A Project Report

On

SOCIAL NETWORK ANALYSIS

BY

Darshan Chandak

2020A7PS2085H

Under the supervision of

Dr. Apurba Das

SUBMITTED IN THE COMPLETE FULFILLMENT OF THE REQUIREMENTS OF

CS F377: DESIGN PROJECT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

HYDERABAD CAMPUS

(December 2023)

ACKNOWLEDGMENTS

I am indebted to Dr. Apurba Das, my project mentor, for his constant guidance and feedback on my work. A big thanks to Dr. Rajib Ranjan Maiti, whose guidance has gone a long way toward the successful execution of the project.

I sincerely thank the Department of Computer Science and Information Systems (CSIS), BITS Pilani Hyderabad Campus, for providing me with this excellent opportunity to gain valuable insights and learn new things via this Formal Design Project.

I am indebted to my parents, teachers, and friends, as without their prayers and best wishes, I wouldn't have reached this far.

Birla Institute of Technology and Science-Pilani,

Hyderabad Campus

CERTIFICATE

This is to certify that the project report entitled "**SOCIAL NETWORK ANALYSIS**" submitted by **Mr. Darshan Chandak** (ID No. **2020A7PS2085H**) in complete fulfillment of the requirements of the course **CS F377, Design Project Course**, embodies the work done by him under my supervision and guidance.

Date: 08/12/2023 **(Dr. Apurba Das)**

BITS- Pilani, Hyderabad Campus

ABSTRACT

Missing data in time series data is a pervasive problem that puts obstacles in the way of advanced data analysis. A popular solution is data imputation, where the fundamental challenge is determining what values should be filled in the place of the missing values.

In this project, a method of Data Imputation has been successfully performed for the Geolife Trajectory Dataset. To accomplish this, firstly, the data is flattened for each user based on time value. The weight values are generated based on the number of occurrences of a particular (latitude, longitude) pair in a specific time interval across all days for a particular user. Finally, using these weight values, a weighted random sampling is carried out on the pre-processed data to fill out the missing values for that user.

Then, clusters are formed for each user using the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) Algorithm. Clustering is carried out based on Latitude and Longitude values for the user across all days on raw and processed data. Also, the Goodness Score of the Clustering, i.e., the Silhouette score, is calculated for each user. After this, a similarity score is calculated for each pair of users using the Jaccard Similarity Metric.

CONTENTS

INTRODUCTION

1. Time Series Data

Time series data is a collection of observations acquired through repeated measurements over time. Plot the points on a graph; one of your axes would always be time. Time series metrics refer to data followed at a specific increment in time.
For example, a metric could refer to how much stock is sold in a store from one day to another.

Time series data is around since time is a component of everything observable. As our world gets increasingly instrumented, sensors and systems constantly emit a persistent stream of time series data. Such data has numerous applications across various industries.

Let us put this in context through some examples. Examples of time series analysis:

- Rainfall measurements
- Stock prices
- Annual retail sales
- Monthly subscribers
- Heartbeats per minute

2. Time Series Data Imputation

Time series imputation refers to the process of filling in missing values in a time series dataset with estimated or predicted values. Time series data typically involves observations taken at successive time points, and missing values can occur due to various reasons such as sensor errors, data collection issues, or simply gaps in the data.

Imputing missing values in time series data is essential for several reasons:

- **Analysis and Visualization**: Complete time series data is often necessary for proper analysis and visualization. Missing values can lead to biased results and misinterpretation of trends and patterns.
- **Modeling**: Many time series forecasting and analysis techniques require a complete dataset. Imputing missing values allows you to use these techniques effectively.
- **Data Integration**: Handling missing values ensures consistency and comparability when combining multiple time series datasets.

3. Geolife GPS Trajectory Dataset

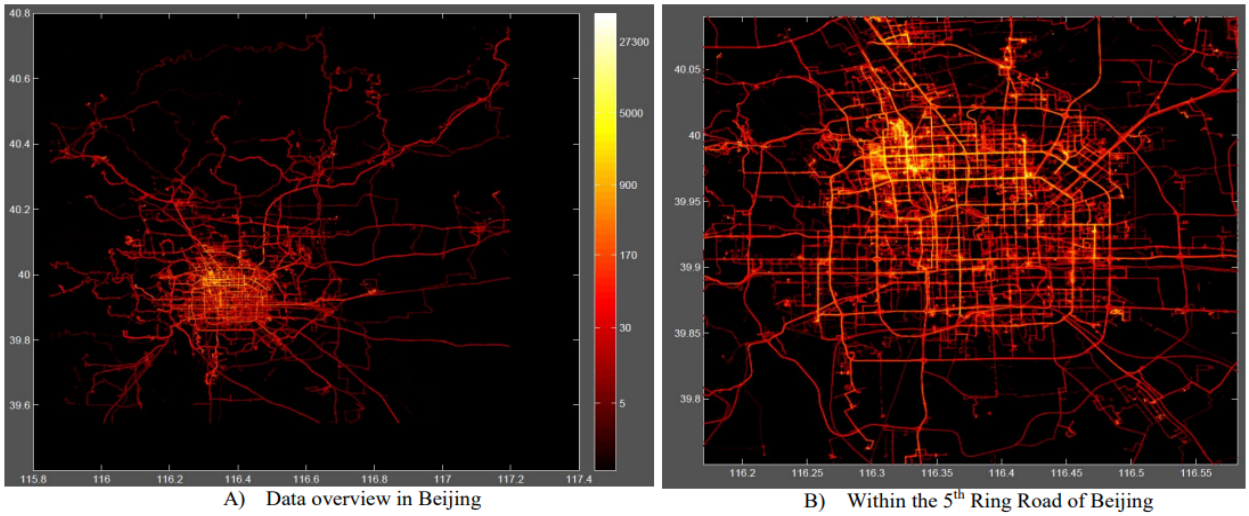
This GPS trajectory dataset was collected in the (Microsoft Research Asia) Geolife project by 182 users over five years (from April 2007 to August 2012).
A GPS trajectory of this dataset is represented by a series of time-stamped points, each containing the details of latitude, longitude, and altitude values. This dataset includes 17,621 trajectories with a total distance of 1,292,951 kilometers and a total duration of 50,176 hours.

These trajectories were registered at a variety of sampling rates. 91.5 percent of the trajectories are logged in a dense representation, e.g., every 1 to 5 seconds or every 5 to 10 meters per point. This dataset documented a wide range of users' outdoor activities, including life routines like going home and work and recreation and sports activities like shopping, sightseeing, dining, hiking, and cycling.

This trajectory dataset can be utilized in many research fields, such as mobility pattern mining, user activity recognition, location-based social networks, location privacy, and location recommendation.

Although this dataset is widely dispersed in over 30 cities in China and even in some cities in the USA and Europe, most of the data was generated in Beijing, China.

Figure 2 plots this dataset's distribution (heat map) in Beijing. The statistics on the right side of the heat bar denote the number of points generated in a location.



A) Data overview in Beijing

B) Within the 5th Ring Road of Beijing

Figure 1: Distribution of the dataset in Beijing city

Dataset Link: [Geolife GPS trajectory dataset - User Guide - Microsoft Research]

DATA FORMAT

Trajectory File

Every folder of this dataset stores a user's GPS log files, which were transformed into PLT format. Each PLT file contains a single trajectory and is named by its starting time. We use GMT in each point's date/time property to avoid time zone confusion.

PLT format:

Lines 1 to 6 are useless in this dataset and can be ignored. Points are defined in the following lines, one for each line.

Field 1: Latitude in decimal degrees.

Field 2: Longitude in decimal degrees.

Field 3: All set to 0 for this dataset.

Field 4: Altitude in feet.

Field 5: Date - number of days (with a fractional part) that have passed since 12/30/1899.

Field 6: Date as a string.

Field 7: Time as a string.

Note that Field 5 and Field 6 & 7 represent the same date/time in this dataset. We may use either of them.

Example:

39.906631,116.385564,0,492,40097.5864583333,2009-10-11,14:04:30
39.906554,116.385625,0,492,40097.5865162037,2009-10-11,14:04:35

In our code, we drop the initial six lines and Fields 3, 4, and 5 in the data pre-processing section.

DATA IMPUTATION METHODOLOGY

1. Data Pre-processing

Here, we read the entire dataset folder (Data Folder).

The folder structure of the dataset is as follows:

The folder has 182 subfolders corresponding to each user. Then, each subfolder has a sub-sub folder named "Trajectory." This Trajectory folder has all the .plt files corresponding to the user's GPS Trajectory Data. Each .plt file is named after the starting time of the first data point.

DATA PRE-PROCESSING

```
[1]
import numpy as np
import pandas as pd
import glob
import os
import os.path
import datetime

print("Imports Done!")
```

```
...
Imports Done!
```

```
[2]
def read_plt(plt_file):
    points = pd.read_csv(plt_file, skiprows=6, header=None, infer_datetime_format=True)
    points.rename(inplace=True, columns={5: 'Date', 6: 'Time', 0: 'Latitude', 1: 'Longitude'})
    points.drop(inplace=True, columns=[2, 3, 4])

    return points
```

We drop the columns - 2, 3, 4, i.e., field values corresponding to 3, 4, 5 (Altitude, Zero Value, Fractional DateTime Value).

```
[3]
def read_user(user_folder):
    plt_files = glob.glob(os.path.join(user_folder, 'Trajectory', '*.plt'))
    temp = pd.concat([read_plt(f) for f in plt_files])

    return temp
```

```
[4]
total_users = 0

def read_all_users(folder):
    subfolders = os.listdir(folder)
    total_users = len(subfolders)
    temps = []
    for i, sf in enumerate(subfolders):
        print(f'Processing User {i+1}/{total_users}')
        temp = read_user(os.path.join(folder,sf))
        temp['User'] = int(sf)
        temps.append(temp)

    return pd.concat(temps)
```

We then iterate through all users for all days to read all the present data.

```
[5]
df = read_all_users('Data')
```

```
Processing User 1/182
Processing User 2/182
Processing User 3/182
Processing User 4/182
Processing User 5/182
Processing User 6/182
Processing User 7/182
Processing User 8/182
Processing User 9/182
Processing User 10/182
```

```
[6]
df
```

	Latitude	Longitude	Date	Time	User
0	39.984702	116.318417	2008-10-23	02:53:04	0
1	39.984683	116.318450	2008-10-23	02:53:10	0
2	39.984686	116.318417	2008-10-23	02:53:15	0
3	39.984688	116.318385	2008-10-23	02:53:20	0
4	39.984655	116.318263	2008-10-23	02:53:25	0
...	...	...	...	...	...
17	40.914867	111.710500	2008-03-14	03:39:56	181
18	40.914267	111.710333	2008-03-14	03:41:17	181
19	40.912467	111.710667	2008-03-14	03:43:02	181
20	40.911517	111.711317	2008-03-14	03:43:28	181
21	40.910933	111.711617	2008-03-14	03:43:40	181

24876978 rows × 5 columns

We then process the data for all 182 users and create a data frame.

2. Data Flattening

ROUNDING OFF TIME VALUE TO THE NEAREST HOUR

```
df['Time'] = pd.to_timedelta(df['Time'])
df['Time'] = (df['Time'] + pd.to_timedelta('30T')).dt.floor('H')
df['Time'] = df['Time'].astype(str).str[-8:]

df
```

[8]

	Latitude	Longitude	Date	Time	User
0	39.984702	116.318417	2008-10-23	03:00:00	0
1	39.984683	116.318450	2008-10-23	03:00:00	0
2	39.984686	116.318417	2008-10-23	03:00:00	0
3	39.984688	116.318385	2008-10-23	03:00:00	0
4	39.984655	116.318263	2008-10-23	03:00:00	0
...	...	...	...	...	...
17	40.914867	111.710500	2008-03-14	04:00:00	181
18	40.914267	111.710333	2008-03-14	04:00:00	181
19	40.912467	111.710667	2008-03-14	04:00:00	181
20	40.911517	111.711317	2008-03-14	04:00:00	181
21	40.910933	111.711617	2008-03-14	04:00:00	181

24876978 rows × 5 columns

We first round off the time value to the nearest hour value to get Latitude and Longitude values for all users across all days in time intervals.

FLATTENING THE DATA BASED ON TIME VALUE

```
grouped = df.groupby(['User', 'Date', 'Time']).agg({'Latitude': 'mean', 'Longitude': 'mean'}).reset_index()
user_dataframes = []
unique_users = grouped['User'].unique()

for user in unique_users:
    user_df = grouped[grouped['User'] == user].copy()
    user_df.rename(columns={'Latitude': 'Average Latitude', 'Longitude': 'Average Longitude'}, inplace=True)
    user_dataframes.append(user_df)

user_dataframes
```

[9]

```
[    User        Date      Time  Average Latitude  Average Longitude
 0      0  2008-10-23  03:00:00         39.984303         116.306918
 1      0  2008-10-23  04:00:00         39.988607         116.310340
 2      0  2008-10-23  05:00:00         39.999054         116.323453
 3      0  2008-10-23  10:00:00         40.008035         116.320250
 4      0  2008-10-23  11:00:00         40.008757         116.321634
..    ...         ...       ...               ...                ...
759     0  2009-07-05  04:00:00         39.970864         116.327721
760     0  2009-07-05  05:00:00         39.999998         116.309603
761     0  2009-07-05  06:00:00         40.079651         116.236529
762     0  2009-07-05  07:00:00         40.058529         116.248749
763     0  2009-07-05  08:00:00         39.991186         116.315803

[764 rows x 5 columns],
```

Now, we average out the Latitude and Longitude values in the same time interval for each day for each user.

```
final_user_dataframes = []

for user_df in user_dataframes:
    user_df['Time'] = pd.to_datetime(user_df['Time']).dt.strftime('%H:%M:%S')
    min_time = user_df['Time'].min()
    max_time = user_df['Time'].max()
    user_dates = user_df['Date'].unique()
    all_hours = pd.date_range(start=min_time, end=max_time, freq='H').strftime('%H:%M:%S')
    date_time_combinations = [(date, hour) for date in user_dates for hour in all_hours]
    missing_rows = [
        {
            'User': user_df['User'].iloc[0],
            'Date': date,
            'Time': hour,
            'Average Latitude': 0,
            'Average Longitude': 0
        }
        for date, hour in date_time_combinations if (date, hour) not in zip(user_df['Date'], user_df['Time'])
    ]
    user_result_df = pd.concat([user_df, pd.DataFrame(missing_rows)], ignore_index=True)
    user_result_df['Date'] = pd.to_datetime(user_result_df['Date'])
    user_result_df.sort_values(by=['Date', 'Time'], inplace=True)
    user_result_df.reset_index(drop=True, inplace=True)
    final_user_dataframes.append(user_result_df)

final_user_dataframes
```

[10]

```
[      User       Date      Time  Average Latitude  Average Longitude
 0        0 2008-10-23  00:00:00          0.000000           0.000000
 1        0 2008-10-23  01:00:00          0.000000           0.000000
 2        0 2008-10-23  02:00:00          0.000000           0.000000
 3        0 2008-10-23  03:00:00         39.984303         116.306918
 4        0 2008-10-23  04:00:00         39.988607         116.310340
 ...    ...        ...       ...               ...                ...
 2995     0 2009-07-05  19:00:00          0.000000           0.000000
 2996     0 2009-07-05  20:00:00          0.000000           0.000000
 2997     0 2009-07-05  21:00:00          0.000000           0.000000
 2998     0 2009-07-05  22:00:00          0.000000           0.000000
 2999     0 2009-07-05  23:00:00          0.000000           0.000000

 [3000 rows x 5 columns],
```

For each user, we find the minimum and maximum time interval for which data is present in the original dataset for that user across all days. Now, we call this Tmin and Tmax.
For the sake of uniformity, we would like to have data for all the time intervals between Tmin and Tmax for all days for each user. Thus, we create a numpy array of data frames that have, for all users, for all days, rows corresponding to each time interval between Tmin and Tmax. If datapoint is already present in the original dataset, it is as it is. But if a datapoint is missing in the original dataset, we add a row with 0 Latitude and Longitude values. This way, we create our flattened data in a numpy array of data frames called **'final_user_dataframes.'**

3. Weighted Random Sampling

GENERATING THE WEIGHT VALUES FOR RANDOM SAMPLING

+ Code + Markdown

```
user_tuples_dataframes = []

for user_df in user_dataframes:
    user_tuples = []

    # Round latitude and longitude to 3 decimal places
    user_df['Average Latitude'] = user_df['Average Latitude'].round(3)
    user_df['Average Longitude'] = user_df['Average Longitude'].round(3)
    unique_time_lat_lon = user_df[['Time', 'Average Latitude', 'Average Longitude']].drop_duplicates()

    # Iterate through unique time, latitude, and longitude pairs
    for index, row in unique_time_lat_lon.iterrows():
        time, lat, lon = row['Time'], row['Average Latitude'], row['Average Longitude']
        lat = round(lat, 3)
        lon = round(lon, 3)
        weight = user_df[(user_df['Time'] == time) & (user_df['Average Latitude'] == lat) & (user_df['Average Longitude'] == lon)].shape[0]
        user = user_df['User'].iloc[0]
        user_tuples.append((user, time, lat, lon, weight))

    user_tuples_df = pd.DataFrame(user_tuples, columns=['User', 'Time', 'Average Latitude', 'Average Longitude', 'Weight'])
    user_tuples_df.sort_values(by=['Time', 'Weight'], ascending=[True, False], inplace=True)
    user_tuples_df.reset_index(drop=True, inplace=True)
    user_tuples_dataframes.append(user_tuples_df)

user_tuples_dataframes
```

[11]

```
[    User      Time  Average Latitude  Average Longitude  Weight
0       0  00:00:00            40.010            116.315       2
1       0  00:00:00            40.008            116.317       2
2       0  00:00:00            40.005            116.308       1
3       0  00:00:00            40.011            116.298       1
4       0  00:00:00            40.011            116.297       1
..    ...       ...               ...                ...     ...
650     0  23:00:00            40.005            116.322       1
651     0  23:00:00            39.987            116.325       1
652     0  23:00:00            40.005            116.324       1
653     0  23:00:00            40.007            116.321       1
654     0  23:00:00            39.918            116.323       1
```

We then generate the weight values based on the number of occurrences of (Latitude, Longitude) values in a particular time interval across all days for each user.
These weights would be used to sample the data for filling in the missing values in the 'final_user_dataframes.'

<u>Weighted Random Sampling</u>:

Weighted random sampling is a statistical and computational technique used to select items from a collection or population in a way that gives each item a different probability of being chosen. In traditional random sampling, each item has an equal chance of being selected. However, in weighted random sampling, items are assigned individual weights or probabilities, and these weights determine the likelihood of selection.

The process of weighted random sampling typically involves assigning weights to each item, which sum up to a total weight. The probability of selecting an item is proportional to its weight, with items having higher weights being more likely to be chosen.

Overall, weighted random sampling is a versatile tool that helps in making more informed and meaningful selections in situations where not all items are of equal value or importance. It allows for a flexible and tailored approach to sampling that can improve the accuracy and effectiveness of various processes and applications.

WEIGHTED RANDOM SAMPLING

```
import random

# Create a dictionary to store user-to-DataFrame mapping
user_tuples_dict = {}
for user_tuples_df in user_tuples_dataframes:
    user = user_tuples_df['User'].iloc[0]
    user_tuples_dict[user] = user_tuples_df

# Iterate through final_user_dataframes to fill missing values
for user_df in final_user_dataframes:
    user = user_df['User'].iloc[0]
    # Identify rows with missing latitude and longitude
    missing_rows = user_df[(user_df['Average Latitude'] == 0) & (user_df['Average Longitude'] == 0)]

    for index, row in missing_rows.iterrows():
        time = row['Time']
        user_tuples_df = user_tuples_dict[user]
        matching_rows = user_tuples_df[user_tuples_df['Time'] == time]

        if not matching_rows.empty:
            weights = matching_rows['Weight']

            if weights.sum() > 0:
                # Perform weighted random sampling
                selected_row = random.choices(matching_rows.index, weights=weights, k=1)[0]
                user_df.at[index, 'Average Latitude'] = user_tuples_df.loc[selected_row, 'Average Latitude']
                user_df.at[index, 'Average Longitude'] = user_tuples_df.loc[selected_row, 'Average Longitude']

final_user_dataframes
```

[12]

```
[      User       Date      Time  Average Latitude  Average Longitude
 0        0 2008-10-23  00:00:00         39.970000         116.317000
 1        0 2008-10-23  01:00:00         40.005000         116.324000
 2        0 2008-10-23  02:00:00         39.987000         116.327000
 3        0 2008-10-23  03:00:00         39.984303         116.306918
 4        0 2008-10-23  04:00:00         39.988607         116.310340
...     ...        ...       ...               ...                ...
2995     0 2009-07-05  19:00:00         39.997000         116.328000
2996     0 2009-07-05  20:00:00         40.008000         116.320000
2997     0 2009-07-05  21:00:00         40.005000         116.324000
2998     0 2009-07-05  22:00:00         39.987000         116.336000
2999     0 2009-07-05  23:00:00         40.005000         116.322000

 [3000 rows x 5 columns],
```

We then apply weighted random sampling to fill in the missing values (Rows with 0 Latitude and Longitude values). This way, we have values for each time interval between Tmin and Tmax for all days for all users.

This is one way to make the data uniform and fill in the missing values in the original dataset.

CLUSTERING

DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a popular clustering algorithm in machine learning and data analysis. It was introduced by Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu in 1996. DBSCAN is designed to discover clusters of data points in a dataset based on the density of points in the feature space rather than assuming that clusters have a specific geometric shape.

DBSCAN identifies clusters based on the density of data points. It defines a cluster as a dense region of data points separated by areas of lower point density.

Here are the different types of points identified by DBSCAN Algorithm:

- **Core points**: A core point is a data point that has at least a specified number of neighboring data points (a specified minimum number of points within a certain distance), including itself. These core points are at the heart of clusters.
- **Border points**: Border points are data points within the specified distance of a core point but do not have enough neighbors to be considered core points themselves. They are part of a cluster but not at its core.
- **Noise points**: Noise points are data points that are neither core points nor border points. These are considered outliers or noise in the dataset and do not belong to any cluster.

Parameters:

- **Radius (eps)**: Defines the maximum distance between two points for one to be considered a neighbor of the other.
- **Minimum number of points (minPts)**: Minimum number of points required to form a core point.

Choosing appropriate values for these parameters can affect the results of DBSCAN, and parameter tuning may be necessary to achieve the desired clustering outcome.

Algorithm Steps:

- Start with an arbitrary data point.
- If the data point has at least MinPts points within its Epsilon neighborhood, create a cluster, and recursively add all connected core points to the cluster.
- If the data point is not a core point, but it is within the Epsilon neighborhood of a core point, consider it a border point and associate it with the cluster.
- Repeat the process until all points have been visited.

Psuedo Code:

DBSCAN(Dataset, Epsilon, MinPts):

```
for each data point P in Dataset:
    if P is not visited:
        Mark P as visited
        NeighborPts = regionQuery(P, Epsilon)
        if size(NeighborPts) < MinPts:
            Mark P as Noise
        else:
            C = next cluster
            expandCluster(P, NeighborPts, C, Epsilon, MinPts)
```

expandCluster(P, NeighborPts, C, Epsilon, MinPts):

```
Add P to cluster C
for each data point P' in NeighborPts:
    if P' is not visited:
        Mark P' as visited
        NeighborPts' = regionQuery(P', Epsilon)
        if size(NeighborPts′) >= MinPts:
            NeighborPts =NeighborPts ∪ NeighborPts'
    if P' is not yet a member of any cluster:
        Add P' to cluster C
```

regionQuery(P, Epsilon):

```
return all data points within the Epsilon neighborhood of P
```

In this pseudocode, **regionQuery** is a function that returns all the points within the Epsilon neighborhood of a given point. The **expandCluster** function is used to grow a cluster by recursively adding connected core points.

Output:

The algorithm produces a set of clusters, where a cluster is a set of dense, connected points.

DBSCAN has several advantages, including its ability to discover clusters of arbitrary shapes and handle noise in the data. Unlike other clustering algorithms like K-means, it does not require specifying the number of clusters beforehand.

Goodness of Clustering

The goodness of clustering, also known as cluster validation or cluster evaluation, is a way to assess the quality of the clusters produced by a clustering algorithm. There are several methods and metrics to measure the goodness of clustering, and the choice of metric depends on the characteristics of your data and the goals of your analysis.
The metric we will use to judge the clustering quality is the **Silhouette Score**.

Silhouette Score:

The Silhouette Score is a cluster validation metric used to measure the quality of clusters formed by a clustering algorithm, such as K-means or DBSCAN. It provides a way to assess how well-separated the clusters are and whether data points are correctly assigned to the clusters.

The Silhouette Score is calculated for each data point in the dataset and **ranges from -1 to 1**. The score is computed as follows:

1. For a data point, calculate the average distance **(a)** between that point and all other data points in the same cluster. The smaller the value of "a," the better, as it indicates that the point is close to other points in its cluster.

2. Calculate the average distance **(b)** between the data point and all data points in the nearest neighboring cluster that the point does not belong to. This nearest neighboring cluster is the cluster other than the one the point is assigned to. The smaller the value of "b," the better, as it indicates that the point is not very close to points in the neighboring clusters.

3. Calculate the **Silhouette Score (S)** for the data point using the formula:

$$S = (b - a) / max(a, b)$$

   This score measures how similar the data point is to its own cluster (a) compared to the nearest neighboring cluster (b). A Silhouette Score close to 1 indicates that the data point is well-matched to its own cluster and poorly matched to neighboring clusters, meaning that the clustering is appropriate. A score close to -1 suggests that the data point is closer to a neighboring cluster and may be assigned to the wrong cluster, while a score close to 0 indicates that the data point is on or very close to the boundary between clusters.

4. To obtain the overall Silhouette Score for the entire dataset, compute the average Silhouette Score of all data points.

In summary, the Silhouette Score measures the **cohesion (a)** and **separation (b)** of clusters. It ranges from **-1 (poor clustering)** to **1 (well-defined, distinct clusters)**, with higher values indicating better clustering. When interpreting the Silhouette Score, remember that it is most useful when you have a priori knowledge of the number of clusters in your data, as you can compare different clustering algorithms or hyperparameter settings based on this metric to determine which one yields the best clustering results.

We apply clustering on both the raw data and flattened or processed data.

For our analysis, we would use the following parameters:

eps = 0.1 and min_samples = 2.

Clustering on the Raw Data

CLUSTERING ON THE ORIGINAL DATA

```
# Define the maximum number of users to process (n)
max_users = 10
user_count = 0

for user_df in user_dataframes:
    user = user_df['User'].iloc[0]
    X = user_df[['Average Latitude', 'Average Longitude']]

    # Perform DBSCAN clustering without specifying the number of clusters
    dbscan = DBSCAN(eps=0.1, min_samples=2)
    labels = dbscan.fit_predict(X)
    # Determine the number of clusters found by DBSCAN
    num_clusters = len(set(labels)) - (1 if -1 in labels else 0)  # -1 represents noise points

    plt.figure(figsize=(8, 6))
    plt.scatter(X['Average Latitude'], X['Average Longitude'], c=labels, cmap='viridis')
    plt.title(f'User {user} Clustering (Number of Clusters: {num_clusters})', color='white')
    plt.xlabel('Latitude', color='white')
    plt.ylabel('Longitude', color='white')
    plt.xticks(color='white')
    plt.yticks(color='white')

    # Calculate the goodness of clustering (silhouette score)
    if num_clusters > 1:
        silhouette_avg = silhouette_score(X, labels)
        print(f'User {user} - Silhouette Score: {silhouette_avg}')
    else:
        print(f'User {user} - No meaningful clustering found.')

    plt.show()

    user_count += 1
    if user_count >= max_users:
        break
```

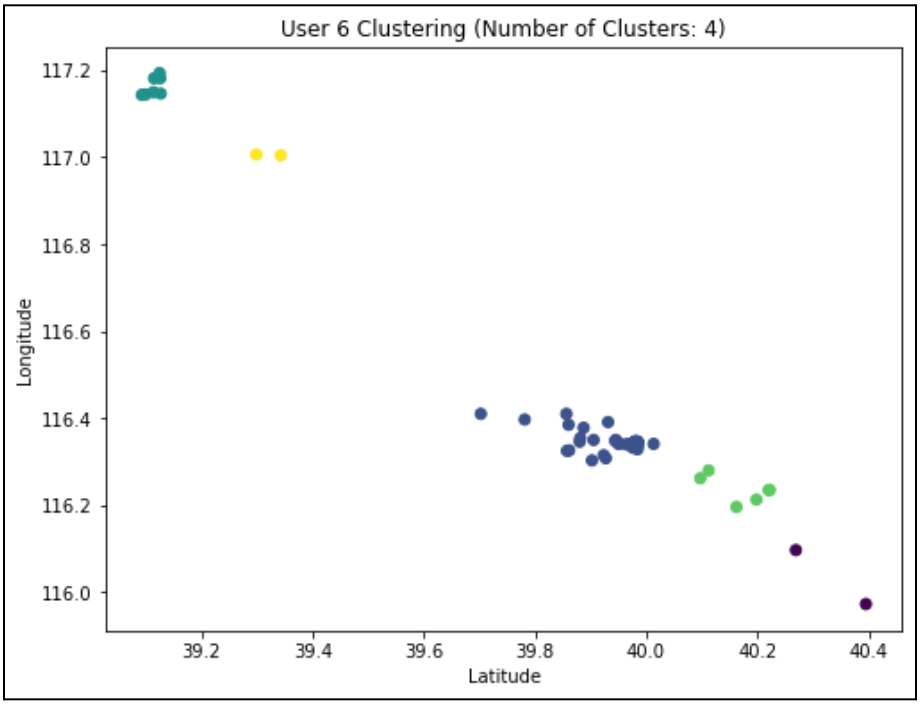
[14]

We find the number of clusters and corresponding silhouette score for clustering the original data for each user.

DBSCAN Algorithm assigns cluster labels to each data point in the User's DataFrame. Data Points belonging to the same cluster are given the same label. The Noise Points are assigned a '-1' cluster label. Noise points would not form a cluster; thus, if noise points are present in the data, we subtract 1 from the total number of unique labels or unique clusters to get the actual number of clusters formed.

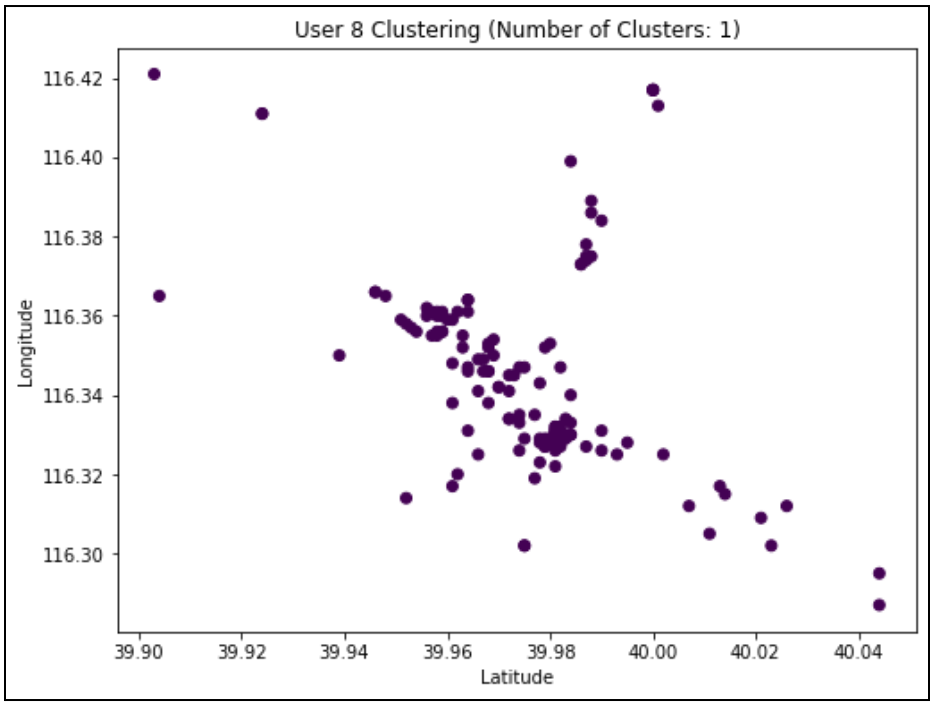
Calculating the Goodness of Clustering or the Silhouette Score will only make sense if the Number of Clusters is greater than 1. Otherwise, if the Number of Clusters formed is 1, then it would mean that the clustering was not good enough; thus, it added all the data points in a single cluster.

Sample Clustering for User 6



Silhouette Score: 0.8228366625281726
Has 4 clusters and few noise points.

Sample Clustering for User 8



Silhouette Score: NaN
It has no meaningful clustering and thus has all data in a single cluster.

Clustering on the Processed data

CLUSTERING ON THE PROCESSED DATA

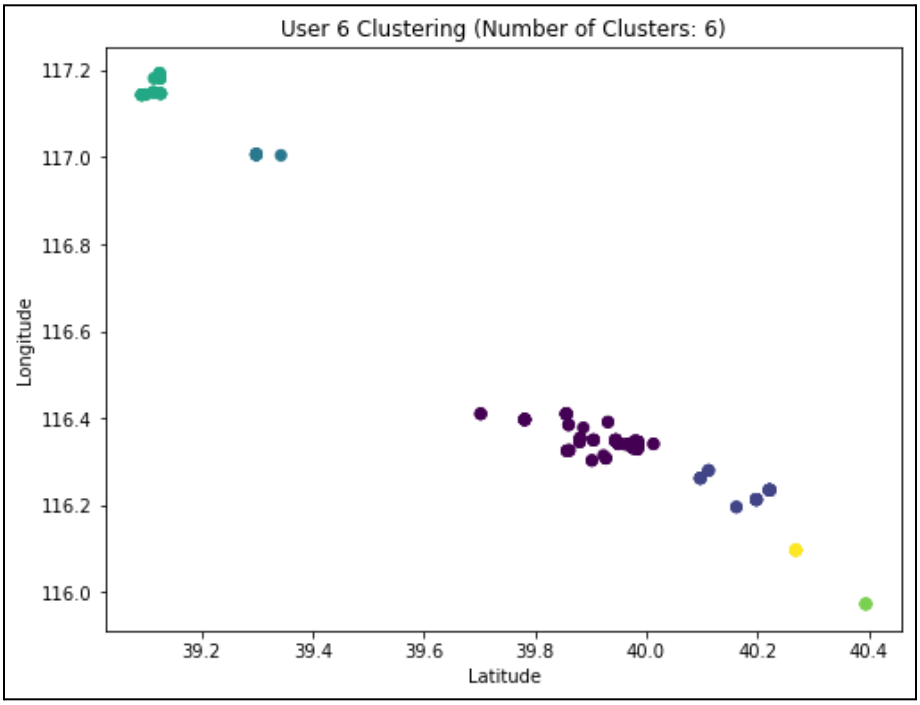
```
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score

# Define the maximum number of users to process (n)
max_users = 10
user_count = 0

for user_df in final_user_dataframes:
    user = user_df['User'].iloc[0]
    X = user_df[['Average Latitude', 'Average Longitude']]

    # Perform DBSCAN clustering without specifying the number of clusters
    dbscan = DBSCAN(eps=0.1, min_samples=2)
    labels = dbscan.fit_predict(X)
    # Determine the number of clusters found by DBSCAN
    num_clusters = len(set(labels)) - (1 if -1 in labels else 0)  # -1 represents noise points

    plt.figure(figsize=(8, 6))
    plt.scatter(X['Average Latitude'], X['Average Longitude'], c=labels, cmap='viridis')
    plt.title(f'User {user} Clustering (Number of Clusters: {num_clusters})', color='white')
    plt.xlabel('Latitude', color='white')
    plt.ylabel('Longitude', color='white')
    plt.xticks(color='white')
    plt.yticks(color='white')

    # Calculate the goodness of clustering (silhouette score)
    if num_clusters > 1:
        silhouette_avg = silhouette_score(X, labels)
        print(f'User {user} - Silhouette Score: {silhouette_avg}')
    else:
        print(f'User {user} - No meaningful clustering found.')

    plt.show()

    user_count += 1
    if user_count >= max_users:
        break
```

[13]

We find the number of clusters and corresponding silhouette score for clustering the processed or flattened data for each user.

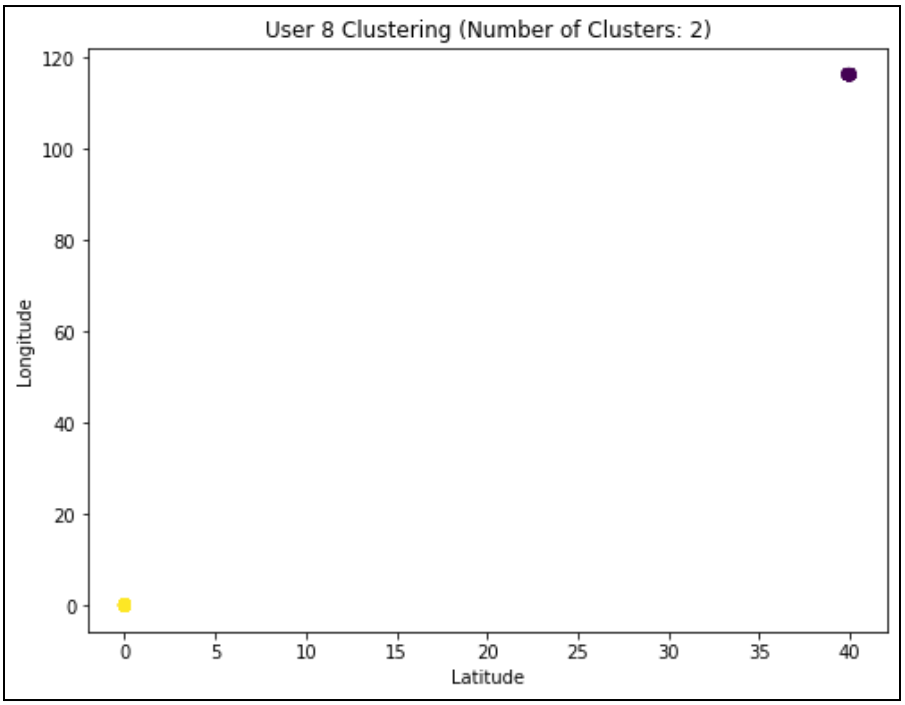
Now if we find that the number of clusters in processed data is equal to or greater than the number of clusters in the original data, then we could say that the clustering is successful and additional clusters are formed due to noise points being clustered in the processed data.
In the processed data, we fill in the gaps in the data; thus, we might get more points initially termed as Noise Points while clustering on the raw data. Now, since we might have more number of data points, they can create a new cluster, thus increasing the total number of clusters.

Sample Clustering for User 6



Silhouette Score: 0.8728271128226047
Has 6 clusters and no noise points.

Sample Clustering for User 8



Silhouette Score: 0.999787531855549
Has 2 clusters.

Table of Clustering

Now, we tabulate the Number of Clusters and Goodness Score or the Silhouette Score for both the raw and processed data.

TABLE OF CLUSTERING

```
results = []

for user_df_original, user_df_processed in zip(user_dataframes, final_user_dataframes):
    user = user_df_original['User'].iloc[0]

    # Original Data
    X_original = user_df_original[['Average Latitude', 'Average Longitude']]
    dbscan_original = DBSCAN(eps=0.1, min_samples=2)
    labels_original = dbscan_original.fit_predict(X_original)
    num_clusters_original = len(set(labels_original)) - (1 if -1 in labels_original else 0)  # -1 represents noise points
    if num_clusters_original > 1:
        silhouette_avg_original = silhouette_score(X_original, labels_original)
    else:
        silhouette_avg_original = 0

    # Processed Data
    X_processed = user_df_processed[['Average Latitude', 'Average Longitude']]
    dbscan_processed = DBSCAN(eps=0.1, min_samples=2)
    labels_processed = dbscan_processed.fit_predict(X_processed)
    num_clusters_processed = len(set(labels_processed)) - (1 if -1 in labels_processed else 0)  # -1 represents noise points
    if num_clusters_processed > 1:
        silhouette_avg_processed = silhouette_score(X_processed, labels_processed)
    else:
        silhouette_avg_processed = 0

    # Append results to the list
    results.append({
        'User': user,
        'NumClustersOriginal': num_clusters_original,
        'GoodnessOriginal': silhouette_avg_original,
        'NumClustersProcessed': num_clusters_processed,
        'GoodnessProcessed': silhouette_avg_processed
    })

results_df = pd.DataFrame(results)
results_df
```

[15]

Data Points belonging to the same cluster are given the same label.
If there are **Noise Points** present in the data, they are given **"-1"** as the label.

Noise points would not form a cluster; thus, if noise points are present in the data, we subtract 1 from the total number of unique labels or unique clusters to get the actual number of clusters formed.

If the number of clusters is greater than 1, we proceed with calculating the Silhouette Score.

	User	NumClustersOriginal	GoodnessOriginal	NumClustersProcessed	GoodnessProcessed
0	0	4	0.746127	4	0.768088
1	1	1	0.000000	1	0.000000
2	2	3	0.824716	10	0.845091
3	3	12	0.938734	38	0.830346
4	4	5	0.827468	16	0.831595
5	5	3	0.884870	4	0.896510
6	6	4	0.822837	6	0.888301
7	7	3	0.779377	5	0.845198
8	8	1	0.000000	2	0.999791
9	9	1	0.000000	1	0.000000
10	10	72	-0.151832	311	0.481654
11	11	1	0.000000	2	0.860334
12	12	16	0.726845	51	0.893192
13	13	5	0.891356	7	0.921809
14	14	2	0.806538	4	0.728507
15	15	1	0.000000	2	0.999507

We can see that the number of clusters in processed data is always greater than or equal to the number of clusters in the original data. This means that the clustering is yielding good results.

CROSS-USER SIMILARITY

Jaccard Similarity Metric

Jaccard Similarity is a common proximity measurement used to compute the similarity between two objects, such as two text documents. Jaccard similarity can be used to find the similarity between two asymmetric binary vectors or to find the similarity between two sets. In literature, Jaccard similarity, symbolized by J, can also be referred to as Jaccard Index, Jaccard Coefficient, Jaccard Dissimilarity, and Jaccard Distance.

Jaccard Similarity is frequently used in data science applications. Example use cases for Jaccard Similarity:

- **Text mining:** Find the similarity between two text documents using the number of terms used in both documents.
- **E-Commerce:** From a market database of thousands of customers and millions of items, find similar customers via their purchase history.
- **Recommendation System:** Movie recommendation algorithms employ the Jaccard Coefficient to find similar customers if they rented or rated many of the same movies.

Jaccard similarity is a measure of similarity between two sets. In the context of clustering, it is often used to assess the similarity between clusters or the similarity of data points within clusters.

The Jaccard similarity coefficient is defined as the size of the intersection of the sets divided by the size of the union of the sets. In the context of clustering, you might use the Jaccard similarity to compare the similarity between two clusters.

If you have two clusters, C1 and C2, you can calculate the Jaccard similarity as:

$$\mathbf{J(C1, C2)} = \frac{|C1 \cap C2|}{|C1 \cup C2|}$$

Here, $(C1 \cap C2)$ represents the number of clusters that are the same (overlapping) for User1 and User2. $(C1 \cup C2)$ represents the total number of clusters (combined) for User1 and User2.

This gives us a score value between 0 to 1. Where 1 signifies high similarity between the users, and 0 signifies low similarity between the users.

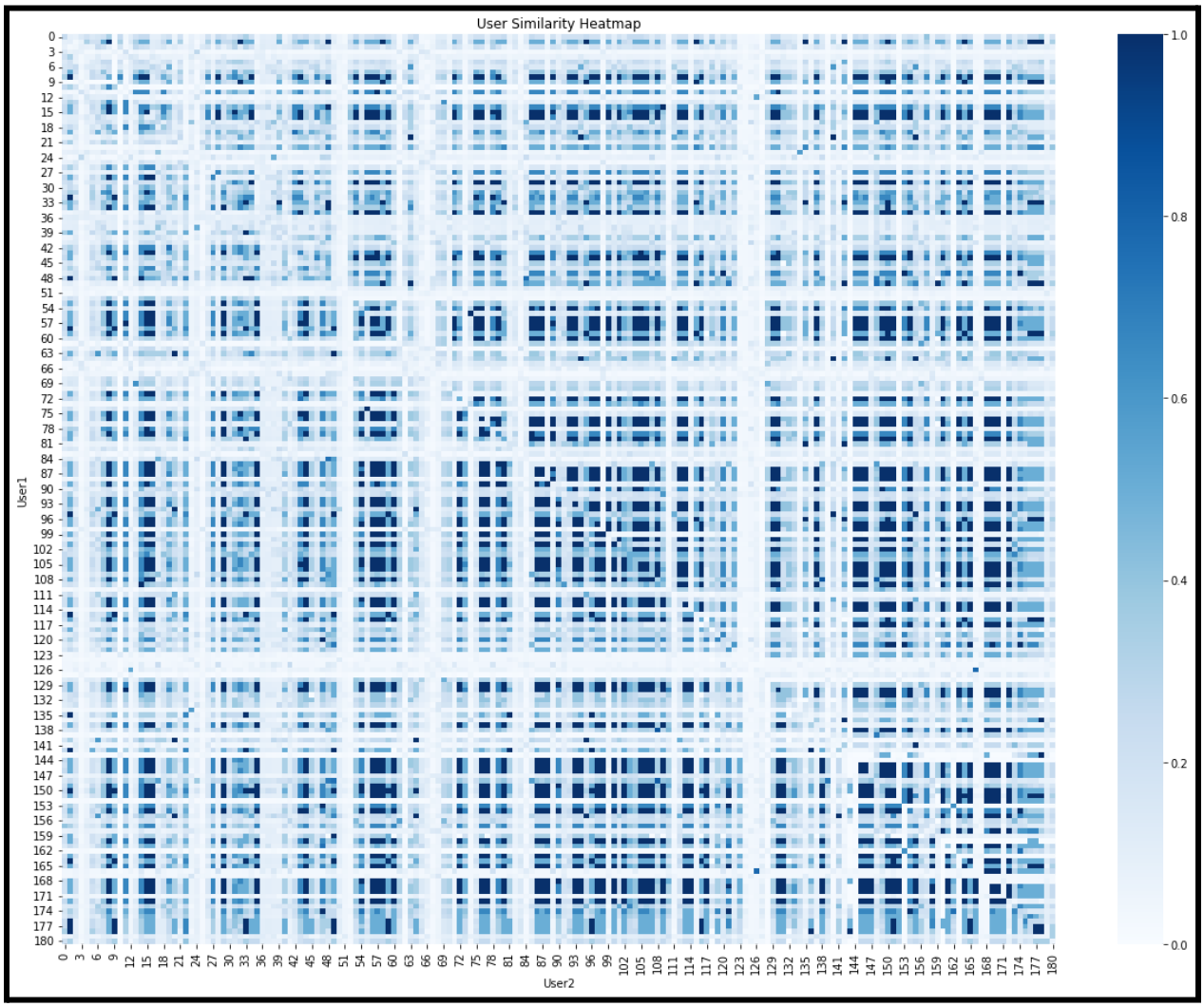
This can help in assessing how much overlap or similarity there is between the elements of the two clusters.

Heatmap

We now combine the processed data for all possible pairs of users i.e. ${}^{182}C_2$ combinations. We then calculate the Number of Clusters and Goodness of Clustering in the combined plot. We also calculate the similarity score based on the Jaccard Similarity Metric for all the pairs of users.

To get an overview of the extent of similarity between the users, we use the seaborn library of Python to generate a heatmap.

Here, similar users are represented with darker shades of Blue color. Whereas, lower similarity users are represented with lighter shades of Blue color.

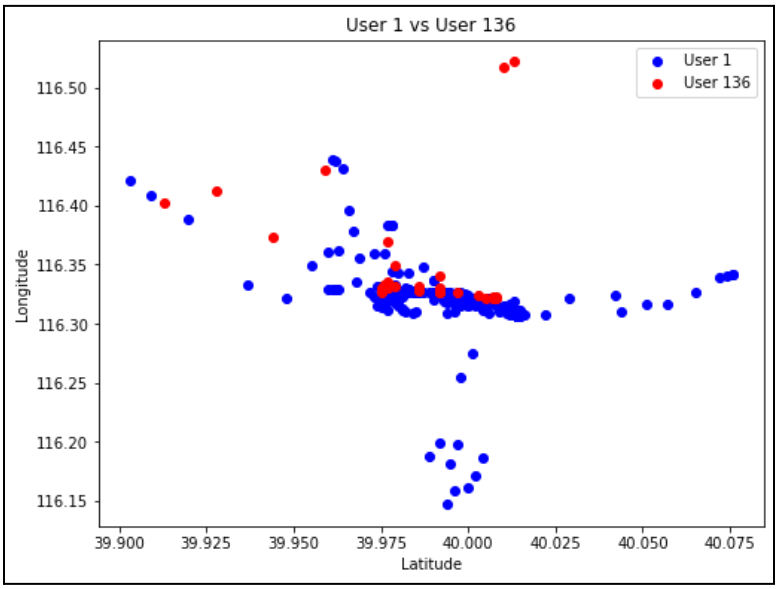


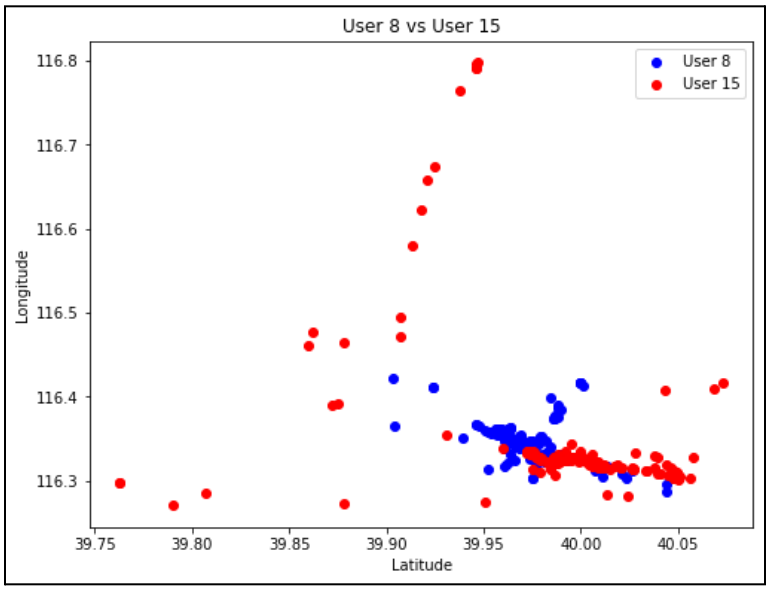
Similarity Score = 1 Plots

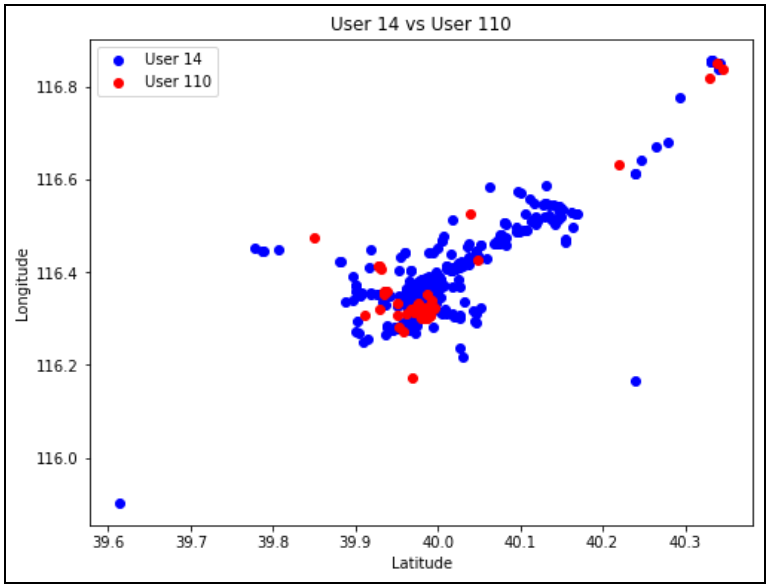
<u>Condition</u>: $(C1 \cap C2) = (C1 \cup C2)$

The plots are based on the original or raw data.

Example: User1 and User136; User8 and User15; User14 and User110.







For all the pairs of users above, we can see that the Number of Clusters in the Combined Plot is 1, and the Goodness Of Clustering of the Combined Plot is NaN.

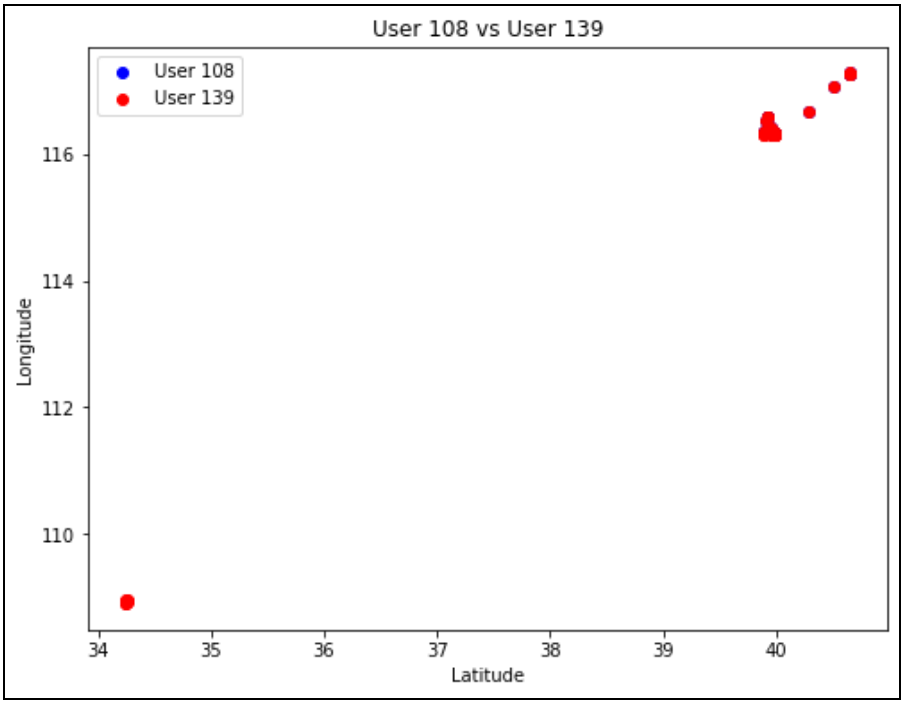
This might be because of the fact that DBSCAN relies on density-based clustering, and if trajectories have varying densities, it may struggle to separate them effectively. We can see the merging of distinct trajectories into a single cluster because of the parameters(epsilon and minPoints) we have set.

High Similarity Score Plots

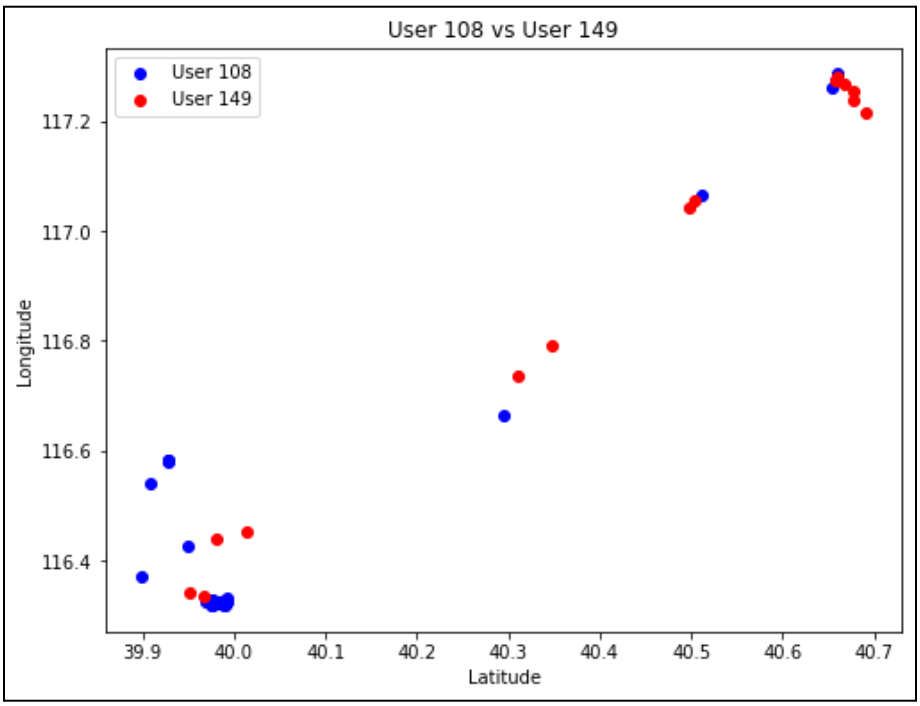
To get more insights into the Similarity Score, we also capture the pair of users having relatively high similarity scores i.e. similarity score value lying in the range (0.75, 1).

The plots are based on the original or raw data.

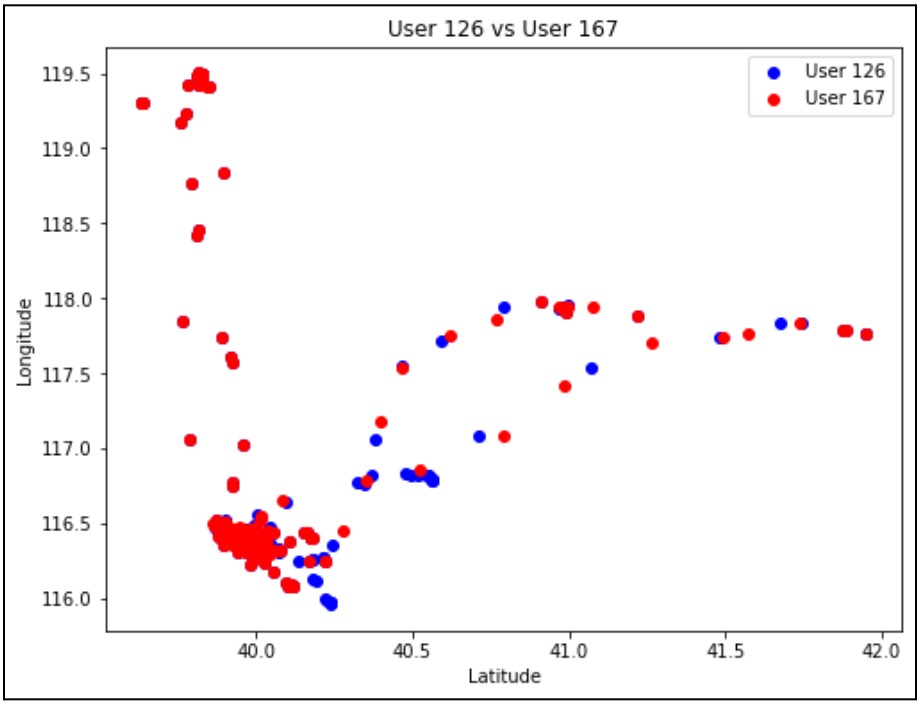
Example: User108 and User139; User108 and User149; User126 and User167.



Number of Clusters: 6
Silhouette Score: 0.8997686433150279
Similarity Score: 0.857143



Number of Clusters: 5
Silhouette Score: 0.8227063359201182
Similarity Score: 0.833333



Number of Clusters: 26
Silhouette Score: 0.7815350570061455
Similarity Score: 0.781250

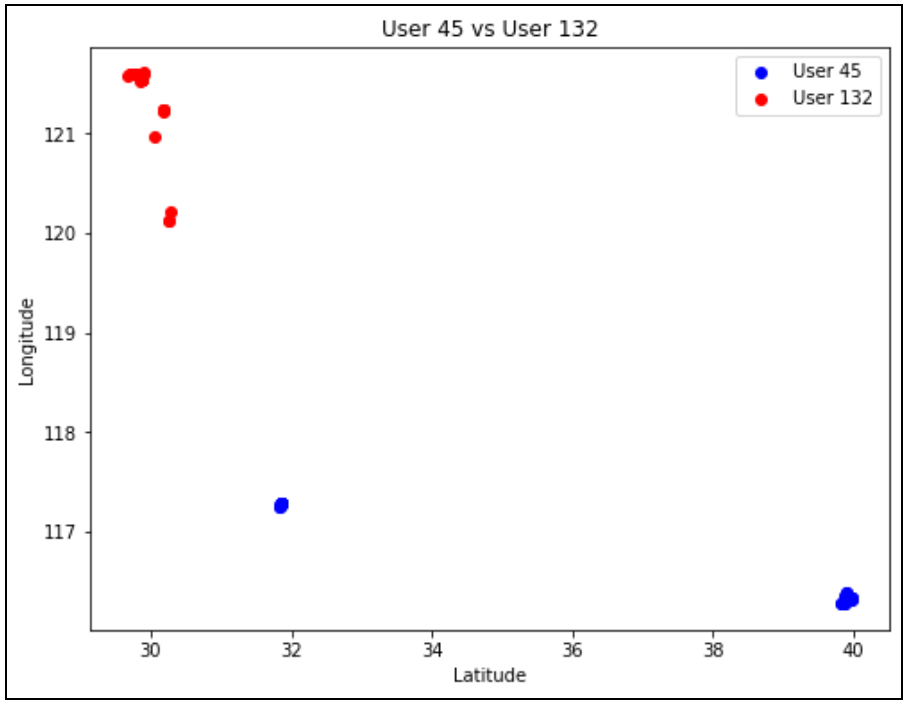
Here, we can observe that the pair of users follow similar trajectories, resulting in higher similarity scores.

Similarity Score = 0 Plots

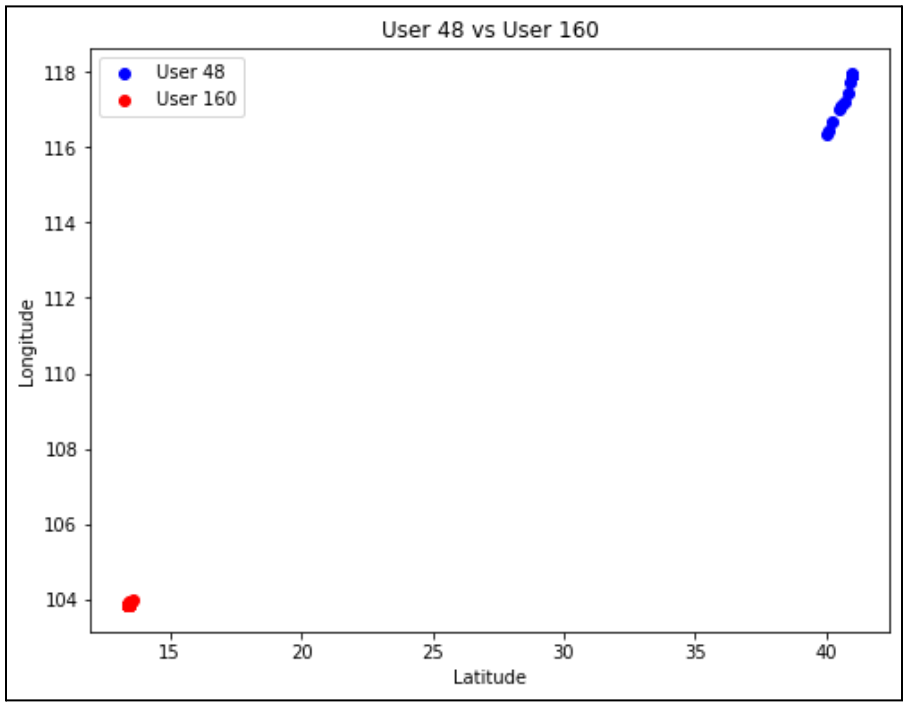
<u>Condition</u>: $(C1 \cap C2) = 0$

The plots are based on the original or raw data.

Example: User45 and User132; User48 and User160.



Number of Clusters: 5
Silhouette Score: 0.9205268942052811

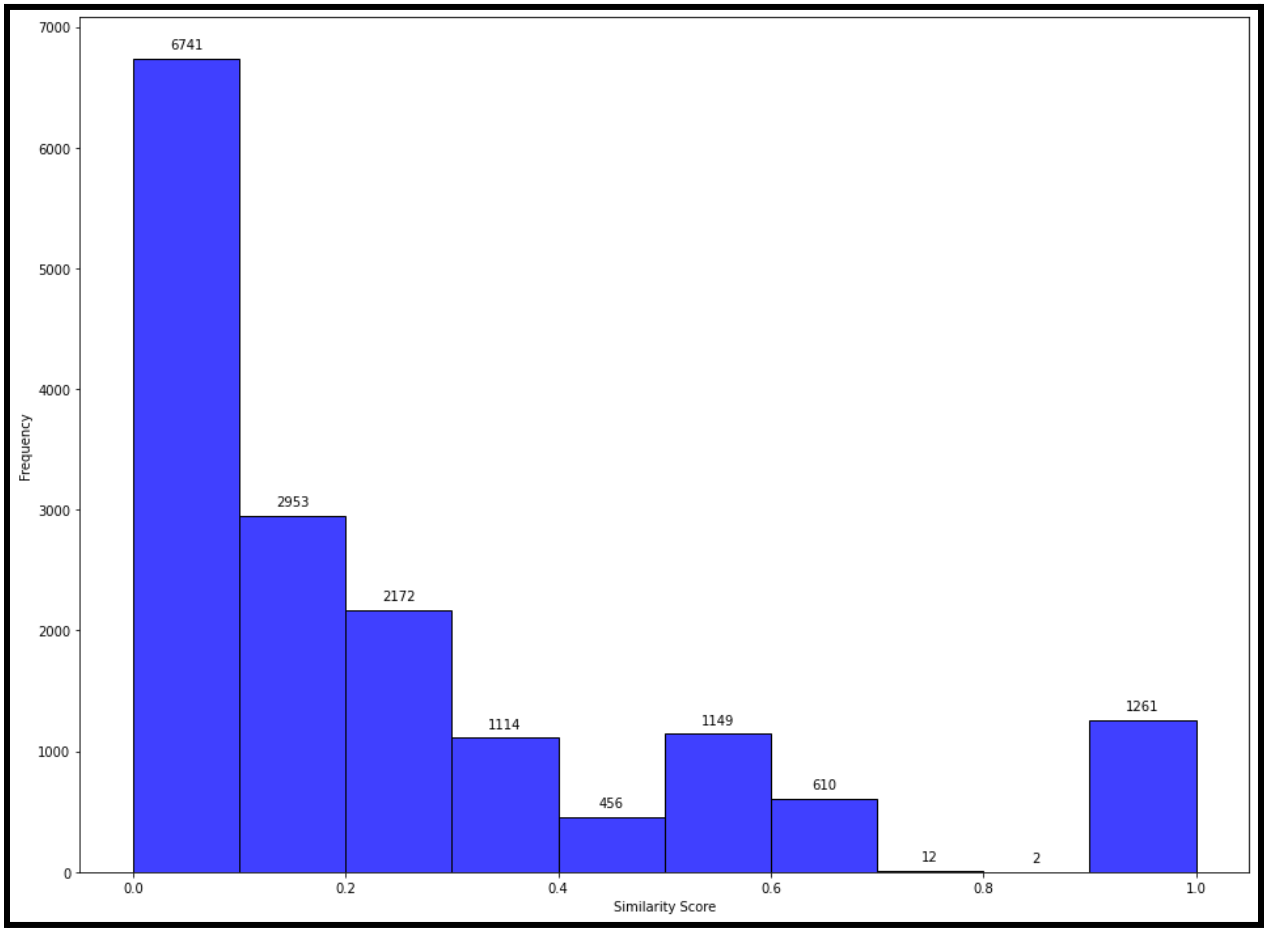


Number of Clusters: 2
Silhouette Score: 0.8733012987224681

Here, we can observe that the pair of users follow completely different trajectories (no overlap at all), resulting in a similarity score of zero.

Histogram

To get insights about the range and spread of similarity scores across different pairs of users, we use the Seaborn library of Python to plot a histogram.



CONCLUSION

In conclusion, addressing missing data in time series datasets is a critical challenge for advanced data analysis. One commonly employed solution is data imputation, where the primary difficulty lies in determining appropriate values to fill in for missing data points.

In the context of the Geolife Trajectory Dataset, this project successfully implemented a method of data imputation. The approach began by flattening the data for each user based on time values. Weight values were then computed, considering the frequency of occurrence of specific (latitude, longitude) pairs within defined time intervals across all days for a given user. These weight values were subsequently used to perform weighted random sampling on the pre-processed data, effectively filling in the missing values for that user.

Additionally, the project involved the application of the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm to form clusters for each user. These clusters were created based on latitude and longitude values for the user across all days, both on raw and processed data. Furthermore, the quality of the clustering was evaluated using the Silhouette score, providing a measure of the goodness of the clustering for each user.

Overall, this project demonstrates a robust approach to address missing data in the Geolife Trajectory Dataset, combining data imputation and spatial clustering techniques to enhance the dataset's utility for advanced data analysis.

Further, we can combine the clusters formed for different users and generate a similarity score matrix for all the users to get an idea about similar users. Finally, this can be used as follows: Say we want to predict the missing data for a particular target user, then we can use the data present of the user which is similar to our target user to fill in the gaps in data and predict the location of the target user.

REFERENCES

[1] Geolife GPS trajectory dataset - User Guide - Microsoft Research

[2] What is time series data? | InfluxData.

[3] 4 Techniques to Handle Missing values in Time Series Data | by Satyam Kumar

[4] What is the weighted random selection algorithm?

[5] DBSCAN Clustering in ML | Density based clustering - GeeksforGeeks

[6] Silhouette Coefficient. This is my first medium story, so… | by Ashutosh Bhardwaj | Towards Data Science.

[7] Jaccard Similarity – LearnDataSci