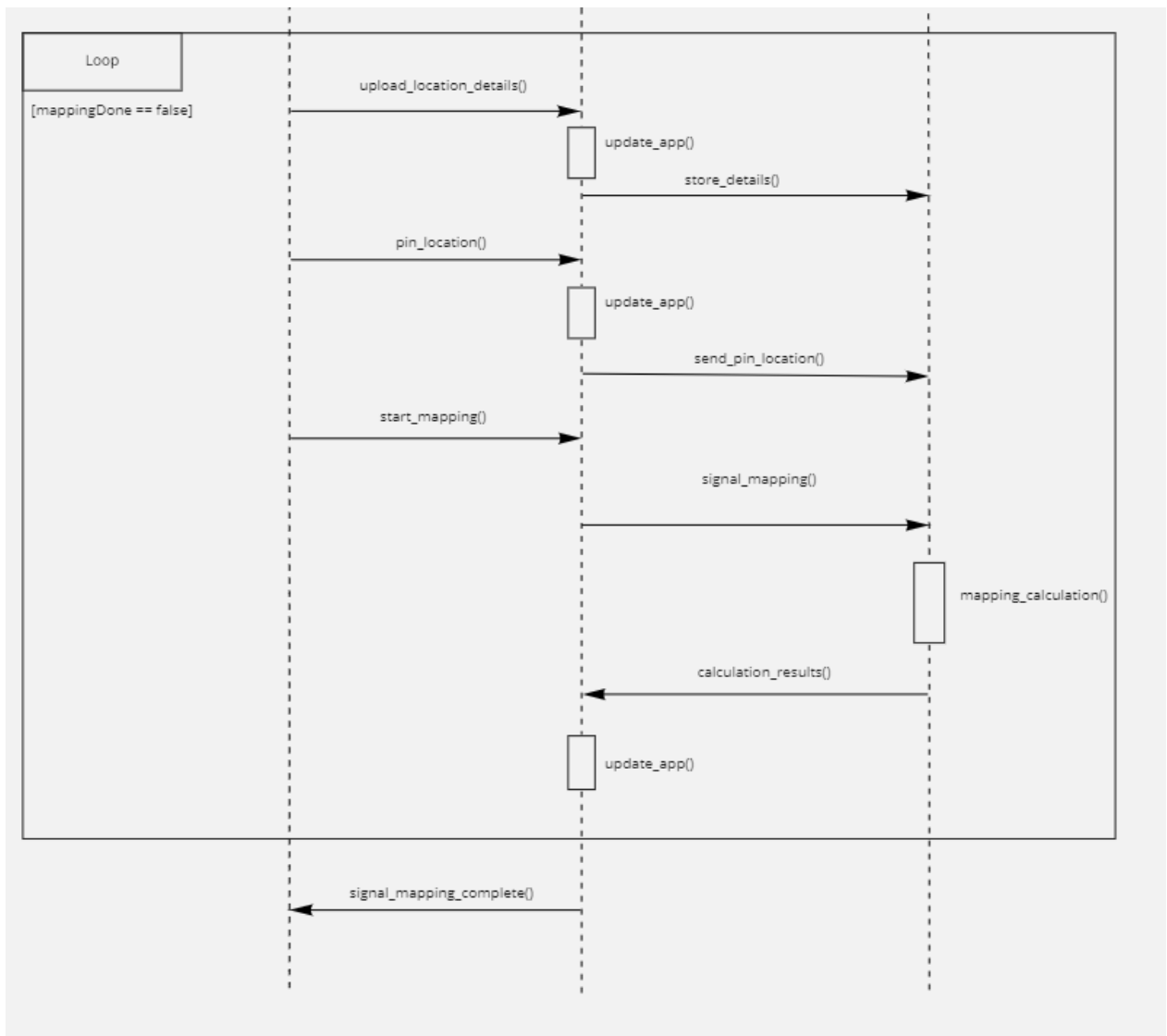
## Registration



# Login



## Mapping



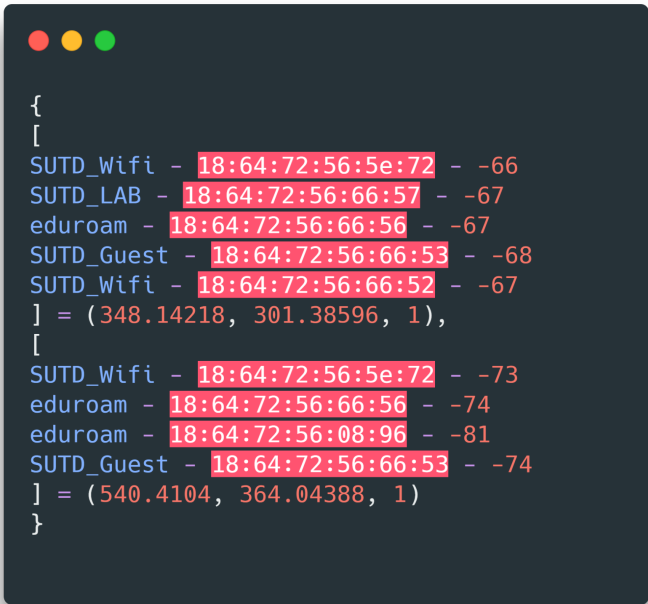
# Implementation Challenges

## Database Challenges

The plan was to connect to the Firebase Realtime Database and Google Cloud Services to the app. Following this initial plan, we used Google Authentication Services to make logging in and signing up a smooth and easy process. Furthermore, we decided to store user information like their names and mobile number in the Realtime Database, but that started giving a lot of errors. After a rigorous amount of work and searching, the error couldn't be resolved. After that, we decided to use Google Firestore instead of Realtime Database to store this information. The connection to the Firestore with the unique ID of the user, which is secured by the user authentication process, was implemented successfully. Then came the tedious task of storing the URL of the profile picture of the person in the database, but that caused several errors and we fixed it by storing the images in Firebase Storage in separate folders renamed by the unique ID of the person, which led us to easily import those pictures from the Android application. We followed the same with storage of map images. As these maps are unique to the person using it, it was easier to store them with the same unique ID.

## Mapping Challenges

In the initial state of mapping, we all agreed to use the common HashMap as the Wi-Fi information database and store it in this format: {[SSID-BSSID-RSSI]=(X,Y,Z)}, where Z refers to the floor. Here is a fragment of an example of real collected data:



```
{
  [
    SUTD_Wifi - 18:64:72:56:5e:72 - -66
    SUTD_LAB - 18:64:72:56:66:57 - -67
    eduroam - 18:64:72:56:66:56 - -67
    SUTD_Guest - 18:64:72:56:66:53 - -68
    SUTD_Wifi - 18:64:72:56:66:52 - -67
  ] = (348.14218, 301.38596, 1),
  [
    SUTD_Wifi - 18:64:72:56:5e:72 - -73
    eduroam - 18:64:72:56:66:56 - -74
    eduroam - 18:64:72:56:08:96 - -81
    SUTD_Guest - 18:64:72:56:66:53 - -74
  ] = (540.4104, 364.04388, 1)
}
```



This is not a problem for collecting data and the structure of data looks nice and clear. However, when we want to use that data to train the model or to predict the user's current coordinates, we find it very troublesome to separate the useful data from the HashMap structure.

For example, in the KNN algorithm implementation, we need to separate all BSSID into one ArrayList and all RSSI into another ArrayList and each of the entries are mapped to each other. Additionally, the file type that the Random Forest and Neural Network algorithms used is a .csv file. We took this for granted before actually implementing our algorithm. If we realized about this and fixed these problems before collecting and storing our data in multiple forms, it can save us a lot of time from data parsing.

## Algorithmic Challenges

The plan was to use Machine Learning libraries to train and test models entirely in Java, so that the Android app does not need to rely on any additional external services (which might add additional points of failure).

### Random Forest

Following the initial plan, the random forest algorithm was trained using an external library called Smile, together with a data handling library called TableSaw. This led to a lot of difficulties as not only was it difficult to implement purely from the documentation, there was very little support for the library found online. After some time of trying to get things to work, we found out that the two libraries were not suitable for training and predicting on a mobile Android phone.

After that, we decided to use another library to implement Random Forest for our mobile application. Upon researching, we found a machine learning library in Java called Weka. The attempt to use Weka and perform training and predicting was a much better experience compared to Smile, as there was a lot more support online. In addition to that, there are external versions of Weka created to suit Android programming. We were hopeful that the algorithm would work out, but then a bigger problem arose.

Our plan was to first train the model with pre-collected data, put it into our phone storage, collect the data from our application and use the model to predict the x, y and z coordinates to be displayed on the mobile application. However, the model requires an .arff file format to be loaded to predict. We attempted to bypass this issue by using the "Test" button in the testing fragment to scan for Wi-Fi and save in a .txt file. Then, the .txt file is read and parsed into the DataParser, which converts the data into the CSV file. This CSV file would then be converted to an .arff file using

classes CSVLoader and ARFFSaver. The ARFFSaver did not work due to the Weka version that was based off of the Weka Android. Since it was the latest Weka Android version, we could not find other versions that could possibly work. We bypassed this issue by manually writing the .arff file using a BufferedReader.

Ultimately, there was a major issue as the library class CSVLoader does not work on Android. This means that we have to either read the CSV file into the .arff file or directly translate the .txt file into an .arff file. At this point, we realise that it is an almost impossible task to train or predict on Android using Weka. Thus, we came to a decision to not include it in the mobile application, but still make use of the Weka GUI to train and predict the model for our RandomForest so that we can still compare the accuracy in the report.

This failure was due to our inability to gauge the difficulty in learning the libraries. The fact that we had faced many difficulties from the start should have been obvious to us that it will be almost impossible to implement using the library we used given the timeframe. The expectation that things will work out at the end was flawed and we should have been objective enough to change our plans to something that has more support and way easier to implement.

In the future should we encounter similar problems to when we implemented the random forest algorithm in Java, we will first do research on what would be the best method to perform localization. One such possible method is to train the model in Python, before deploying the model onto a cloud server and do our prediction on the cloud, and fetching the data to be displayed on our mobile application by opening up web server endpoints on the server.

## KNN (K-Nearest-Neighbor)

To get a good accuracy, the KNN algorithm actually needs to build a fingerprinting map database. Each fingerprint is 5 meters away from other points surrounding them. However, during the mapping period, it is hard to collect all the data to accurately reflect the actual coordinates due to the Campus Centre not having many notable references to help us mark correctly on the map. Additionally, since the density of the data is not equal, the KNN algorithm does not work with a high efficiency. This is because it is likely to happen that data points in all 4 directions or orientations of the user have a huge density difference. Hence, it becomes unfair when we calculate the average coordinates of those coordinates. This, in turn, makes the resulting coordinate to lean to the side that has a higher density.

To improve the accuracy, giving a penalty is one possible way (this is if we really cannot get a fingerprinting map database with an equal density map). In other words, if most of those K points are on one side, we give them a bit less weight when

calculating the average coordinate. But when implementing the penalty, it is hard to set different standards for different conditions properly, and hence the KNN algorithm implemented by us can satisfy the accuracy within 5 meters in some specific area (most of them have points with which no penalty is needed), but in some area that has a very biased density (like the corners or the edges of the building), it does not work very well.

## Convolutional Neural Network

For our final product, we followed the CNN algorithm implementation specified by [this research paper](#). It was a challenge to find an appropriate CNN algorithm to be implemented on an Android mobile device in the first place, since most academic papers that we could find either utilize information that are not normally available on Android (and require either some kind of low-level hacking to the Wi-Fi PHY layer such as [this](#)), require a lot of computational power (since the research papers utilize desktops and computers), require two separate instances of physical location visits (one for mapping and another one for testing, between which the data would be manually parsed, which does not satisfy our requirements of an “all-in-one package” Android app product) or they simply state a comparison/survey between algorithms that require some form of proprietary SDK or physical devices, which is unfeasible for our project.<sup>1</sup> Some research papers also seem to implement quite “arbitrary” decisions without any rationale, which is why we do not follow them. Further specific details on these arbitrary decisions, as well as some considerations and discussions regarding the rationale of why we do not follow the implementations of certain academic research papers, can be found in the code’s comments, particularly in the NeuralNetwork.java class file. To take a snippet from the discussion there:

*“This network uses RSSI instead of CSI information data to solve a fingerprinting classification problem. Based on [this research paper](#), using CSI data will result in relatively better and more accurate localization results. However, no current smartphone API allows access to the Wi-Fi AP’s physical layer CSI information (this includes the Android API). Some level of hacking to the lower layers could be executed but its legality is questionable. Manual offline mapping using [the Linux 802.11n CSI Tool](#) could be conducted but this is not entirely in line with the project’s requirements to actually conduct the mapping on-the-fly on the smartphone mobile device itself. Alternatively, [a more-recently published academic paper](#) proposed a HDLM-based model and established that it is much more performant (in terms of both accuracy and training/inference speed). However we decided*

---

<sup>1</sup> A list of all these research papers that we have encountered are available on our project repository’s main README file, for those who are curious and interested.

*against implementing the approach in said paper due to 2 main reasons:*

- The decision to scale the rectangles indicating the different Wi-Fi APs following a somewhat random ordering seems to be quite arbitrary.*
- The scaling of the boxes would inevitably lead to an inherent bias that leans towards those Wi-Fi APs later in the somewhat arbitrary ordering, and hence it is not that scalable to many and much more Wi-Fi APs."*

However, the research paper that we refer to is not without its own kind of limitations. First of all, the research paper has explicitly specified that the optimal values of the hyperparameters are highly dependent on the collected dataset. As such, this affects the actual accuracy of our physical testing since simply conforming to the specified hyperparameters in the research paper would most likely not work. Deciding on what the hyperparameters should be by doing physical testing is also rather time-consuming since it is more of a trial-and-error iterative process and a repetitive cycle of training and testing.

## Engineering Challenges

In the initial implementation of our application, we used image views to display the required floorplans. Therefore, our team made use of the Picasso dependency to aid us in setting the image into the ImageViews after querying from our database.

However, our team later replaced the image views with a new custom image view dependency that we found online that supported the zooming in of the set image. After implementing the new custom image views, we found out that using Picasso with the new image view introduces many new bugs.

Therefore, our initial solution to the problem would be switching the used dependency from Picasso to Glide. This new dependency eventually allowed us to successfully query our image from our database and set it into the new image view. However, we later also found out that although the use of the dependency allowed us to get the image from our database and set it into the image view, it also crashes when executing the same command for a second time.

We later found out that the reason for the error was because we were trying to query and load the image as a bitmap and loading the image as a bitmap again would return an error as there are some implementation issues inherent in the Bitmap's onDestroy() and onRecycle() methods. As such, we solved the issue by querying and loading the image as a File rather than a Bitmap, which eventually allowed for the successful implementation of the querying and loading of images from our database.

We also encountered several integration issues, especially since some of the dependencies are not compatible initially due to duplicate classes. We solved this by slightly downgrading the Firebase Firestore dependency version, as well as the ND4J dependency versions. As such, we are not using the latest version of the dependencies, but since they are working properly, we deem this as good enough.

## Testing Challenges

One of the main features of our application would be the uploading of floorplan images before we can start with the entire algorithmic procedure. As such, our group believed that it was imperative that we had to test it and get it to work. In order to test the image uploading function, we had to simulate the opening of the phone's gallery application, select an image, and see if the image properly displays onto the image view.

However, since our group uses the Espresso dependency for user interface testing, we found that it is impossible to simulate opening the gallery application to choose a random image from there since it involves leaving the current application to go to an external application.

Therefore, to counter the above mentioned problem, we decided on a two step solution. Firstly, since it is possible to test the changing of intents with Espresso, we tested that clicking on the image upload button would indeed trigger the gallery intent with the picking action "ACTION\_PICK". Secondly, we chose a resource that every android studio project would inherently contain, which is namely "ic\_launcher\_background". By doing it like this, we can effectively reach our intended testing goals with a different approach.

As mentioned in the previous point, our group's main methods for user interface testing is by using the Espresso dependency. However, we found out that Espresso does not inherently have any methods to check for Toast messages. Since the main testing method for the image upload function is the checking of toast messages, we encountered a problem. This problem is later solved by creating a new Toast matcher. The newly created matcher checks if the Toast message displayed on the application is the same as what we have specified in the test, which is also the message that we want to see displayed.

# Testing

## Algorithmic Testing

### Random Forest

With the failure in implementing the random forest algorithm into our mobile application, we decided that instead we will collect data at different points of the Campus Centre, place them into a text file, and feed the data to our random forest algorithm to predict the x, y and z values. Each coordinate x, y and z has their own model, and the model will predict the coordinates from the BSSIDs and RSSI values at each point. Since the actual coordinate values have already been collected, the model will calculate the difference between the expected and actual value predicted by the model, and give the resulting root mean squared error and root relative squared errors to show the accuracy of the model. The following is a prediction using a training set on our model.

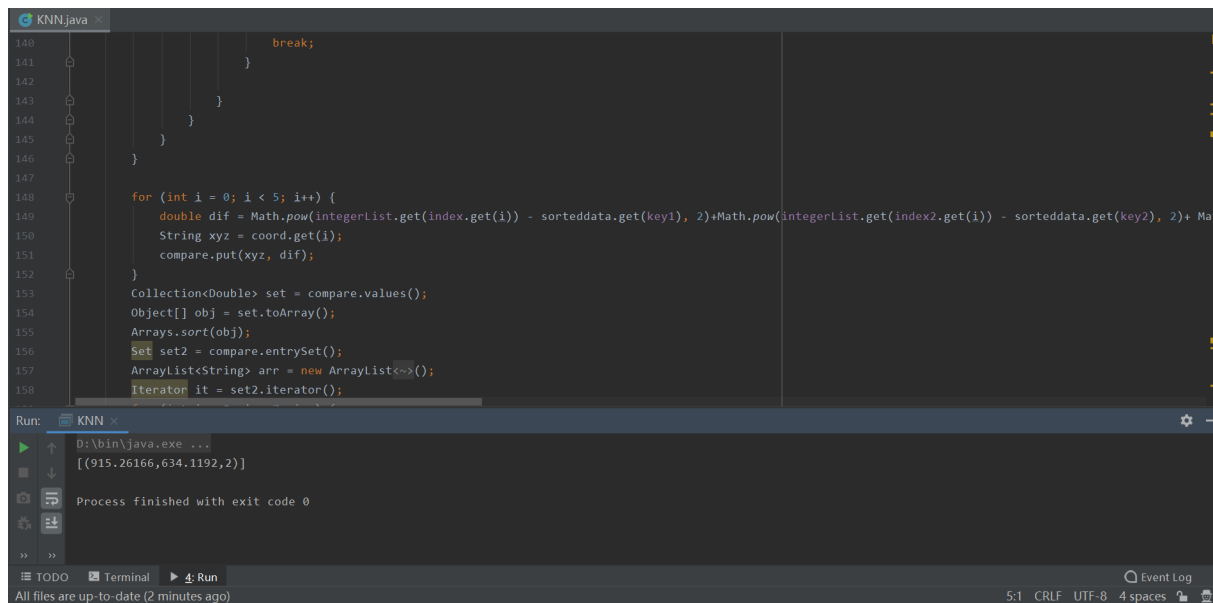
```
=== Summary ===
```

Correlation coefficient	0.9952
Mean absolute error	298.4949
Root mean squared error	387.7304
Relative absolute error	36.9162 %
Root relative squared error	38.9042 %
Total Number of Instances	77

### KNN (K-Nearest-Neighbor)

The way we test the KNN algorithm is to stand on a spot and scan the Wi-Fi information at that point (we use BSSID-RSSI pairs as inputs) and after calculating, a marker will pop up to indicate the user's predicted location on the map (a Toast message with the exact float-typed coordinates will also be shown for a short while). To get the relationship between coordinates and meters, we use the biggest x and y to divide by the length and width of the Campus Centre. By calculating the difference between these two coordinates, and transforming it into the physical unit of meters, we can get the accuracy of the algorithm.

Here is a test case for the KNN algorithm:



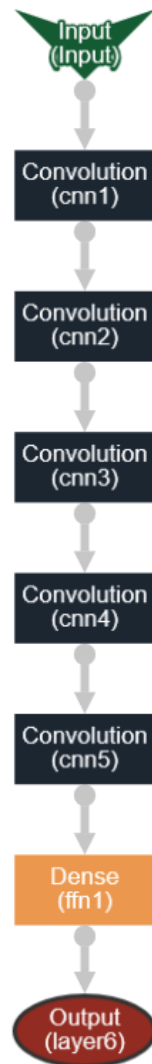
```
140         } break;
141     }
142 }
143 }
144 }
145 }
146 }
147 }
148 for (int i = 0; i < 5; i++) {
149     double dif = Math.pow(integerList.get(index.get(i)) - sorteddata.get(key1), 2)+Math.pow(integerList.get(index2.get(i)) - sorteddata.get(key2), 2)+ Ma
150     String xyz = coord.get(i);
151     compare.put(xyz, dif);
152 }
153 Collection<Double> set = compare.values();
154 Object[] obj = set.toArray();
155 Arrays.sort(obj);
156 Set set2 = compare.entrySet();
157 ArrayList<String> arr = new ArrayList<>();
158 Iterator it = set2.iterator();

Run: KNN
D:\bin\java.exe ...
[(915.26166,634.1192,2)]
Process finished with exit code 0
```

## Convolutional Neural Network

As mentioned in the previous 2 algorithmic testing sections, for our special situation, we only conducted algorithmic testings for the Campus Centre building. For the CNN algorithm, we implemented a simple feed-forward CNN model and attempted to closely mimic and follow the specifications defined in the research paper that we referred to. The only significant change that we implemented would be to consider 10 x 10 pixels for the input fingerprint images into the CNN model (instead of 7 x 7) since we are using 10 hardcoded AP BSSIDs. Due to this implementation of hardcoded APs, we can only test and ensure its reliability within the confines of the Campus Centre building. Other parts of the configuration are highly similar to the specifications prescribed in the academic paper (except for the hierarchical model of the multiple CNNs utilized, where we only implement the last part of that hierarchy to detect the x and y coordinates of the end user).

To recap, this is how we built the layers of the CNN model (one for each axis):

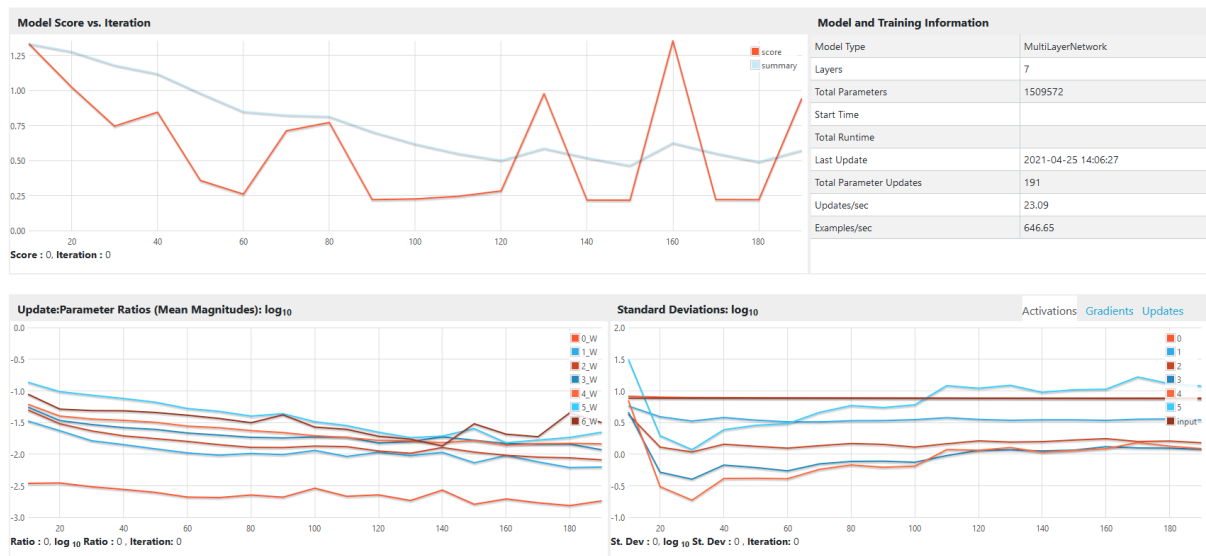


The CNN configurations are:

- Inputs are 2-dimensional array representations of 10 x 10 grayscale images of the pre-processed Wi-Fi AP fingerprints. Grayscale would imply one channel.
- 5 hidden convolution layers, each using the ReLu activation function.
- Filter/kernel size is 2 x 2.
- Stride size is 1 x 1.
- Adam algorithm updater.
- 32 layer size/number of filters for the first convolution layer, which doubles for subsequent convolution layers until it reaches 256 number of filters for both layer 4 and layer 5.
- 100 classes for the output since we split each axis into 100 classes/grids.

We found out that 12 epochs of training would allow us to achieve the best accuracy (of around 50-75% for a split dataset of 80% training data and 20% testing data), both for the x-coordinate CNN and the y-coordinate CNN for both level 1 and level 2:





Some statistics for the best model for the x-coordinate:

```
=====Evaluation Metrics=====
# of classes:      100
Accuracy:          0.7500
Precision:         0.7500      (99 classes excluded from average)
Recall:            0.3333      (97 classes excluded from average)
F1 Score:          0.8571      (99 classes excluded from average)
Precision, recall & F1: macro-averaged (equally weighted avg. of 100 classes)
```

Some statistics for the best model for the y-coordinate:

```
=====Evaluation Metrics=====
# of classes:      100
Accuracy:          0.5000
Precision:         0.3333      (98 classes excluded from average)
Recall:            0.2222      (97 classes excluded from average)
F1 Score:          0.6667      (99 classes excluded from average)
Precision, recall & F1: macro-averaged (equally weighted avg. of 100 classes)
```

## User Interface Testing

For our user interface, we have three separate test cases which are the login feature, the image upload function and the image upload process. The dependency that we used to do user interface tests would be “Espresso”.

### Login Feature

For the login feature, the main things that we are looking out for would be that the user inputs the required fields correctly. To do this, we created 5 different mini test cases.

1. Empty email and password fields.  
This case would happen if the user leaves both the email and passwords fields blank. To check this, we would first simulate a button click on the login button. We would then check if the email field returns an error text "Email is Required.". If so, the test case would be a success.
2. Empty email field.  
This case would happen if the user leaves only the email field blank. Similar to the previous point, we would check this by simulating a button click on the login button and then checking if the email field returns an error text "Email is Required.". If so, this test case would be a success.
3. Empty password field.  
This case would happen if the user leaves only the password field blank. To check this test case, we would simulate having the email field correctly inputted but having the password field be left blank. Clicking the login button with these conditions would have the password field returning the "Password is Required." error text. If this error text is returned with the above stated conditions, the test case would be a success.
4. Password length incorrect.  
This test case would happen if the user inputs a password with a length smaller than 6. To check this test case, we would simulate having the email field correctly inputted but having the password field be input with a password with a length smaller than 6. The test case would be a success if the password field returns the error text "Password Must be >= 6 Characters" after clicking the login button with the above mentioned conditions.
5. Successful login.  
This test case simulates the successful login of a user if all the fields have been correctly inputted. To check this test case, we would simulate having both the email and password fields be correctly inputted. Upon clicking the login button under these conditions, we would check if the next activity displayed is the correct activity. If so, the test case would be considered a success.

If all the mini test cases pass, we can confidently say that our login feature works as intended.

## Image Upload Function

For the upload function, what we would be testing would be that clicking on the upload button would bring us to a gallery intent with the picking action and upon successfully picking an image, it would return to the activity with an "ACTIVITY.RESULT\_OK" result. In addition to that, we would also test that returning from the gallery image picking intent would populate the image view in the original activity with the selected image. In our case, we used a default image that is inherently in all android studio projects "ic\_launcher\_background". When both tests are successful, we say that the upload function is working as intended.

In our project, we have 2 buttons that use the image upload function. Therefore, the test is split into 2 mini test cases, each corresponding to the 2 buttons. If both the buttons pass the test specified above, we say that both buttons are working correctly as intended.

## Image Upload Process

In the image upload process, we are concerned with the correct uploading of images and its related name tags. Therefore, we will split the test into 5 smaller test cases. This test would be using Espresso to simulate all input and clicking processes. We also created a custom Toast matcher since our main testing method would be checking if the correct toast messages are shown at every mini test case. Thus, all inputs and clicks specified below are done through the use of Espresso functions.

1. All fields are empty.

The first test case happens when the user leaves all fields blank. To check this test case, we would simulate leaving all fields empty before clicking on the confirm button. Clicking on the confirm button with these conditions should return a Toast message "Please fill in Location 1 Name" which we would check with our custom Toast matcher. If the toast message shown in the application matches what we are looking for, we can say that the test case is successful.

2. Only the name tag of the first location is filled.

The second test case would happen when the user only fills in the name of the first location. To check this test, we would only fill in the name tag field of the first location before clicking on the confirm button. After clicking on the confirm button, we should see a Toast message "Please upload Location 1 Floor Plan" which we would check with our custom Toast matcher. If the toast message shown in the application is the same as what we are looking for, we can deem the test case a success.

3. Only the name tag and image of the first location is filled.

This test case would happen when the user only fills in the name and uploads the image for the first location. To check this test, we would first fill in the fields for the first location name tag and upload an image for the first location. Upon clicking the confirm button after filling in the fields, we should see the toast message "Please fill in Location 2 Name". If the toast shown on the application matches what we are looking for, we can say that this test case is successful.

4. Only the first location name tag, first location image and second location name tag is filled.

This test case happens when all fields are filled with the exception of uploading the second location image. To check this test case, we would first fill in all the previously mentioned fields before clicking on the confirm button. Next, a toast message should show up stating "Please upload Location 2 Floor Plan". This toast message on the application would then be compared to a message that we want and have planned beforehand. If both the messages are the same, we can say that this test case is a success.

5. All fields are properly filled.

This last test case aims to test for a successful image upload. To do this, all the required fields would be correctly inputted before clicking on the confirm button. Afterwards, we would check if we are at the correct activity that we intended to reach. If so, we can then say that the image upload is successful and this test case is passed.

If all of the mini test cases above are successful, we can say that this test is successfully completed where all our functions in this section are working as intended.

## Firestore/Cloud and Authentication Testing

For testing the cloud services such as authentication, Firebase Storage and Firestore Realtime Database, we had to be sure that using these services are safe and we don't have any issues or loopholes where the user either has some problems or his/her data is taken by a malicious party.

To solve this problem we used Node.js and dependencies like Mocha to test whether any other user can't read or detect different users data. We also checked the bruteforce attack and made sure that it wasn't possible, as the user ID is blocked and the IP address is detected for whoever is trying to attack the app. We also check that all the public information is accessed by any user, and the private information is just accessed or being written by the specific user. Further, we also tested that

certain information which is written by admin but seen by the public, and tested the write and read of such data.

Some tests initially gave some error, but were corrected by editing the JSON file and the permissions in the Cloud.

## WiFiDataManager Testing

The WiFiDataManager works in two places: Mapping and Testing (both fragments).

Mapping is used to collect the WiFiData for localization and at that part, we save the WiFiData in a String by calling "gets()". To test if that function is working, we used `assertTrue(!wifiDataManager[0].gets().isEmpty());`

If the return value of "gets()" is not empty, the test will pass, which means that one of the functions of WiFiDataManager is working.

During the Testing phase, the return value is a HashMap that has been filtered by WiFi SSID and RSSI values, so if we do not run the test in Campus Centre, the HashMap will be empty (due to unique hardcoded SSIDs). To test if that function is working, we tested it in two locations, in the SUTD hostel and another place far away from the Campus Centre, and we use `assertTrue(wifiDataManager[0].getsorteddata().isEmpty());`

Due to being away from Campus Centre, there is no qualified Wi-Fi data, so the HashMap will be empty and the test will pass. We also test it in the Campus Centre in a different section of the code. This is important because it should be working properly in the Campus Centre, which is the most important area for the purposes of our demo and final presentation. For this case, we use `assertTrue(!wifiDataManager[0].getsorteddata().isEmpty());`

In that case, although some Wi-Fi data is filtered, there is still some qualified data being put in the HashMap so the HashMap is not empty and passes the test.

It is also important to notice that users can manually deny the Wi-Fi scanning process through the Android Settings. To imitate that condition, we setWifiEnabled as false and check if the sorted data and the original collected raw data are both empty. These should pass the test if it is working correctly. For this, we use `assertTrue(!wifiScanner[0].getMacRssi().isEmpty());` and `assertTrue(!wifiScanner[0].getWifiDataAPs().isEmpty());`

Since all the tests above passed, we are confident to say that the WiFiDataManager is working properly.

## Integration/System Testing

For integration testing, we focus on the actual accuracy of the algorithms, as well as on whether they fulfill the requirements of the project. Since the “output” of the algorithms are not that straightforward to be automatically tested using tools, we decided to test the whole system manually by conducting physical testing in the Campus Centre level 1 and 2.

For the mapping component, we were able to upload floor plans to the Firebase Storage database (verified by checking and accessing the database manually) successfully. We were also able to train the algorithm models on the Android phone.

For the testing component, we were able to parse the collected data properly and pass it over to the algorithms to be processed. The algorithms were also able to spew out properly-formatted output and the Android application does not crash at all.

In our limited time, we attempted to collect some form of dataset for our algorithms’ perusal. However, after collecting this dataset and testing all of our algorithms, they do not satisfy the initial requirements of accuracy within 5 meters. For example, the CNN algorithm only achieves an accuracy of around 15-20 meters. Hence, and unfortunately, this was deemed to be a **failure** by the client.

To improve the accuracy of the algorithms, we can always collect more dataset or to tune and optimize the hyperparameters of the algorithms. While the latter one is more difficult, both are relatively time-consuming since it is more of a trial-and-error process (as well as referring to relevant research papers, if any).

## Robustness Testing

Finally, for robustness testing, we need to ensure that the Android application does not crash under a multitude of conditions. There are three things that we have tested under this section:

- Exceptional Circumstances Related to File Access

In certain cases, such as when there are no relevant filtered Wi-Fi AP data being collected (wrong physical area) or when the end user does not follow the normal flow of Mapping before Testing, the physical files that are needed for the algorithms to work properly might not be created or might be missing crucial details and information. If we do not implement enough checks, the Android application might crash. As such, we have added appropriate try-catch statements, as well as propagated the necessary exceptions and data up and down the hierarchical function call chains accordingly so that if certain conditions are not met, friendly warning Toast messages are shown to the user. One example would be when the user presses the Test button

before they ever pressed the Map button (with the Neural Network option selected in the algorithm choice's spinner). For this particular case, we displayed a Toast message to kindly remind or ask the user that the reason the localization prediction failed might be due to the fact that the User never mapped and trained the CNN model before.

- Dependencies

The algorithms that we employed for this project have various dependencies, some of which might conflict with each other. We have combed through the [MvnRepository website](#) to ensure that the dependencies and their versions do not conflict with each other. As such, the specific/special set of dependencies in the module's build.gradle file are the appropriate ones that will allow us to run everything smoothly without any weird or random crashes.

- Performance

The algorithms that we implemented for this project have a heavy amount of dependencies, especially the Convolutional Neural Network. As such, we need to ensure that the Android application UI remains functional and operational during both Mapping and Testing. For this, 2 phases of testing were conducted: training the model on a laptop CPU (instead of GPU, like how other neural networks are conventionally trained on), as well as training the model on an actual Android application. This is to simulate the constraint of limited available computing resources on the Android device. We were able to conduct the training on an ASUS ROG Zephyrus G14 GA401IV's CPU (AMD Ryzen 9 4900HS) quite quickly. On an Android device, the time required to train the model is more significant and the entire smartphone slows down for a while, but with enough patience and time, the model was able to be trained without crashing the Android app.

## Lessons Learnt

Obviously, there are some things that can be learnt from this project. Among other things, through this project, we have learnt that:

- We should always provide and conduct constant communication and feedback sessions with the client. This is to keep the client in-the-loop with the progress of the project, as well as to find out and understand what the client cares about the most.
- We should research even more to select better academic research papers, preferably peer-reviewed ones, with better assertions and assured reproducibilities on the algorithms' effectiveness and accuracy in our specific configuration (although there is a tradeoff between time spent researching and time spent doing, so balance is also needed especially in a time-constrained project such as this). More engineering effort could be

employed if the actual results do not match the paper and the algorithms should be tested as early as possible.

- We should start with small-scale testing first, such as by mapping and testing a small area of the Campus Centre first.
- We should utilize the availability of numerous localization API/SDK providers or other alternatives to ease the development of the algorithms.

We followed a combination of the **Agile Method** and the **Rapid Prototyping** software development methodologies and processes. There were significant challenges with following these processes, in particular due to the high workload of this SUTD academic term. Since we need to juggle and manage our work between other academic courses that we take during this term as well, it becomes harder and harder as the term progresses to keep up with any forms of pre-determined timelines such as Gantt charts or timetables. In addition, falling behind in one component will compound and roll up the amount of work to the next one, cascading the amount of workload into a huge giant mess at the end. In general, we have learnt that we should manage our time better and balance the amount of workload between the different modules properly. We should also not aim for too high or too ambitious of a goal, such as implementing certain algorithms from scratch, since the timeline that we had been given were quite tight.

## Source Code Deliverables

Our entire codebase, along with relevant comments and other related documentations such as the floor plans used, the research papers being referenced to (and the modifications that we have made to the algorithms), as well as the Wi-Fi AP model specifications, can be found here: <https://github.com/jamestiotio/FindMyTag>