Project Report

Trajectory Analysis and Contact Detection

Spring 2019

Project Mentor: Prof. **Petko Bogdanov**

By,

Santosh Kumar Goli **001353402**

Department of Computer Science

The University at Albany

State University of New York

Overview

“Trajectory analysis and Contact detection “ aims to efficiently pre-process and build a pipeline from Microsoft Geolife dataset in order to identify potential contacts between various users. This dataset consists of GPS trajectories of more than 150 users spread across several days. The contacts would be identified across various time and distance thresholds. Later, the variation in the behavior of the network with changes in time and distance thresholds is analyzed. The locations with large number of contacts are visualized on a map and thereby potential hotspots are identified. The various parameters of a contact are stored in a flat file for future processing.

Approach

An initial approach was to compute the pairwise distance and time computation between all points. The pair of points which obey the distance and time thresholds would be our points of contact. This would lead to a quadratic time complexity which is very expensive as the dataset is huge.

Next approach was to initially cluster the points based on time and more specifically hour of the day. Once clusters are identified distance computation is carried out pairwise within each cluster. This would reduce the time complexity as we employ “Divide and Conquer “strategy. But still the complexity is significantly high

Final approach was to make use of a HashMap to map all points to specific buckets. This reduced the complexity significantly. The generation of a good hash key is important. In order to generate the key, the following formulae were used.

```
tileLatConv = int(ptLatConv / ds) * ds

tileLngConv = int(ptLngConv / ds) * ds

t=int(t.timestamp()/dt)*dt
```

Here ds and dt are distance and time thresholds. Each point in a trajectory has Latitude, Longitude and Timestamp along with other attributes. The key for hashing takes into account latitude,longitude and the time. It is generated in the following way.

```
key=str(hash_lat) + '_' + str(hash_lon) +'_'+str(t)
```

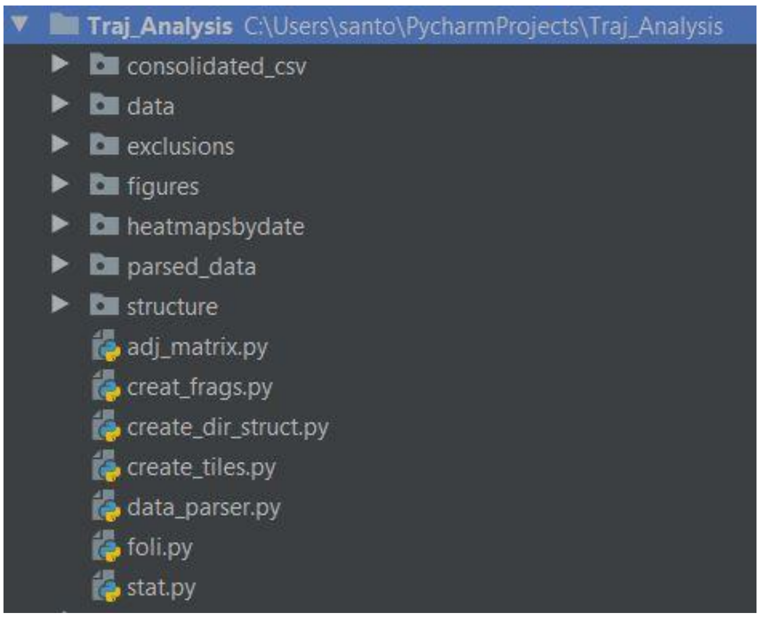
The points which map to the same key are put into same bucket. Once we have all the points mapped, the next step is to create fragments. A fragment is a subset of the points of a specific user which are contiguous within a bucket. A specific user can have multiple fragments within the same bucket. A representative fragment is selected among all the fragments pertaining to a single user by taking the mean of the latitudes, mean of longitudes, maximum of the timestamps and minimum of the timestamps.

Finally, we compute the distances among the representative fragments of several users within a bucket. We apply the same for all the other buckets and create a flat file per each setting of thresholds. The flat file would have several attributes of the contacts.

<u>Implementation</u>

This project is implemented using Python. It has few dependencies on libraries such as numpy,pandas and others.

Here is an overview of the directory structure:



```
▼ Traj_Analysis C:\Users\santo\PycharmProjects\Traj_Analysis
    ▶ consolidated_csv
    ▶ data
    ▶ exclusions
    ▶ figures
    ▶ heatmapsbydate
    ▶ parsed_data
    ▶ structure
      adj_matrix.py
      creat_frags.py
      create_dir_struct.py
      create_tiles.py
      data_parser.py
      foli.py
      stat.py
```

Create a project directory called Traj_Analysis or with any other name and navigate to it. Place all .py files in it.

Creating directory structure:

Code file: create_dir_struc.py

This file has code to generate the required directories within the project directory. It has a function “create_struct(ls)”. It takes in one parameter which is list of names of directories to be created. After running the file, place the folders ‘000 to 181’ from Microsoft Geolife dataset into ‘data’ folder created.

Parsing raw data:

Microsoft Geolife dataset has one folder each for 182 users. Each user has one file per specific date containing his GPS data for that date. We need to organize the data such that each user has only one single file associated with all his GPS data spread across several days.

Code file: data_parser.py

It has a function parse(n) which takes in one parameter. This parameter represents the number of users whose data needs to be parsed. In our case it is 182. So, this would generate 182 files with '.txt' extension. The file name representation is as follows.

'output_'+user+'.txt'.

Eg: output_009.txt

The files thus created are placed in the directory 'parsed_data'

Creating Tiles/Hashing:

After the data is parsed, we need to create tiles which cluster the users. In python we can achieve the functionality of a HashMap using dictionary. But in python a dictionary can't have duplicate keys, so we make use of class called Dictlist which essentially appends the values to a list and appends this list to the key. So, each key is going to be associated with a list of multiple values in the dictionary. Once the dictionary is created it is written to disk as a '.txt'. This file is useful to pull up the dictionary and its structure for further processing.

Code file: create_tiles.py

This file has a class Dictlist which handles the issue of duplicate key by creating a list and appending values to it. A dictionary of this class type is created.

Functions:

asRadians(degrees):

It takes in a one parameter and converts it into radians. This is essential for converting the latitude and longitude to their corresponding values in meters. It is a helper function.

getXYpos(relativeNullPoint, p):

It takes in two parameters:

> relativeNullPoint: It is a dictionary with two keys Latitude and longitude. Here latitude and longitude are representative of region under consideration. In our case it is the latitude and longitude of Beijing.
>
> p: It is also a dictionary with two keys latitude and longitude. Here latitude and longitude are of the current point being considered.

While creating the hashmap our key involves the following:

tileLatConv = int (ptLatConv / ds) * ds

tileLngConv = int (ptLngConv / ds) * ds

Here ptLatConv, ptLngConv are latitude and longitude of current point in degrees. ds is the distance threshold in meters. In order to implement these formulae, we must convert latitude and longitude into meters. So, this function handles this conversion efficiently. It is a helper function.

mapp(ds,dt):

This is the main function which implements the hashmap using dictionary. In this function all the users' files from 'parsed_data' are scanned. All the GPS points are put into the dictionary based on the key generated. The key is generated in the following way as already mentioned in the approach section:

```
tileLatConv = int (ptLatConv / ds) * ds

tileLngConv = int (ptLngConv / ds) * ds

t=int(t.timestamp()/dt)*dt
```

Here ds and dt are distance and time thresholds respectively. Once the dictionary is created, it is written to the disk using json.dumps() and the tile structure is retained for future use.

Creating fragments:

The dictionary written to disk in previous is loaded.

We also consider the point id(pid) in the list of attributes of a point while creating the hashmap. This is essential in determining the fragments of a single user within a tile. The point_id is the line number. If two points say p1,p2 of the same user within a tile differ in point_id by more than 1 then that would essentially indicate a break point for the determination of fragments. The set of all the points until p1(including p1) will be considered as one fragments and the set of points from p2 till wherever the difference between point_ids is 1, can be considered as the second fragment. Once all the fragments of a single user within a tile are identified a representative fragment (which is a single point) is created as discussed in the approach section. This process is applied to all users.

Code file: create_frags.py

It has a class DictList which handles the dictionary part.

Functions:

checkForOutliers(key, p1,p2):

It takes in three parameters Points p1 and p2 are the points for which the distance is to be computed and the key(tile) associated with these points. This function returns the distance between the points by using the Haversine formula if the distance is within the distance threshold. It also writes to disk a file containing outliers just in case if the two points exceed our distance threshold. The file if created will be placed in the 'exclusions' folder. The name of this file is representative of the key associated with the points. This is to emphasize the fact that few points can slightly fall off the threshold. This is a helper function.

create_fragments(load_dict,ds,dt)

It takes in three parameters. The loaded dictionary from the previous step(hashing). The parameters ds and dt are the distance and time thresholds respectively. A dictionary called fragment_dict is created. The keys in this dictionary are in the following form:

fragment_dict[key + "&" + curr_usr + "&" + str(frag_count)] = [values[pt_i]]

Here key is the key from the loaded dictionary. Curr_user is the current user for which fragments need to created and frag_count is the id of the fragment for that particular user within the tile(represented by the key). The values associated with the key are appended to the dictionary .

Once frag_dict is filled up , we have a dictionary with fragments of all users . Now we have create a new dictionary call final_frag in which each key would be associated with a representative fragment per user within that tile. This is done in two steps. Each key in frag_dict is uniquely identified by curr_user and frag_count. The values in such keys are iterated and a representative fragment is created based on the mean of latitudes, mean of longitudes, and maximum and minimum of all timestamps within that fragment(for that specific user) and also minimum of all the timstamps(for that specific user) . The representative fragments thus created are appended to a key in final_frag. The key would essentially be the same as the key used in "Creating Tiles" section.

After we have the final_frag dictionary we proceed to identify the distances between the representative fragments of several users pairwise within that fragment. Also, the interaction time is computed based on intersection or union of the timestamps(lines 109-116). The distance computation part is handled by the checkForOutliers() function. After the function returns the distance, the next step would be to create the '.CSV' file. The file has the following columns or attributes.

UserA,LatA,LngA,Min_time_UserA,Max_time_UserA,UserB,LatB,LngB,Min_time_UserB,Max_time_UserB,Distance_apart(m),Interaction_time(ms),Tile_key.

Similarly, the '.CSV' files are generated for the other settings of thresholds.

Note: Creating tiles followed by creating fragments can be done in one go by using code file ‘frag_ext.py’. This combines both the implementations but skips the storing of dictionary to disk.

Generating statistics:

Code file: stat.py

The main aim of this file is to generate the largest connected component and average_degree for the entire set of users per setting. Later, two graphs are plotted. 1) Largest_connected _component vs settings 2) Average_degree vs settings.

Functions:

create_graph(filename):

It takes in the one parameter which is the filename of the “.CSV’ to be considered.

It creates a graph from the .csv based on the values from “UserA” and “UserB” which hold the user ids for that specific contact.

This is a helper function.

get_largest_connected_component(G_list)

It takes in a single parameter called G_list which is a list of graphs (one for each setting) created by create_graph. This function calculates the largest connected component from the graph and prints it to console. It appends this value to a list (lconn). This is done for other graphs in the G_list as well. A figure representing the users in the largest connected component is displayed one each per setting.

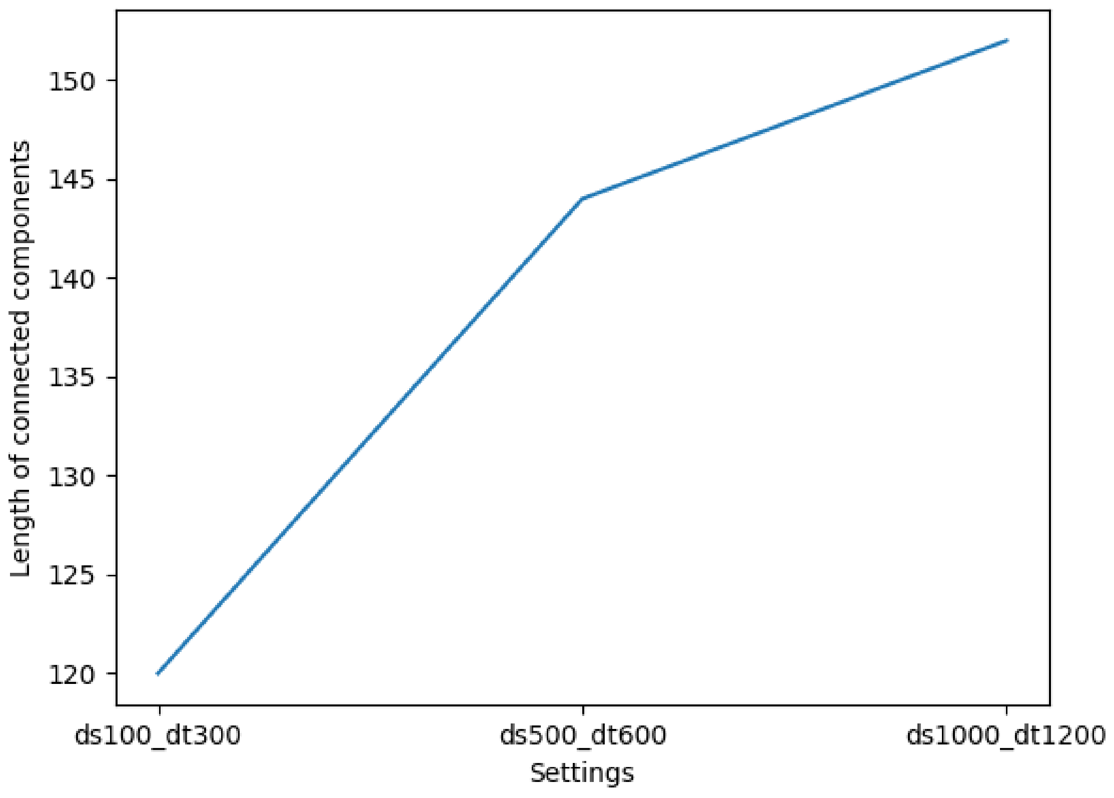
After lconn is populated we plot the largest_components against settings of thresholds and display it. The figure thus generated is saved in the folder “figures”

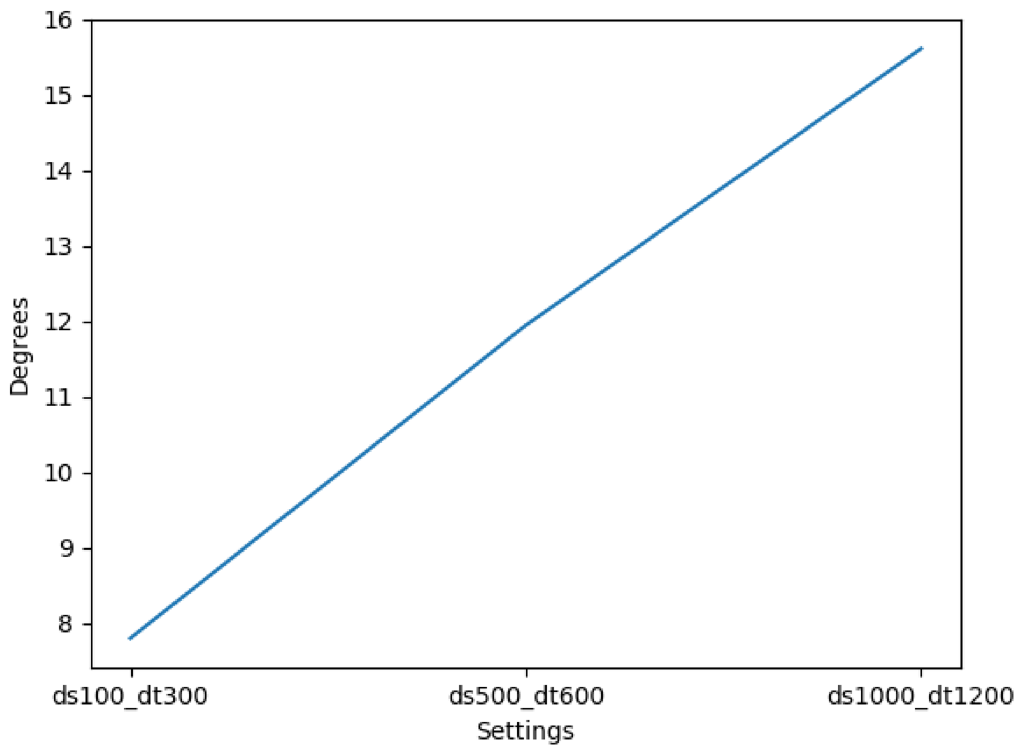
get_average_degree(G_list):

It takes in a single parameter called G_list which is a list of graphs (one for each setting) created by create_graph . This function calculates the largest connected component from the graph and prints it to console. It appends this value to a list (avg_deg). This is done for other graphs in the G_list as well.

After avg_deg is populated we plot the average_degree against settings of thresholds and display it. The figure thus generated is saved in the folder “figures”

The graphs generated by running this code file can be seen below:





The following console output gives a better understanding of the graphs shown above.:

```
Largest Connected component for ds100_dt300

120

Largest Connected component for ds500_dt600

144

Largest Connected component for ds1000_dt1200

152

Average degree for ds100_dt300

7.8

Average degree for ds500_dt600

11.947712418300654

Average degree for ds1000_dt1200

15.615384615384615
```

Creating adjacency matrix:

Code file: adj_matrix.py

The file creates an adjacent matrix per date in which values are the distances between the points in contact. All such adjacency matrices are stored in dictionary in which the key is the date.

Functions:

create_adj_matrix(n)

This function takes in one parameter n which is the size of the adjacency matrix to be created.

It is 182 x 182 in our case as there are 182 users. This function also initialized all values with "False" in the matrix.

This is a helper function and returns the matrix created.

load_mat(filename, n)

This function takes in two parameters filename which is the name of the '.CSV' to be considered and n which is the number of users.

It calls the create_adj_matrix() and creates an adjacency matrix of size n x n. The values in the file are grouped by date and for each date an adjacency matrix is created. The value at cell [i][j] in the matrix is filled with distance value if there is contact between user with id i and user with id j. Even values at cell [j][i] are filled with same value. All such matrices generated are then stored in a dictionary with keys representing the dates.

update_dist_from_date_user_id(T, key, UserA, UserB, dist_value):

This is a helper function to update distance value in some cell pointed by 'UserA' and 'UserB. Here T is the dictionary of dates, key is the date where an update has to be made. This key points to the matrix where the update needs to be done. UserA and UserB are the cell indices and dist_value is the new distance which needs to be stored.

Creating hotspot(heatmap) visualizations:

code file: foli.py

This file plots the average of latitude and average of longitude in a contact point and visualizes it on map. A heatmap of all such points is created to show where the hotpots exist and how these hotspots vary with time.

The initial lines of the code load the .csv to be considered and create a few additional columns such as date_from_key . This values in date_from_key column as based on the substring extracted from the "Tile_key ".

Functions:

generateBaseMap(default_location=[39.913818, 116.363625], default_zoom_start=12):

Here default location is a list which has latitude and longitude value for the region being considered and default_zoom_start is the zoom level at which the map should be initially loaded. This function creates an initial map. This map is provided by MapBox API.There are other providers as well. Here default location points to coordinates of Beijing.

create_visual(curr_date):

This function takes in the curr_date parameter which represents the date and generates a heatmap based on density of average latitude and average longitude(count). The heatmap is dynamic and changes with time. The heatmap is written to disk as .html file and can be opened in any browser to visualize it. The heatmap is generated making use of Folium library.

Use: A list of dates to be analysed can be created and passed to this function. The results are stored in 'heatmapsbydate' folder. They can be distinguished based on their name which contains the date.

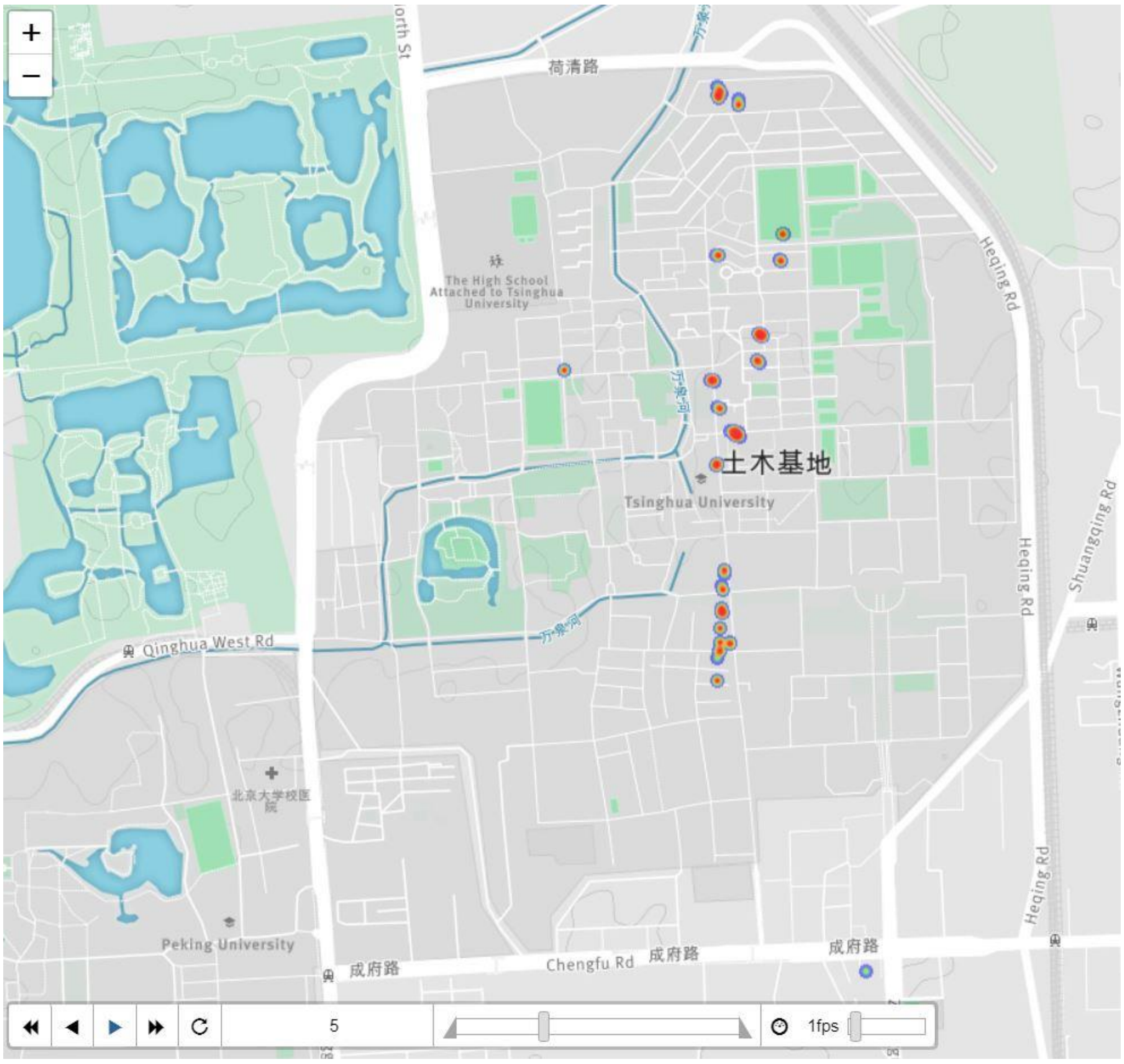
For future use: The heatmaps can also be generated based on the location or key(lat,lng) spread across all days.

A list of dates is specified at the bottom of tile file called 'curr_list'. The dates from this list can be passed one at a time to this function.

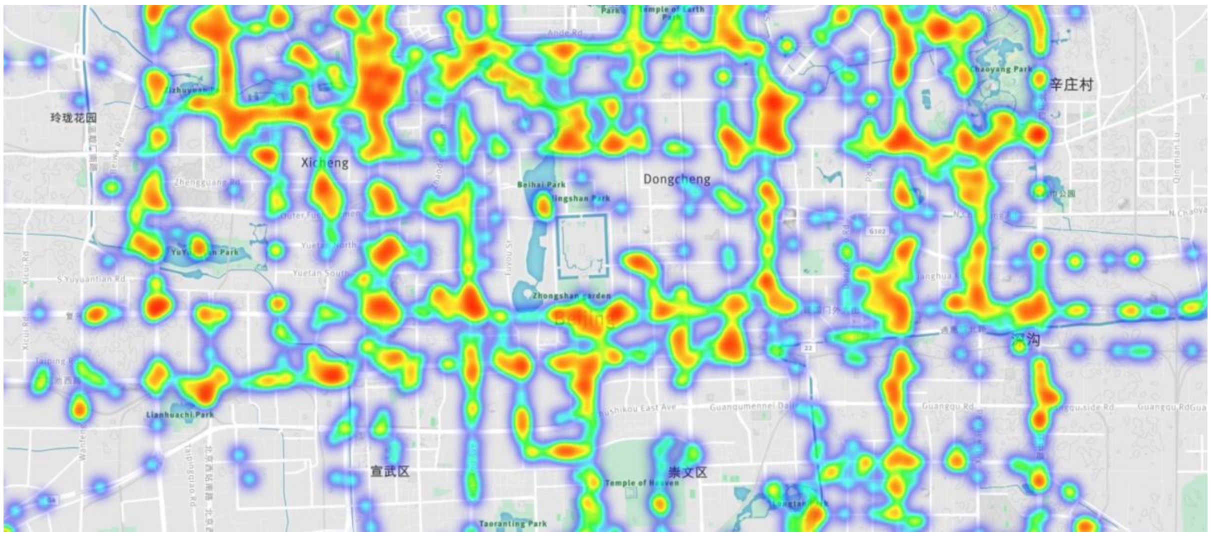
create_general_visual(filename):

This function takes in the filename of the .csv for which a static heatmap needs to be generated. The heatmap is written to disk as .html and can be opened in any browser to visualize it. The results are written to heatmapsbydate folder. They have the phrase ‘general’ in their name.

Some of the snapshots of such heatmaps are given below.



A snapshot showing heatmap of contacts of Tsinghua University on 11-04-2009 at around 5 AM.



A snapshot showing the heatmap of contacts in Beijing for the setting ds 1000 and dt 1200 spread across several days.

Dependencies:

Programming Language:

Python – 3.4 or higher.

Packages:

1)numpy

2)pandas

3)networkx

4)folium

5)matplotlib

Dataset: https://www.microsoft.com/en-us/download/confirmation.aspx?id=52367

Conclusion:

Thus, this project helps us to pre-process the data efficiently and identify the contacts. It can be used with other GPS trajectory datasets with a few changes in extraction of raw data. Also, the collection (Tensor) of adjacency matrices can be used in future for various purposes. The degree of a user can be used as a measure of dependability of that user. The average degree and largest connected component can be used to judge which setting of thresholds is best suited for our purpose.