1. Subscriber Mobility Status Detection

This is a data science problem which aims at developing a feature to derive mobility patterns of each cell user (i.e. the time periods where the user is stationary and moving). GPEH data was used for this task. Running the program will return you a list of start & end times where an IMSI remained stationary in a single location. It also returns the routes found between such locations, which is used for the work elaborated upon in Section 4.

3.1 Visualizing the Data

After extracting 5 days of GPEH data for 8 IMSIs, I started off by using Tableau to visualize each user's locations throughout one day. An example of a resulting visualization is shown in Figure 2.

From these visualizations, we can deduce that the areas with a high density of GPEH points on the map signify a cell user remaining stationary at that location, while middle points between these high-density areas represent the path taken by the cell user between locations.

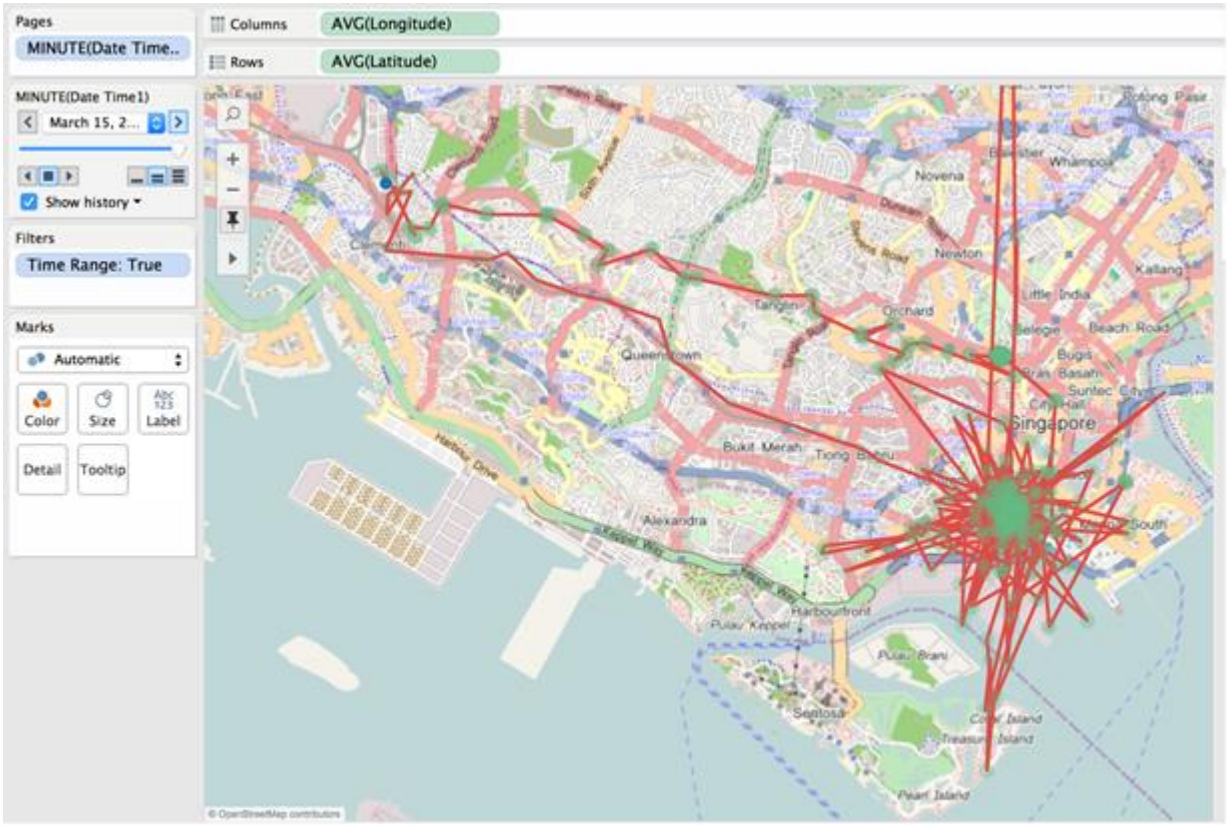

Figure 2: Visualization of the Routes and Locations of a Subscriber in One Day

Hence, to distinguish between stationary and mobile time periods, we will need to separate points in the high concentration areas with those that are in between the high concentration areas.

3.2 DBSCAN Clustering

Using Python, I performed DBSCAN clustering on the data to identify areas of high density. Unlike k-means clustering, DBSCAN does not require a specified number of clusters to be formed. Rather, its clustering algorithm determines the optimal number of clusters for each dataset based on two parameters specified – eps and min_samples. Hence, DBSCAN is particularly useful in the context of this problem, as different subscribers have different numbers of stationary locations over various days.

I experimented with a range of parameter values on the multiple IMSI datasets, each time plotting the clustering algorithm's output in Tableau to determine whether its cluster points had been correctly identified. I applied DBSCAN clustering to both the raw GPEH datasets (having exact datetime field, eg. 2016-03-13 23:08:31) and to GPEH data with averaged lat-longs by the minute (eg. datetime field: 2016-03-13 23:08; Latitude and Longitude are the average of the Lat-longs which fall within that minute).

DBSCAN clustering did not work effectively on the raw GPEH datasets, hence I focused on clustering the GPEH data-by-minute datasets instead. Eventually, I decided that the parameter values of eps=0.2 and

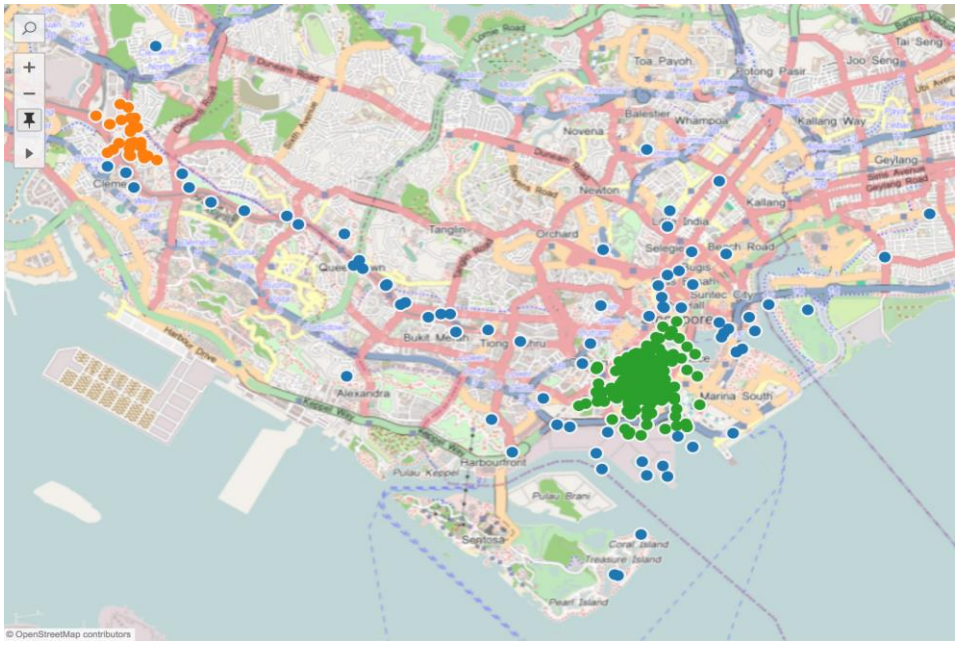

Figure 3: Visualization after DBSCAN clustering is applied

min_samples=10 clustered the GPEH points most effectively. The figure below shows some appropriately clustered data of an IMSI under these parameter values.

3.3 Overclustering of Single Locations

However, applying the same parameter values to all data posed a problem of overclustering in certain datasets – where a single location is detected as multiple clusters. An example is shown in the visualization below, where the location on the left is split into 4 clusters.

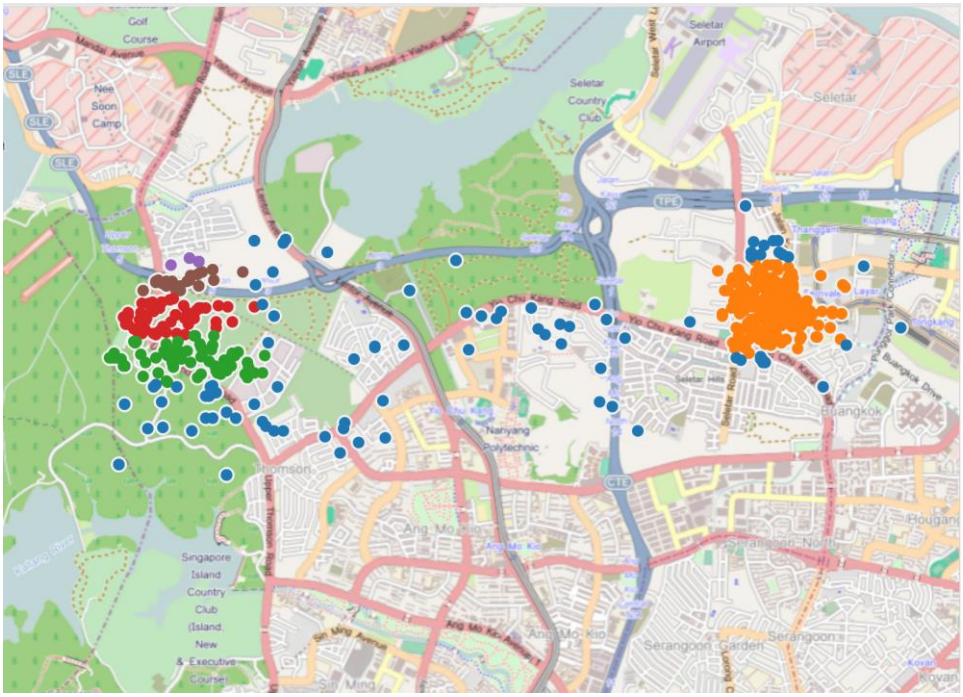

Figure 4: Over-clustering of a Single Location

To resolve this problem, I first attempted to merge clusters with centroids that were very near each other. However after inspecting all the visualizations, I soon realized that there were instances where centroids of separately visited locations were very close together as well – hence they would be incorrectly merged if I applied this strategy.

A zoom-in into the overclustered clusters reveals that many of their boundary points are very close to each other. An example is in Figure 5 below, which zooms in on the points between the red and green clusters from Figure 4. Hence, I decided on merging clusters which were found to have too many points close together. The final program merges two clusters together if >= 3 pairs of points between the clusters have haversine distances of <=1000m.

Figure 5: Boundary Points Between Two Nearby Clusters

3.4 Extracting of Routes Between Clusters

After processing of the data in Section 3.2 and 3.3, its output is in the following format, sorted by time:

155	##########	1.27969324	103.81942	2	core
156	##########	1.28974727	103.82091	2	non-core
157	##########	1.28213015	103.821491	2	core
158	##########	1.28207876	103.820854	2	core
159	##########	1.28217596	103.823887	2	non-core
160	##########	1.28429884	103.827372	-1	non-core
161	##########	1.28422533	103.831869	-1	non-core
162	##########	1.28784789	103.839189	-1	non-core
163	##########	1.29732213	103.845386	-1	non-core
164	##########	1.30775837	103.840197	-1	non-core
165	##########	1.30856752	103.841962	-1	non-core
166	##########	1.32262395	103.850872	-1	non-core
167	##########	1.3317437	103.860924	-1	non-core
168	##########	1.34545107	103.863671	-1	non-core
169	##########	1.36952418	103.863572	-1	non-core
170	##########	1.3911585	103.85662	0	core
171	##########	1.39174486	103.854898	0	core
172	##########	1.39237399	103.852534	0	core
173	##########	1.39057811	103.8529	0	core
174	##########	1.39809746	103.864681	-1	non-core
175	##########	1.39068925	103.851562	0	core

Figure 6: Clustered Data by Rows

Next, we want to extract the routes between them. Points that do not belong in any cluster are considered outliers and labelled as “-1” in the clustered data. “Non-core” is used to label points that do not fall within the core of a cluster – hence they are either points at the boundary of a cluster, or outliers. Consecutive

points labelled as “Non-core” are indicative of a travel route, such as the rows highlighted in yellow in Figure 6. This is what we intend to extract.

3.4.1 Parameters to Adjust Level of Noise in Extracted Routes

After extracting consecutive non-core points like that of the highlighted rows in Figure 6, I found that there were still a number of extracted points that were unlikely to be real routes, on inspection of their paths in Tableau. Hence, I included additional criteria to weed out these routes. Some of these criteria can be adjusted by the user, by specifying parameters to control the level of noise in the routes-output.

A few of the parameters are as listed below:

(a) **eliminate_long_intervals** (Default = True)

- If True, eliminates routes that contain long intervals between points, if the route starts and ends in the same cluster.
- For eg., the route below will be eliminated because its points are all half an hour apart, and it starts and ends in Cluster 0:

[['2016-03-14 05:33', '1.390039812419913', '103.85543312877417', '0', 'core'], ['2016-03-14 06:03', '1.3945057127016869', '103.86214535683393', '-1', 'non-core'], ['2016-03-14 06:33', '1.4006756396430178', '103.84902197867632', '0', 'non-core'], ['2016-03-14 07:03', '1.3938822837200153', '103.86101480573416', '-1', 'non-core'], ['2016-03-14 07:33', '1.3910051233143867', '103.85876309126616', '0', 'non-core'], ['2016-03-14 08:03', '1.3945848144316464', '103.8428133353591', '-1', 'non-core'], ['2016-03-14 08:33', '1.3882781190203766', '103.85421272367239', '0', 'core']]

(b) **max_negative_dot_product** (Default = 0.7)

- Ranges from 0 to 1
- A higher max_negative_dot_product will allow for more jagged routes to be included in the final route list.

Explanation of the Dot Product

Given two vectors A and B as shown in Figure 7, |A| and |B| represent the lengths of vectors A and B respectively, and Θ is the angle between the two vectors. The dot product of vectors A and B will have the following relationship to these values:

$A \cdot B = |A| * |B| * \cos(\Theta)$

- If **$\Theta < 90°$**, $\cos(\Theta)$ will be positive.
  As the vector lengths are always positive values, the **dot product will be positive**.

- If **$\Theta > 90°$**, $\cos(\Theta)$ will be negative.
  As the vector lengths are always positive values, the **dot product will be negative**.

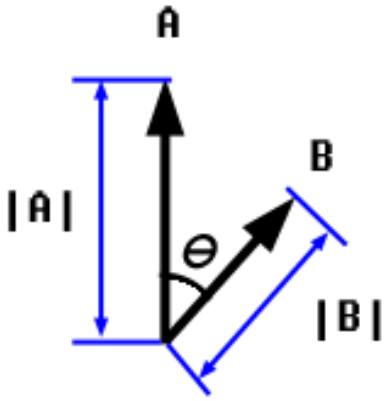

Figure 7: Vector Example

Application to Route Extraction

To derive the vector between two points on a map, we first use the following Mercator projection derivations to convert the lat-long coordinates of each point to x-y coordinates. For R, the radius of the globe was used.

$$x = \frac{\pi R(\lambda^\circ - \lambda_0^\circ)}{180}, \qquad y = R \ln\left[\tan\left(45 + \frac{\varphi^\circ}{2}\right)\right]$$

Every three consecutive points in a route are connected by two vectors, an example are the paths shown in Figure 8 or Figure 9.

For Figure 8, the angle between the two pink vectors is < 90°. Hence, $\Theta > 90°$ and the dot product between the two vectors is negative.

For Figure 9, the angle between the two pink vectors is > 90°. Hence, $\Theta < 90°$ and the dot product between the two vectors is positive.

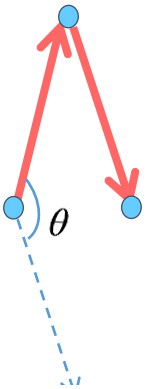

Figure 8: Example of Route Points that Result in Negative Dot Product

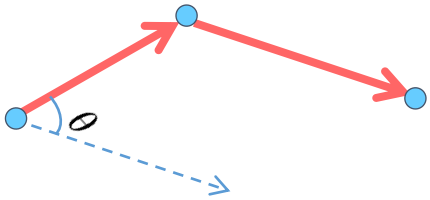

Figure 9: Example of Route Points that Result in Positive Dot Product

Paths that have a significant number of vector pairs with negative dot products are more likely to be noise rather than actual routes. For example, Figure 10 below shows the path of a detected route which has several negative dot products between its vector pairs. We can deduce from the visualization that the path is unlikely to be a real route as it does not seem to be leading towards anywhere.

Hence, a high percentage of negative dot products in a route increases the likelihood of the route being noise.

The parameter *max_negative_dot_product* is the maximum ratio of negative dot products in a path for it to be considered a legitimate route. For example, the path in Figure 10 has a negative dot product ratio of 0.78 (78% of its vector pairs have angles of <90° between each other). If *max_negative_dot_product*= 0.7, the path will be excluded from the final set of detected routes.

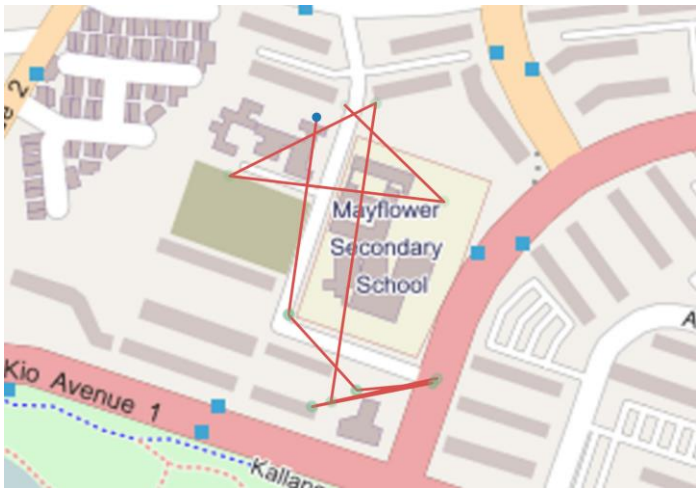

Figure 10: Path with Negative Dot Product Ratio of 0.78

(c) **max_speed** (Default = 150) and **above_max_speed_max_num** (Default = 1)

- Speed is calculated between pairs of points in a route.
- Here we exclude routes where there are more than 'above_max_speed_max_num' pairs of points with speeds that exceed 'max_speed'
- For eg, max_speed=150 and above_max_speed_max_num=1 means that we will exclude a route that has more than 1 pair of points with speeds of >150km/hr between them

2. Deciphering Transport Mode From Route Data

After identifying the time periods of travel for a particular IMSI, as well as extracting the points of each travel route, the next step is to use the route data to derive insights into the transport mode taken. The transport modes were classified into three possible categories – MRT/LRT, Bus, and Driving.

First, we want to see if the points of a detected route consistently fall on a specific public transport route in Singapore. If so, it is likely that the IMSI travelled by that particular mode of transport. To do this, we used OpenStreetMap data as a source to provide us with information on the public transport routes in Singapore. This source is also useful in providing us with any relevant expressways, shuttle bus routes or road names that the route points fall on.

4.1 Parsing of OpenStreetMap data

The OpenStreetMap project collects detailed and regularly updated geodata, and makes it available online for free. The raw OpenStreetMap (OSM) data is in XML format. Hence, I utilized the Python ElementTree XML API to parse the OSM data.

The OSM data is made up of nodes, ways and relations. A node is a point on the map, and is assigned lat-long coordinates. Ways are paths that consist of nodes. Relations too, are paths, but they are more extensive than ways. They consist of ways and/or nodes.

Each of these components includes tags that contain information of the transport infrastructure that fall on the relevant node/way/relation. In parsing of the OSM data, I saved the relevant tags that would provide me with enough information on public transport routes, expressways, road names and shuttle bus routes in Singapore.

4.1.1 Saving OSM Data Into Pickle Files

After parsing the OSM data, the lat-long coordinates of OSM nodes are stored into a KDTree data structure. KDTree is used for quick nearest-neighbour lookup. Querying the KDTree will rapidly return the indexes of the nearest neighbours of any point.

I further stored the tag descriptions into a list, to be retrieved accordingly from the indexes returned by the KDTree.

Both the KDTree of lat-long coordinates and descriptions list were stored in pickle files. This way, we do not have to parse the OSM file again for each query of the nearest neighbours of a route. We can simply lookup the KDTree and list for each query.

4.2 Querying of KDTree

There are two options to query a KDTree – query returns the k nearest neighbours of a given array of points, as specified by its parameter k; query_ball_point returns all points within distance r of a given array of points, as specified by its parameter r.

The program takes in an array containing the points of a route, and returns the tags associated with the resulting points from the query or query_ball_point methods, and the frequencies of these tags.

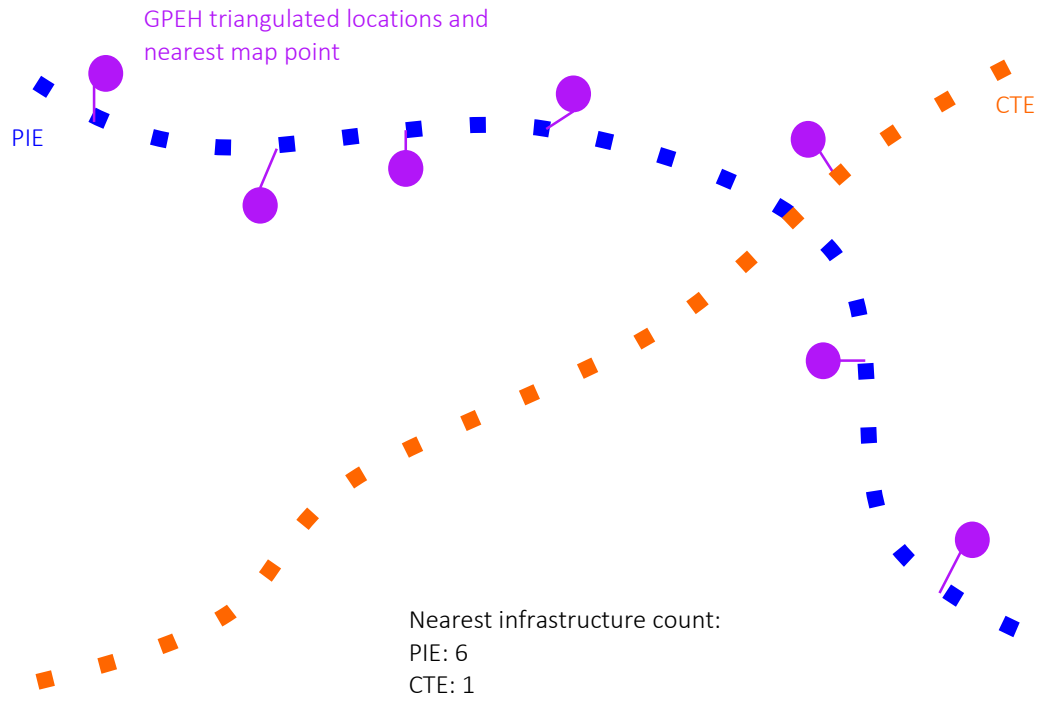

Figure 11: Illustration of Algorithm

For example, in Figure 11 above, an array of triangulated GPEH points of a subscriber is represented by the purple points in the visualization. Applying the *query* method with *k=1* will return the nearest neighbour of each GPEH point. Each nearest neighbour contains tag(s) which will then be collated and calculated for their frequencies.

4.3 Deriving Transport Mode by Time Range

Finally, based on the tags derived from the OSM data, we wish to identify the mode(s) of transport of a particular trip taken.

4.3.1 Sorting of Tags By Transport Mode

First, we need to identify the tags that correspond to each particular transport mode and traffic infrastructure. For example, to identify MRT as a mode of transport, we would look out for tags such as "ref = East-West Line". To identify Bus as a mode of transport, we look for tags that begin with "name = Svc". For expressways, we look out for tags such as "ref = CTE".

4.3.2 Identifying Transport Mode By Time Periods

Following this, we want to find the time periods during which a subscriber uses a particular mode of transport in a captured route.

My approach was to first fit in the most commonly occuring MRT and bus routes, till most of the points in the route had a predicted mode type attributed to it.

For example, for a particular route, we can obtain the following tag-frequency output:

Train stations:
[('ref = EW21;CC22', 2), ('ref = EW19', 1), ('ref = EW23', 1), ('ref = EW22', 1), ('ref = NE3;EW16;TE17', 1), ('ref = EW18', 1)]

Train lines:
[('ref = East-West Line', 16), ('ref = Circle Line', 2), ('ref = North-East Line', 1)]

Bus Services:
[('name = Svc 145', 11), ('name = Svc 147', 9), ('name = Svc 105', 8), ('name = Svc 970', 8), ('name = Svc 111', 7), ('name = Svc 32', 7), ('name = Svc 185', 7), ('name = Svc 166', 6), ('name = Svc 106', 6), ('name = Svc 33', 6), ('name = Svc 196', 6), …]

Expressways:
[('name = Central Expressway', 1), ('ref = CTE', 1)]

Looking at the tags under "Train Lines" and "Bus Services", we see that "ref = East-West Line" appears the most often. Hence, I decided to fit it in first as a predicted transport mode and derive the respective start

and end times of that East-West Line trip. I stored these in a dictionary, along with other relevant information such as the starting and ending MRT stations for that trip:

[{'starting_platform': {'ref = EW23'}, 'transport_mode': 'rail', 'end_time': '2016-03-14 07:31', 'line_or_service_no': 'ref = East-West Line', 'ending_platform': set(), 'middle_platforms': {'ref = EW19', 'ref = EW18', 'ref = NE3;EW16;TE17', 'ref = EW22', 'ref = EW21;CC22'}, 'fraction_of_route': '16/18', 'start_time': '2016-03-14 07:14'}]

The start and end times are taken from the first and last timestamps that contain the East-West Line tag:

[[['2016-03-14 07:06', '1.3233600223081028', '103.76461889594793', '0'], ['name = Svc 185', 'name = Svc 189', 'name = Svc 653']]
[['2016-03-14 07:08', '1.3180506592921482', '103.76874078065157', '0'], ['name = Svc 151', 'name = Svc 154', …, 'name = Svc 7', 'name = Svc 74', 'name = Svc CT28']]
[['**2016-03-14 07:14**', '1.3133589248117155', '103.76519858837128', '-1'], ['name = Svc 105', 'name = Svc 106', …, 'name = Svc 99', 'ref = EW23', '**ref = East-West Line**']]
…...
….
..
….
…...
[['**2016-03-14 07:31**', '1.2786948784713894', '103.84327869862318', '1'], ['name = Svc 145', 'name = Svc 166', 'name = Svc 197', …, 'name = Svc 80', '**ref = East-West Line**']]]

4.4 Issues with Current Algorithm & Future Work

However, while the above route has been correctly classified as an MRT trip, there were several other issues that cropped up while testing the program on other routes. Due to time constraints, I was not able to resolve some of them as listed below. With more time, we could tweak the algorithm to resolve these problems.

4.4.1 Dealing with Transport Modes of Same Frequency

One difficulty that I faced was in dealing with modes of the same frequency. For example, a particular route returns the following frequency-tuples from *find_route_type.py:*

Train Lines:
[('ref = North-East Line', 3), ('ref = Circle Line', 2), ('ref = Sentosa Express', 2), ('ref = East-West Line', 1)]

Bus Services:
[('name = Svc 166', 7), ('name = Svc 665', 6), ('name = Svc 131', 6), ('name = Svc CT8', 6), ('name = Svc 850E', 6), ('name = Svc 61', 6), ('name = Svc 761', 6), ('name = Svc 951E', 6), ('name = Svc 656', 6), ('name = Svc 143', 6), …, ('name = Svc 13', 1)]

The table above shows that the are quite a few bus services having frequencies of 6 in the route. After inputing Bus 166 (highest frequency of 7) into the route, the program randomly picks one of the bus numbers with frequency of 6 to fill the remaining route.

Hence, multiple runs give different bus numbers for the first portion of the route:

Output from Run #1: [{**'transport_mode': 'bus', 'line_or_service_no': 'name = Svc 665', 'expressways': {'ref = CTE', 'name = Central Expressway'}, 'end_time': '2016-03-14 12:07', 'start_time': '2016-03-14 12:01', 'fraction_of_route': '6/18'**}, {'transport_mode': 'bus', 'line_or_service_no': 'name = Svc 166', 'expressways': {'ref = CTE', 'name = Ayer Rajah Expressway', 'ref = AYE', 'name = Central Expressway'}, 'end_time': '2016-03-14 12:21', 'start_time': '2016-03-14 12:08', 'fraction_of_route': '11/18'}]
Output from Run #2: [{**'line_or_service_no': 'name = Svc 951E', 'fraction_of_route': '6/18', 'transport_mode': 'bus', 'end_time': '2016-03-14 12:07', 'start_time': '2016-03-14 12:01', 'expressways': {'ref = CTE', 'name = Central Expressway'**}}, {'line_or_service_no': 'name = Svc 166', 'fraction_of_route': '11/18', 'transport_mode': 'bus', 'end_time': '2016-03-14 12:21', 'start_time': '2016-03-14 12:08', 'expressways': {'ref = CTE', 'name = Ayer Rajah Expressway', 'name = Central Expressway', 'ref = AYE'}}]

To solve this problem, I initially tried to combine all modes of transport with the same frequencies together – placing them in a common frequency tuple like ('name = Svc665; name = Svc131; name = Svc951E; …', 6).

However, this had its problems as well, as not all the modes with same frequency fall within a common timeframe.

For example, a hypothetical sorted list of frequency tuples [('ref = Circle Line', 7), ('name = Svc 123', 7), ('name = Svc 98', 5), ('name = Svc 45', 5),…] could return two different modes of transport, if its modes with same frequencies do not fall in the same timeframes:

Possible Route 1

8-8.20am: Circle Line

8.21-8.30am: Svc 98

Possible Route 2

8-8.10am: Svc 45

8.11-8.30am: Svc 123

Here, “Circle Line” and “Svc 123” do not fall in the same timeframe although their frequencies are both 7. This is likewise for “Svc 98” and “Svc 45”. Hence, it would not make sense to group them in a common tuple like (‘ref = Circle Line; name = Svc 123’, 7) to fit into a route.

Also, based on the frequency tuples alone, Route 1 and Route 2 seem equally likely. Hence for cases such as these, there are multiple permutations of transport mode types, and the difficulty is in concluding a single most likely transport mode combination.

4.4.2 Routes of Transport Modes That Overlap

Another bug in the program is that some of the routes of the predicted transport modes may overlap. For example, a transport mode may be predicted to happen during route interval B and C. The program then attempts to fit another transport mode in the route from A to D. As A and D both fall outside of the B to C interval, the program detects that there are no overlap in the routes, and returns both trips in the final result.

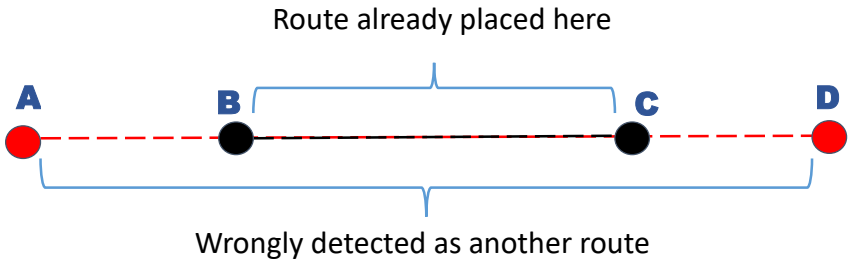

Figure 11: Illustration of Overlapping Routes

For this case, the program could be fixed by tracking the empty spaces not yet filled by routes, and ensuring that new trips only fall within these empty spaces (ie. A - B and C - D).

After which, we still need to consider whether the trip mode is still the most likely to happen amongst all the other modes. As it is split into two separate segments, there are now fewer points in each segment, hence both have a smaller probability of being an actual mode of transport taken by the IMSI.

4.4.3 Appropriate Time and Length for Trips

Additionally, we should consider what should be a reasonable time and length of a trip, for it to be deemed as valid. For example, in the result below, the bus trip on Svc 663 lasts for only 3min, and only takes up a small fraction (4/24) of the route.

[{'start_time': **'2016-03-18 08:18'**, 'end_time': **'2016-03-18 08:21'**, 'fraction_of_route': **'4/24'**, 'transport_mode': 'bus', 'line_or_service_no': 'name = Svc 663', 'expressways': {'ref = CTE', 'name = Central Expressway'}}, {'start_time': '2016-03-18 08:22', 'end_time': '2016-03-18 08:32', 'fraction_of_route': '6/24', 'starting_platform': set(), 'ending_platform': {'ref = CC16'}, 'transport_mode': 'rail', 'line_or_service_no': 'ref = Circle Line', 'middle_platforms': set()}, {'start_time': '2016-03-18 08:33', 'end_time': '2016-03-18 08:47', 'fraction_of_route': '13/24', 'transport_mode': 'bus', 'line_or_service_no': 'name = Svc 855', 'expressways': set()}]

In such cases, we may want to consider the speed and distance travelled during that short time period. If the speed is low and the distance is short, the subscriber may have walked the distance instead of taking a bus/MRT/car.

4.4.4 Waiting/Walking Time Between Transport Modes

[{'start_time': '2016-03-18 08:18', **'end_time': '2016-03-18 08:21'**, 'fraction_of_route': '4/24', 'transport_mode': 'bus', 'line_or_service_no': 'name = Svc 663', 'expressways': {'ref = CTE', 'name = Central Expressway'}}, {**'start_time': '2016-03-18 08:22'**, **'end_time': '2016-03-18 08:32'**, 'fraction_of_route': '6/24', 'starting_platform': set(), 'ending_platform': {'ref = CC16'}, 'transport_mode': 'rail', 'line_or_service_no': 'ref = Circle Line', 'middle_platforms': set()}, {**'start_time': '2016-03-18 08:33'**, 'end_time': '2016-03-18 08:47', 'fraction_of_route': '13/24', 'transport_mode': 'bus', 'line_or_service_no': 'name = Svc 855', 'expressways': set()}]

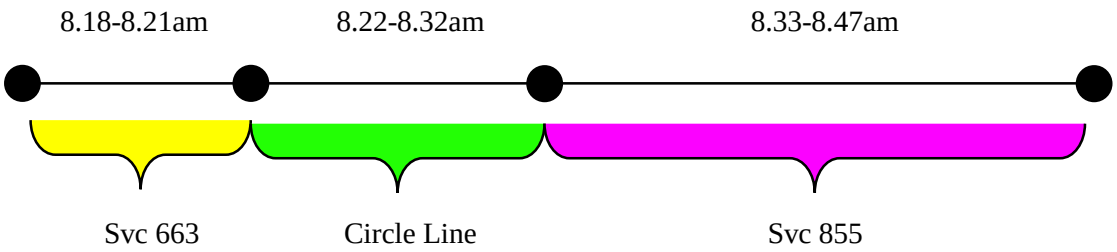

Figure 12: No Waiting or Walking Time between Transport Modes

Another thing to consider is the waiting time between public transport modes. In the output above, the transition between each transport mode only occurs one minute apart. This may not be very reasonable as there are usually waiting and/or walking time between bus and MRT/LRT trips.