CityMomentum - Crowd Behavior Prediction

The premise of the article is a tragic event that happened in Shanghai on New Year's Eve of 2014.

During the celebration of the new year, on 31 December 2014, a deadly stampede occurred in Shanghai, near Chen Yi Square, where around 300,000 people had gathered for the New Year celebration. Thirty-six people were killed and another 49 were injured, 13 of them seriously (see appendix). It hits close to home as we had the disastrous stampede in Mount Meron in 2021.

As statisticians, we can use our knowledge to prevent horrific events like this from happening.

The article "CityMomentum" (2015) used well-known methods and adjusted these tools for human movement and crowd dynamics. The article used the following methods: Hidden Markov Models and MCMC method for sampling, specifically Gibbs sampling.

First of all, we set the stage with some underlying assumptions for our model:

Definition 1 (GPS **log** data):

The original dataset can be described by a set of 4-tuples:

$$X = \{(u, \tau, \text{lat}, \text{lon})\}$$

When looking at GPS records:

- u - unique mobile phone user ID
- τ - timestamp
- lat - latitude
- lon - longitude

Definition 2 (**Momentary** movement):

We denote X_t as the subset of the original GPS log dataset that contains all the **previous** records with timestamps ranging from $t - \Delta t$ to t:

$$X_t = \{x \in X \quad | \quad t - \Delta t < x.\tau \leq t\} \subset X$$

Definition 3 (**Predicted** movement):

The predicted movement at time t is the collection of **predicted** records from t to $t + \Delta t$, which share the same data structure of GPS log data:

$$\hat{X}_t = \{(\hat{u}, \hat{\tau}, \hat{\text{lat}}, \hat{\text{lon}}) \quad | \quad t < \hat{\tau} \leq t + \Delta t\}$$

Definition 4 (**Displacement**):

We define an operator T to extract the **displacement** segments of each subject in terms of momentary movement X_t:

$$T_u(X_t) = \{(x_o, x_d) \quad | \quad x_o, x_d \in X_t \cap x_o.u = x_d.u = u \cap C\}$$

- x_o - records of the origin
- x_d - records of the destination
- C is the consequential condition that guarantees each transition is defined based on two temporally **consecutive** records from the **same** subject, that is:

$$C = (x_o.\tau < x_d.\tau) \cap (\nexists x \in X_t \quad \text{s.t.} \quad x_o.\tau < x.\tau < x_d.\tau)$$

Definition 5 (Subject cluster):

To **maximize** the predictability of human flow—clusters of mobile phone users with **similar** momentary movements at time t, which we denote as Z_t:

$$Z_t = \{z_u^t = k \quad | \quad u \in U_t,\ k \in \{1, \ldots, K\}\}$$

- where the allocation of the subject u to cluster $k \in \{1, \ldots, K\}$ at time t is denoted as z_u^t. K is the predefined number of clusters.

Definition 6 (CityMomentum model):

The CityMomentum model is the online updating predictive model of human movement, which gives the **momentary** movement that we observed and the **latent cluster** assignment of each subject. The future human movement is predicted by a sampling process that considers the conditional distribution: $p(\hat{X}_t | Z_t, X_t)$.

We can notice that these definitions are very similar to the definitions of the Hidden Markov Model.

The Hidden Markov Model is a statistical model that can be used to describe the movement of a subject in a sequence of states. The states are hidden and can only be inferred from the observations, in an online updating manner.

The observations are the GPS log data. The CityMomentum model is a special case of the Hidden Markov Model that is tailored to the movement of human subjects in a city. This can be achieved by first clustering the subjects based on their momentary movements and then predicting the future movement of each subject based on the movement of other subjects in the shared cluster.

The model proposed in the article has two stages:

1. **Clustering**: The subjects are clustered based on their momentary movements at time t. The clustering is done using the Gibbs sampling method, which is a Markov Chain Monte Carlo (MCMC) method. The Gibbs sampling method is used to sample from the conditional distribution of the latent cluster assignment of each subject given the momentary movements of all subjects at time t. The number of clusters K is predefined. We're going to address the issue of choosing the number of clusters in the section (C).

2. **Prediction**: The future movement of each subject is predicted based on the movement of other subjects in the shared cluster. The prediction is done using the Hidden Markov Model. The Hidden Markov Model is used to model the transition of each subject between the states (locations) over time. The transition matrix of the Hidden Markov Model is estimated using the momentary movements of all subjects in the same cluster.

Let's delve into the probabilistic model of the CityMomentum model.

We have this equation for the joint distribution of the predicted movement, the momentary movement, and the latent cluster assignment, from simple probability theory:

$$p\left(\hat{X}_{1:t}, X_{1:t}, Z_{1:t}\right) = \left(\prod_{\tau=1}^{t} p(X_\tau|Z_\tau)\right) \cdot \left(p(Z_1)\prod_{\tau=2}^{t} p(Z_\tau|Z_{\tau-1})\right) \cdot \left(\prod_{\tau=1}^{t} p(\hat{X}_\tau|X_\tau, Z_\tau)\right)$$

Keep in mind that Z_1 is the initial cluster for our model.

The first and second terms can be regarded as the movement similarity in the same cluster and online weights terms of the clustering phase respectively, and the last is the prediction of human movements in the short-term future (See figure 2).

CityMomentum is an extension of the Hidden Markov Model. We do it by looking at displacements.

The displacement of a subject is the movement of the subject from one location to another, at different times.

Each subject is conditionally independent given the momentary movement. To determine the predicted movement for each subject, we utilize a random walk on a mobility graph constructed by X_t, with transition distribution:

$$p(\hat{l}_{i+1}, \Delta\hat{\tau}|\hat{l}_i, X_t) = p(\hat{l}_{i+1}|\hat{l}_i, X_t) \cdot p(\Delta\hat{\tau}|\hat{l}_{i+1}, \hat{l}_i, X_t) = \frac{\eta_{\hat{l}_{i+1},\hat{l}_i}}{\sum_l \eta_{l,\hat{l}_i}} \cdot \frac{\eta_{\Delta\hat{\tau},\hat{l}_i}}{\sum_l \eta_{l,\hat{l}_i}}$$

- $\hat{l}_i$ - predicted location
- $\eta_{\hat{l}_{i+1},\hat{l}_i}$ - number of displacements from $\hat{l}_i$ to $\hat{l}_{i+1}$
- $\eta_{\Delta\hat{\tau},\hat{l}_i}$ - the number of displacements starting from $\hat{l}_i$ with a time interval of $\Delta\hat{\tau}$

Intuitively, the predicted location $\hat{l}_{i+1}$ is drawn from a multinomial distribution that counts the occurrences of the endpoints from an origin. Meaning, the predicted location is the most likely location to move to, given the movements in the same cluster. The time interval $\Delta\hat{\tau}$ is drawn from a multinomial distribution that counts the occurrences of the time intervals between the origin and the endpoints.

However, the NMP model relies on the first-order Markovian assumption, which has the limitation of multi-step prediction. For example, people driving a car on the same road are likely to travel in the same direction, rather than making a sudden U-turn. However, a random walk under a first-order Markovian assumption would give an **equal** probability of moving forward and backward if the traffic flows are equivalent on both sides of the road.

Consequently, to maximize the predictability of human movements, the CityMomentum model is based on **mixing** Markov chains instead of a single Markovian assumption. Each of the basic Markov chains is trained, but X_t is partitioned into clusters X_t^k to train the basic predictors separately. Finally, a predicting-by-clustering is implemented.

Model-based clustering involves a widely used family of clustering algorithms, which assume that the data are drawn from a distribution that comprises a mixture of finite **latent** components. Each component includes a cluster of data, a model trained by the cluster of data, and the weight of each component. However, determining the optimal cluster assignment for each subject is a very complex problem. Markov Chain Monte Carlo (MCMC) has been proved to be a powerful technique for effectively estimating the posterior distribution of latent variables (cluster assignment).

We define the posterior distribution of each component k after removing one subject u from U_t as:

$$p(z_u^t = k|Z_{t,-u}, Z_{t-1}, X_t) = \frac{p(z_u^t = k, Z_{t,-u}, Z_{t-1}, X_t)}{p(Z_{t,-u}, Z_{t-1}, X_t)} = \frac{p(X_t|z_u^t = k, Z_{t,-u}, Z_{t-1}) \cdot p(z_u^t = k, Z_{t,-u}, Z_{t-1})}{p(Z_{t,-u}, Z_{t-1}|X_t) \cdot p(X_t)}$$

Where in the last step we ignored the constants from $p(X_{t,-u}|Z_{t,-u}, Z_{t-1})$ and $p(Z_{t,-u}, Z_{t-1})$.

Simplifying:

$$p(z_u^t = k|Z_{t,-u}, Z_{t-1}, X_t) \propto p(X_t|z_u^t = k, Z_{t,-u}, Z_{t-1}) \cdot p(z_u^t = k|Z_{t,-u}, Z_{t-1})$$
$$\propto p(X_{t,u}|z_u^t = k, X_{t,-u}^k) \cdot p(z_u^t = k|Z_{t,-u}, Z_{t-1})$$

- where $X_{t,u}$ is the momentary movement of subject u and $X_{t,-u}^k$ is the momentary movement of all the subjects that belong to cluster k without subject u. Z_t is the clusters at time t and $Z_{t,-u}$ is the clusters at time t without u.

The **first** term indicates the **likelihood** function used to estimate the likelihood of subject u belonging to cluster k, whereas the **latter** is the **weight** of component k. The estimates of these two terms are explained in detail in the following subsections.

Likelihood

For the NMP model, it might appear that it can be adapted to an estimator of the likelihood function. However, given that the sample rate of the dataset is **not** stable and the presence of fast-moving subjects in cars or trains, even small differences in velocity or events such as **stopping** at traffic lights could cause significant geographical disparities with each other. Thus, in the design of the likelihood function, they consider the direction and speed of **displacement** to cluster the subjects with *similar* momentary movement.

$$p(x_d|x_o, Z^k_{t,-u}, X^k_{t,-u}) = p(D|\mathbb{I}(x_o), k) = \frac{n^k_{I(x_o),D(x_d,x_o)} + \gamma}{\sum_D n^k_{I(x_o),D} + L_{I(x_o)} \cdot \gamma}$$

Where we have:

- $I(x)$ - index of the location of GPS record x
- $D(x_d, x_o)$ - type of displacement
- γ - hyperparameter that regulates sparsity—being scattered
- $L_{I(x_o)}$ - number of endpoint locations for all the displacements starting from location indexed by $I(x_o)$

In this experiment, they categorized the displacements into three speed levels (low, medium, high) and four directions (north-east, north-west, south-west, and south-east). Thus, by considering the combination of speed and direction, they set nine types of displacement (we do not consider the direction when the speed is low).

Thus, they construct a random Markov chain, and the likelihood of each subject's trajectory given the momentary movement of cluster k is formulated as a random walk on the chain, for all origin-destination pairs in the displacement segments of the subject:

$$p(X_{t,u}|z^t_u = k, X^k_{t,-u}) \propto \prod_{(x_o,x_d)\in T_u(X_t)} p(x_d|x_o, Z^k_{t,-u}, X^k_{t,-u})$$

Which is the prior likelihood function for our clustering algorithm.

Prior Cluster Distribution

We define the prior cluster distribution (the weights from earlier) as the prior distribution used to draw a cluster label k for every subject. A widely used prior distribution is the discrete Dirichlet distribution, which includes a hyperparameter that describes prior knowledge of the prior distribution. To consider the temporal continuity, we utilize the cluster distribution at the previous time with a damping factor by adding up as the hyperparameter of the Dirichlet-Multinomial Distribution:

$$p(z^t_u = k|Z_{t,-u}, Z_{t-1}) = \frac{n^k_{t,-u} + \alpha n^k_{t-1} + \beta}{n_{t,-u} + \alpha n_{t-1} + \beta K}$$

- $n^k_{t,-u}$ - number of subjects belonging to cluster k except subject u
- $n_{t,-u}$ - total number of subjects at time t except subject u
- α, β - hyperparameters of discrete Dirichlet distribution

Regarding the Dirichlet-Multinomial Distribution, a probability vector p is drawn from a Dirichlet distribution with parameter vector $\boldsymbol{\alpha}$, and an observation is drawn from a multinomial distribution with probability vector p and number of trials n. A known formula which is commonly used for Gibbs sampling is:

$$\Pr(z_n = k \mid \mathbb{Z}^{(-n)}, \boldsymbol{\alpha}) \propto n_k^{(-n)} + \alpha_k$$

Here they took the formula for a single variable and adjusted it for looking at the probability of a specific cluster of a user, conditioned on clusters of other users in the same time point and all clusters in the previous time point, with added regularization.

α can be looked at as a concentration parameter for the previous time (similarly to alpha in Dirichlet distribution).

Lastly, after using these steps we get an intuitive formula for the weights of a specific cluster of a subject when we condition on the subset of all *other* clusters and all the clusters in the **previous** time point.

Predicting-by-clustering

Predict future movements from a mixing model based on the predictability-maximum partition of X_t, i.e., rather than the entire X_t. Considering the conditional independence, we formulate our predicting-by-clustering distribution as:

$$p(\hat{X}_t|Z_t, X_t) = \prod_{u\in U_t} p(\hat{X}_{u,t}|X^k_t)$$

- $k = z^t_u$ is the cluster at time t to which subject u belongs
- X^k_t is the subset of the movement at time t that contains all the GPS records of the **subjects** from cluster k

Therefore, the CityMomentum model involves the mixing of Hidden Markov Models, each of which is a random walk on the mobility graph constructed by X^k_t.

My Implementation

I tried to implement the whole model on the Geolife data from Microsoft. This data consists of 182 users over a period of more than five years (from April 2007 to August 2012).

The main difficulties were with processing the data. This is big data with duplicate times and locations, and deciding which subset to work on was a challenge.

The running time of the model was fast. First, because I used online methods that take less memory and reduce running time. In addition, I used the properties of Python dictionaries, NumPy, and Pandas (with the added benefits of `iterrows` in Pandas). Lastly, we worked with conjugate priors in the cluster weights (Dirichlet-Multinomial Distribution is conjugate prior for multinomial distribution), which reduced computation time.

Numerical Stability

To improve numerical stability and computational efficiency, we added small constants to prevent underflow:

```
prob = max(prob, 1e-10)
```

Log-Posterior Adjustment:

First of all, I worked with log-likelihoods instead of likelihoods to prevent underflow. Moreover, I subtracted the maximum log-posterior value before exponentiating to prevent underflow after exponentiation:

```
log_posterior -= np.max(log_posterior)
```

Moreover, we added regularization parameters to ensure that probability calculations remain robust, even with sparse data.

Attempts to Improve Our Model

Validation

I tried incorporating validation for choosing optimal hyperparameters.

The measures I used for validation:

a. Silhouette Coefficient

- Measures how similar an object is to its own cluster compared to other clusters.
- For each sample i:
  - $a(i)$: Mean distance between i and all other points in the same cluster.
  - $b(i)$: Mean distance between i and all points in the nearest cluster.
  - Silhouette score $s(i) = \frac{b(i)-a(i)}{\max\{a(i),b(i)\}}$.

Compute the average silhouette score across all samples. A higher average score suggests well-separated and cohesive clusters.

b. Adjusted Rand Index (ARI)

Given a set of n elements:

$$S = \{o_1, \ldots, o_n\}$$

and two partitions of S to compare:

- **Partition X**:

  $$X = \{X_1, \ldots, X_r\}$$

  where X is a partition of S into r subsets.

- **Partition Y**:

  $$Y = \{Y_1, \ldots, Y_s\}$$

  where Y is a partition of S into s subsets.

Define the following:

- **a**: The number of pairs of elements in S that are in the **same subset** in both X and Y.

  $$a = \text{Number of pairs } \{o_i, o_j\} \text{ such that } o_i, o_j \in X_k \text{ and } o_i, o_j \in Y_l \text{ for some } k, l.$$

- **b**: The number of pairs of elements in S that are in **different subsets** in both X and Y.

  $$b = \text{Number of pairs } \{o_i, o_j\} \text{ such that } o_i \in X_k, o_j \in X_m \text{ and } o_i \in Y_l, o_j \in Y_n \text{ for some } k \neq m, l \neq n.$$

- **c**: The number of pairs of elements in S that are in the **same subset** in X but in **different subsets** in Y.

  $$c = \text{Number of pairs } \{o_i, o_j\} \text{ such that } o_i, o_j \in X_k \text{ and } o_i \in Y_l, o_j \in Y_m \text{ for some } l \neq m.$$

- **d**: The number of pairs of elements in S that are in **different subsets** in X but in the **same subset** in Y.

  $$d = \text{Number of pairs } \{o_i, o_j\} \text{ such that } o_i \in X_k, o_j \in X_m \text{ and } o_i, o_j \in Y_l \text{ for some } k \neq m.$$

The **Rand Index** measures the similarity between two data clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings.

$$R = \frac{a+b}{\binom{n}{2}} = \frac{a+b}{\frac{n(n-1)}{2}}$$

The **Adjusted Rand Index** adjusts the Rand Index for the chance grouping of elements, providing a measure of the similarity between two clusterings that accounts for random chance.

$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2} - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right]}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right]}$$

Where:

- n_{ij} is the number of elements in the intersection of cluster X_i from partition X and cluster Y_j from partition Y.
- a_i is the number of elements in cluster X_i.
- b_j is the number of elements in cluster Y_j.

The Adjusted Rand Index ranges from -1 to 1, where 1 indicates perfect agreement between the two partitions, 0 indicates random agreement, and negative values indicate less agreement than expected by chance.

When I used these methods, I had some insights but overall it was not beneficial enough for the final results.

Additionally, I plotted the series of cluster assignments to check for convergence of cluster distribution using Gibbs sampling.

Summary and Reflection

Initially, I thought that using a Markov chain for prediction could be convoluted, and I couldn't connect the dots. After experimenting with this article, I realized that HMM methods can be easy to understand and extremely powerful. By using the benefits of Bayesian methods for getting a unique distribution for our target variable, we can predict new movements, utilizing the straightforward properties of Markov chains.

Moving forward, I'd want to test the model on a data with less duplicates and more unique locations, and perform extensive cross validation on the prediction function using EMD (Earth mover's distance).

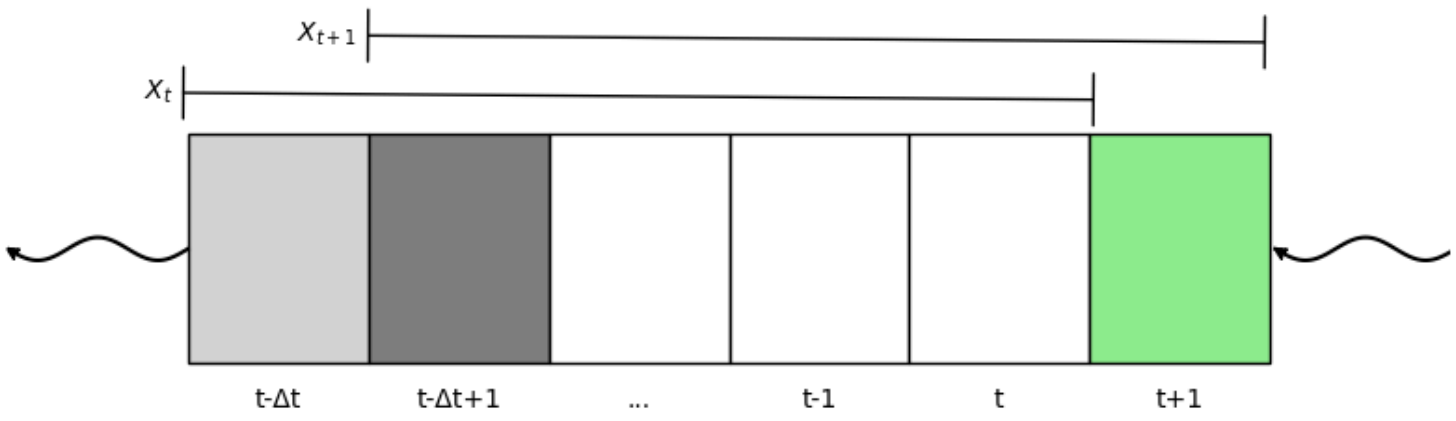


Figure 1: each time point is defined as $t \in [t - \Delta t, t]$

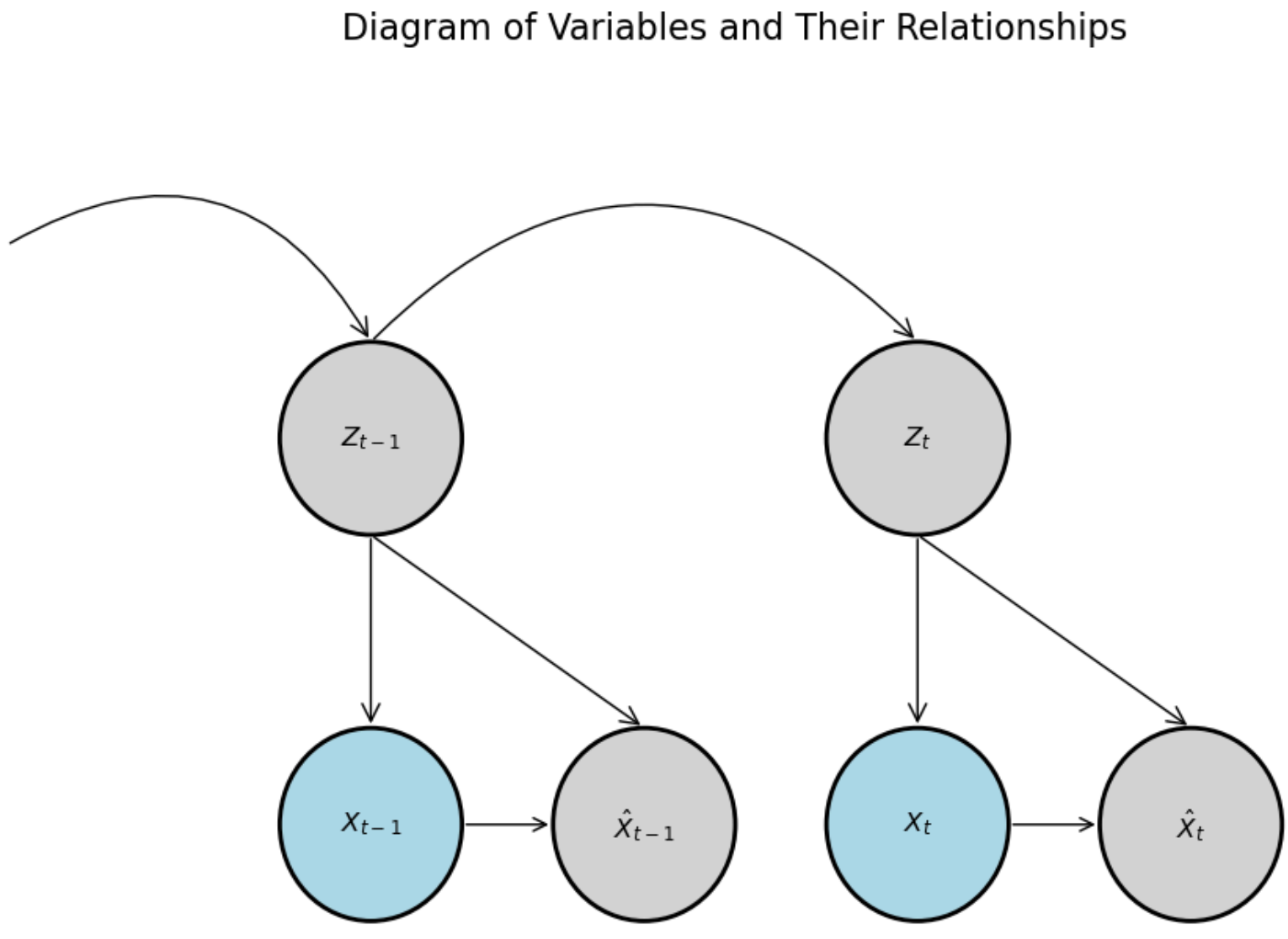
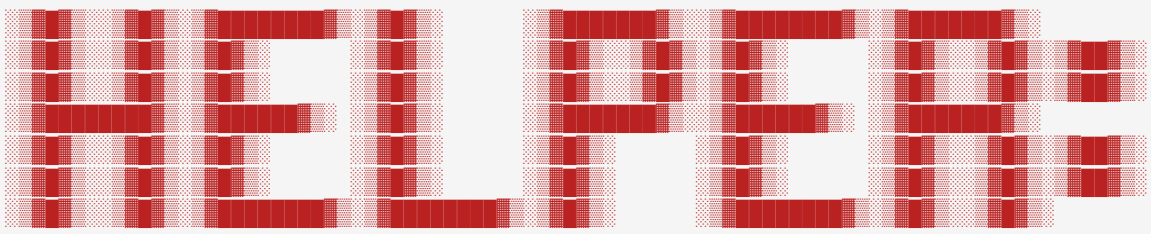


Figure 2: the light blue circles represent the observed data, the gray represent the latent variables

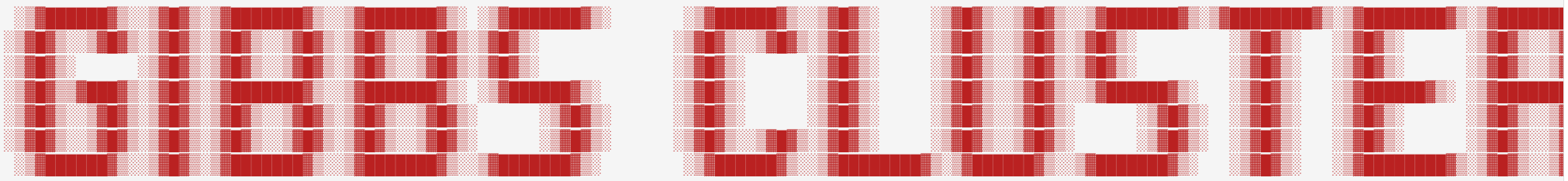
Model code:

In [ ]:
```
import numpy as np
import pandas as pd
from datetime import datetime, timedelta
import logging


"""

"""

# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)




"""


"""

def determine_direction(origin, destination):
    """
    Determine movement direction based on latitude and longitude differences.

    Args:
        origin (tuple): (latitude, longitude) of the origin.
        destination (tuple): (latitude, longitude) of the destination.

    Returns:
        str: Direction string.
    """
    lat1, lon1 = origin
    lat2, lon2 = destination
    delta_lat = lat2 - lat1
    delta_lon = lon2 - lon1

    # Determine direction based on latitude and longitude differences
    if delta_lat > 0 and delta_lon > 0:
        return 'North-East'
    elif delta_lat > 0 and delta_lon < 0:
        return 'North-West'
    elif delta_lat < 0 and delta_lon > 0:
        return 'South-East'
    elif delta_lat < 0 and delta_lon < 0:
        return 'South-West'
    elif delta_lat == 0 and delta_lon > 0:
        return 'East'
    elif delta_lat == 0 and delta_lon < 0:
        return 'West'
    elif delta_lat > 0 and delta_lon == 0:
        return 'North'
    elif delta_lat < 0 and delta_lon == 0:
        return 'South'
    else:
        return 'Undefined'


"""


"""

def categorize_speed(speed, speed_thresholds):
    """
    Categorize speed into Low, Medium, or High based on thresholds.

    Args:
        speed (float): Speed value.
        speed_thresholds (dict): Dictionary with 'low' and 'medium' speed thresholds.

    Returns:
        str: Speed category.
    """
    # Categorize speed based on thresholds
    if speed <= speed_thresholds['low']:
```

```
        return 'Low'
    elif speed <= speed_thresholds['medium']:
        return 'Medium'
    else:
        return 'High'


"""
DISPLACEMENT:
"""

def compute_displacement_type(df, speed_thresholds, decimal_places=4):
    """
    Compute displacement type and origin_id for each movement.

    Args:
        df ->(pd.DataFrame): DataFrame containing Main data
        speed_thresholds ->(dict): Dictionary with 'low' and 'medium' speed thresholds.
        decimal_places ->(int): Number of decimal places to round coordinates to.

    Returns:
        pd.DataFrame: Updated DataFrame with 'speed_level', 'direction', 'displacement_type', and 'origin_id'.
    """
    df = df.copy()
    df['speed_level'] = df['speed'].apply(lambda x: categorize_speed(x, speed_thresholds))
    df['direction'] = df.apply(lambda row: determine_direction(
        (row['origin_lat'], row['origin_lon']),
        (row['destination_lat'], row['destination_lon'])
    ), axis=1)
    # For low speed, direction is not considered
    df['displacement_type'] = df.apply(
        lambda row: row['speed_level'] + '_' + row['direction'] if row['speed_level'] != 'Low' else 'Low',
        axis=1
    )
    # Compute consistent origin_id
    #df['origin_id'] = df.apply(lambda row: compute_origin_id(row, decimal_places), axis=1)
    return df



"""
INITIALIZE:
"""

def initialize_cluster_counts(num_clusters):
    """
    Initialize counts for each cluster.

    Args:
        num_clusters -> (int): Number of clusters.

    Returns:
        dict: Initialized clusters dictionary.
    """
    clusters = {}
    for k in range(num_clusters):
        clusters[k] = {
            'n_d_o': {},        # Counts of transitions from origin to destination
            'n_tau_o': {},      # Counts of time intervals from origin
            'n_tot_o': {}     # Total counts from each origin
        }
    return clusters


"""

UPDATE:

"""
def update_cluster_counts(clusters, k, user_data, increment=True, bin_size=60):
    """
    Update cluster counts when a user is added or removed from a cluster.

    Args:
        clusters (dict): Clusters dictionary.
        k (int): Cluster ID.
        user_data (pd.DataFrame): DataFrame containing user movement data.
        increment (bool): Whether to add (True) or remove (False) counts.
        bin_size (int): Size of each time bin in seconds.
    """
    factor = 1 if increment else -1
    user_data = user_data.copy()

    # Filter transitions where trip_indicator is 'M' and displacement_type is not 'Undefined'
```

```
    if user_data.empty:
        return  # No valid transitions to update

    # Aggregate counts per origin_id and displacement_type
    displacement_counts = user_data.groupby(['origin_id', 'displacement_type']).size().reset_index(name='count')

    # Aggregate counts per origin_id and time_interval
    #user_data['destination_id'] = user_data['origin_id'].shift(-1)
    user_data['time_interval_binned'] = user_data['time_interval_seconds'].apply(lambda x: bin_time_interval(x, bin_size=bin_size))
    time_interval_counts = user_data.groupby(['origin_id', 'time_interval_binned']).size().reset_index(name='count')
    #Change to using loc


    # Update n_d_o
    for _, row in displacement_counts.iterrows():
        origin_id = row['origin_id']
        #destination_id = row['destination_id']
        displacement_type = row['displacement_type']
        count = row['count']

        if origin_id not in clusters[k]['n_d_o']:
            clusters[k]['n_d_o'][origin_id] = {}
            #clusters[k]['n_ud_o'][origin_id] = {}
        if displacement_type not in clusters[k]['n_d_o'][origin_id]:
            clusters[k]['n_d_o'][origin_id][displacement_type] = 0
        #if destination_id not in clusters[k]['n_d_o'][origin_id]:
         #   clusters[k]['n_ud_o'][origin_id][destination_id] = 0
        clusters[k]['n_d_o'][origin_id][displacement_type] += factor * count
        #clusters[k]['n_ud_o'][origin_id][destination_id] += factor * count

    # Update n_tau_o
    for _, row in time_interval_counts.iterrows():
        origin_id = row['origin_id']
        time_interval = row['time_interval_binned']
        count = row['count']

        if origin_id not in clusters[k]['n_tau_o']:
            clusters[k]['n_tau_o'][origin_id] = {}
        if time_interval not in clusters[k]['n_tau_o'][origin_id]:
            clusters[k]['n_tau_o'][origin_id][time_interval] = 0
        clusters[k]['n_tau_o'][origin_id][time_interval] += factor * count

    # Update n_tot_o
    total_counts = user_data.groupby('origin_id').size().to_dict()
    for origin_id, count in total_counts.items():
        if origin_id not in clusters[k]['n_tot_o']:
            clusters[k]['n_tot_o'][origin_id] = 0
        clusters[k]['n_tot_o'][origin_id] += factor * count




"""
[illegible]:

"""

def clean_clusters(clusters,cluster_assignments,df):
    """
    Remove origin_ids from clusters where counts are zero.

    Args:
        clusters (dict): Clusters dictionary.

    Returns:
        dict: Cleaned clusters dictionary.
    """

    df = df.copy()
    df['clusters'] = df['user_id'].map(cluster_assignments)
    origin_id_cluster = df.groupby('origin_id')['clusters'].apply(list).to_dict()


    for k in clusters:
        # Clean n_d_o, n_tau_o, and n_tot_o
        n_tot_o = clusters[k]['n_tot_o']
        n_tau_o = clusters[k]['n_tau_o']
        n_d_o = clusters[k]['n_d_o']
        origin_ids_to_remove = []
        for origin_id in list(n_d_o.keys()):
            if origin_id not in origin_id_cluster:
                origin_ids_to_remove.append(origin_id)
            displacement_counts = n_d_o[origin_id]
            # Remove displacement_types with zero counts
            displacement_types_to_remove = [dt for dt, count in displacement_counts.items() if count <= 0]
            for dt in displacement_types_to_remove:
                del displacement_counts[dt]
            # If after removing zero-count displacement_types, the displacement_counts dict is empty, mark origin_id for removal
            if not displacement_counts:
                origin_ids_to_remove.append(origin_id)
        # Remove origin_ids with empty displacement_counts
        for origin_id in origin_ids_to_remove:
```

```
            del n_d_o[origin_id]
            del n_tau_o[origin_id]
            del n_tot_o[origin_id]

    return clusters




"""
```



```
"""

def gibbs_sampling_over_time(df, num_clusters, num_iterations, gamma, alpha, beta, speed_thresholds,
                             delta_time_hours=1, decimal_places=4, distance_threshold=100,
                             time_threshold_minutes=1):
    """
    Perform Gibbs sampling over iterations to assign clusters to users.

    Args:
        df (pd.DataFrame): DataFrame containing movement data.
        num_clusters (int): Number of clusters.
        num_iterations (int): Number of Gibbs sampling iterations.
        gamma (float): Smoothing parameter.
        alpha (float): Hyperparameter for prior.
        beta (float): Hyperparameter for prior.
        speed_thresholds (dict): Speed thresholds for categorization.
        delta_time_hours (int): Time interval in hours for clustering.
        decimal_places (int): Decimal places for rounding coordinates.
        distance_threshold (float): Distance in meters to consider still at Origin.
        time_threshold_minutes (float): Maximum time gap in minutes between movements.

    Returns:
        dict: Cluster assignments mapping user_id to cluster_id.
        pd.DataFrame: Processed DataFrame.
        list: List of vocabulary sizes per cluster per user.
        dict: Final cluster counts.
    """
    # Initialize lists to store log-likelihoods and priors (optional for analysis)
    L_list = []
    log_p = []
    log_lik = []
    cluster_sizes_over_iterations = []
    aris = []
    # Define the complete list of displacement types globally
    displacement_list = [
        'High_North-West', 'High_North-East', 'High_South-East', 'High_South-West',
        'Medium_North-West', 'Medium_North-East', 'Medium_South-West', 'Medium_South-East',
        'Low', 'High_West', 'High_East', 'High_North', 'High_South',
        'Medium_West', 'Medium_East', 'Medium_North', 'Medium_South'
    ]

    # Compute displacement types
    df = compute_displacement_type(df, speed_thresholds, decimal_places)

    # Ensure 'start_time' and 'end_time' are in datetime format
    df['start_time'] = pd.to_datetime(df['start_time'], errors='coerce')  # Convert to datetime
    df['end_time'] = pd.to_datetime(df['end_time'], errors='coerce')

    # Drop rows with invalid datetime entries
    df.dropna(subset=['start_time', 'end_time'], inplace=True)

    # Sort data by time
    df = df.sort_values('start_time')
    #df['destination_id'] = df['origin_id'].shift(-1)
    # Define time intervals (e.g., every hour)
    df['time_interval'] = df['start_time'].dt.floor(f'{delta_time_hours}H')  # e.g., floor to nearest hour

    # Compute time interval seconds for each row
    df['time_interval_seconds'] = (df['end_time'] - df['start_time']).dt.total_seconds()  # In seconds


    # Initialize clusters

    clusters = initialize_cluster_counts(num_clusters)


    L_fixed = len(displacement_list)  # Fixed vocabulary size

    # Initialize cluster assignments before iterations
    cluster_assignments = {}  # user_id -> cluster_id
    previous_cluster_assignments = {}  # user_id -> cluster_id at previous time
    k_new = np.random.choice(num_clusters)
    user_ids = df['user_id'].unique()
    for user_id in user_ids:
```

```
        k = np.random.choice(num_clusters)
        cluster_assignments[user_id] = k
        user_data = df[df['user_id'] == user_id]
        if not user_data.empty and (user_data['displacement_type'] != 'Undefined').any():
            update_cluster_counts(clusters, k, user_data, increment=True)
    previous_cluster_assignments = cluster_assignments.copy()
    # Perform Gibbs Sampling iterations
    for iteration in range(num_iterations):
        logger.info(f"Starting Gibbs Sampling Iteration {iteration + 1}/{num_iterations}")

        sorted_users = df.groupby('user_id')['start_time'].min().sort_values().index.tolist()

        for user_id in sorted_users:
            user_data = df[df['user_id'] == user_id]
            user_data = user_data.copy()
            if user_data.empty:
                continue

            k_old = cluster_assignments[user_id]


            if (user_data['displacement_type'] != 'Undefined').any():
                update_cluster_counts(clusters, k_old, user_data, increment=False)

            log_likelihoods = []
            log_priors = []
            for k in range(num_clusters):
                log_likelihood = 0.0


                if not user_data.empty:
                    displacement_counts = user_data.groupby(['origin_id', 'displacement_type']).size().reset_index(name='displacement_count')
                    user_data['time_interval_binned'] = user_data['time_interval_seconds'].apply(lambda x: bin_time_interval(x, bin_size=60))
                    time_interval_counts = user_data.groupby(['origin_id', 'time_interval_binned']).size().reset_index(name='interval_count')
                    displacement_counts = displacement_counts.merge(time_interval_counts, on='origin_id', how='left')
                    displacement_counts = displacement_counts[displacement_counts['time_interval_binned'] == 60] #maybe should return

                    for _, row in displacement_counts.iterrows():
                        origin_id = row['origin_id']
                        displacement_type = row['displacement_type']
                        displacement_count = row['displacement_count']


                        if origin_id not in clusters[k]['n_d_o']:
                            continue


                        cluster_numerator = displacement_count + gamma
                        cluster_denominator = displacement_counts['displacement_count'].sum() + L_fixed * gamma
                        print(f"cluster_numerator: {cluster_numerator}, cluster_denominator: {cluster_denominator}")
                        if cluster_denominator == 0:
                            logger.warning(f"Denominator zero for cluster {k}, origin_id {origin_id}.")
                            prob = 1e-10
                        else:
                            prob = cluster_numerator / cluster_denominator
                        log_lik.append(prob)
                        prob = max(prob, 1e-10)

                        log_likelihood += displacement_count * np.log(prob)

                log_likelihoods.append(log_likelihood)

                # Calculate prior
                n_total_t_minus_u = len(cluster_assignments) - 1
                n_k_t_minus_u = sum(1 for uid, ck in cluster_assignments.items() if ck == k and uid != user_id)

                n_total_t_minus_1 = len(previous_cluster_assignments)
                n_k_t_minus_1 = sum(1 for uid, ck in previous_cluster_assignments.items() if ck == k)

                numerator_prior = n_k_t_minus_u + alpha * n_k_t_minus_1 + beta
                denominator_prior = n_total_t_minus_u + alpha * n_total_t_minus_1 + beta * num_clusters

                prior = numerator_prior / denominator_prior if denominator_prior != 0 else 1.0 / num_clusters
                log_p.append(prior)
                prior = max(prior, 1e-10)
                log_priors.append(np.log(prior))

            # Calculate log-posterior
            log_posterior = np.array(log_likelihoods) + np.array(log_priors)
            log_posterior -= np.max(log_posterior)  # Numerical stability

            posterior_probs = np.exp(log_posterior)
            posterior_probs /= posterior_probs.sum()

            # Sample new cluster
            k_new = np.random.choice(num_clusters, p=posterior_probs)
            cluster_assignments[user_id] = k_new

            # Update cluster counts
            if (user_data['displacement_type'] != 'Undefined').any():
                update_cluster_counts(clusters, k_new, user_data, increment=True)
                for _, row in displacement_counts.iterrows():
                    origin_id = row['origin_id']
```

```
                    displacement_type = row['displacement_type']
                    displacement_count = row['displacement_count']
                    time_interval = row['time_interval_binned']
                    time_interval_count = row['time_interval_binned']
                    clusters[k_new]['n_d_o'][origin_id][displacement_type] = displacement_count
                    clusters[k_new]['n_tot_o'][origin_id] = sum(clusters[k_new]['n_d_o'][origin_id].values())
                    clusters[k_new]['n_tau_o'][origin_id][time_interval] = time_interval_count
        cluster_sizes = [sum(1 for c in cluster_assignments.values() if c == k) for k in range(num_clusters)]
        cluster_sizes_over_iterations.append(cluster_sizes)
        labels_prev = [previous_cluster_assignments[user_id] for user_id in sorted_users]
        labels_curr = [cluster_assignments[user_id] for user_id in sorted_users]
        ari = adjusted_rand_score(labels_prev, labels_curr)
        aris.append(ari)
        previous_cluster_assignments = cluster_assignments.copy()
        logger.info(f"Completed Gibbs Sampling Iteration {iteration + 1}/{num_iterations}")

    return cluster_assignments, df, L_list, clusters, log_lik, log_p, cluster_sizes_over_iterations, aris




def bin_time_interval(seconds, bin_size=60):
    """
    Bin the time interval into buckets.

    Args:
        seconds (int): Time interval in seconds.
        bin_size (int): Size of each bin in seconds.

    Returns:
        int: Binned time interval.
    """
    return (seconds // bin_size) * bin_size


"""
[redacted]:
"""

import pandas as pd
import numpy as np
from datetime import timedelta
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


import numpy as np
import pandas as pd
from datetime import timedelta
import logging

def predict_future_movements_binned(
    df,
    cluster_assignments,
    clusters,
    delta_t_hours=1,
    decimal_places=4,
    area_bounds=None,  # Tuple: (min_lat, max_lat, min_lon, max_lon)
    prediction_start_time=None,
    prediction_end_time=None
):
    """
    Predict future movements for each user based on their cluster transitions with binned time intervals,
    within a specified geographical area and time window.

    Args:
        df (pd.DataFrame): DataFrame containing movement data.
        cluster_assignments (dict): Cluster assignments mapping user_id to cluster_id.
        clusters (dict): Transition counts per cluster.
        delta_t_hours (int): Time window for predictions into the future.
        decimal_places (int): Decimal places for rounding coordinates.
        area_bounds (tuple, optional): Geographical boundaries as (min_lat, max_lat, min_lon, max_lon).
        prediction_start_time (pd.Timestamp, optional): Start time for predictions.
        prediction_end_time (pd.Timestamp, optional): End time for predictions.

    Returns:
        pd.DataFrame: Predicted movements with columns ['user_id', 'predicted_time', 'predicted_lat', 'predicted_lon'].
    """

    predicted_movements = []

    # Ensure 'start_time' and 'end_time' are datetime
    df['start_time'] = pd.to_datetime(df['start_time'], errors='coerce')
```

```python
    df['end_time'] = pd.to_datetime(df['end_time'], errors='coerce')

    # Compute 'destination_id' by shifting 'origin_id' within each user
    df['destination_id'] = df.groupby('user_id')['origin_id'].shift(-1)

    # Drop rows with invalid datetime entries or missing 'destination_id'
    df = df.dropna(subset=['start_time', 'end_time', 'destination_id']).copy()



    # Apply area filtering if bounds are provided
    if area_bounds is not None:
        min_lat, max_lat, min_lon, max_lon = area_bounds
        df = df[
            (df['origin_lat'] >= min_lat) & (df['origin_lat'] <= max_lat) &
            (df['origin_lon'] >= min_lon) & (df['origin_lon'] <= max_lon) &
            (df['destination_lat'] >= min_lat) & (df['destination_lat'] <= max_lat) &
            (df['destination_lon'] >= min_lon) & (df['destination_lon'] <= max_lon)
        ]
        logger.info(f"Area filtered data: {len(df)} rows")

    # Apply time window filtering if provided
    if prediction_start_time is not None:
        df = df[df['end_time'] >= prediction_start_time]
    if prediction_end_time is not None:
        df = df[df['end_time'] <= prediction_end_time]
    logger.info(f"Time filtered data: {len(df)} rows")

    df['cluster_id'] = df['user_id'].map(cluster_assignments)
    # Build mapping from origin_id to (lat, lon)
    origin_id_to_location = df.groupby('origin_id').agg({'origin_lat': 'mean', 'origin_lon': 'mean'})

    # Set the prediction time window
    delta_t = timedelta(hours=delta_t_hours)

    for user_id in df['user_id'].unique():
        user_cluster = cluster_assignments.get(user_id)
        if user_cluster is None:
            continue  # Skip if user not in cluster assignments

        # Get user's data and last location within the filtered data
        user_data = df[df['user_id'] == user_id].sort_values('start_time')
        logger.info(f"Predicting movements for user {user_id} in cluster {user_cluster} with {len(user_data)} movements.")
        if user_data.empty:
            continue

        last_movement = user_data.iloc[-1]
        hat_O = last_movement['origin_id']
        hat_tau = last_movement['end_time']

        # Set the end time for predictions
        t_end = hat_tau + delta_t

        while hat_tau < t_end:
            cluster = clusters.get(user_cluster, {})
            df_cluster = df[df['cluster_id'] == user_cluster]
            if not cluster:
                logger.warning(f"No data for cluster {user_cluster}")
                break  # No data for this cluster

            # Get transition counts from current origin
            n_d_o = df_cluster[df_cluster['trip_indicator'] == 'D']['origin_id'].value_counts()
            n_tau_o = cluster['n_tau_o'].get(hat_O, {})
            n_tot_o = cluster['n_tot_o'].get(hat_O, 0)

            if n_tot_o == 0  or not n_tau_o:
                # Cannot proceed further
                break

            # Compute probabilities for destinations
            dest_ids = n_d_o.keys() #list(n_d_o.keys())
            dest_counts = n_d_o.values #list(n_d_o.values())
            dest_probs = dest_counts / dest_counts.sum()

            if len(dest_ids) == 0 or len(dest_probs) == 0:
                logger.warning(f"No destination probabilities for cluster {user_cluster}, origin {hat_O}")
                break
            # Draw next location
            hat_D = np.random.choice(dest_ids, p=dest_probs)

            # Compute probabilities for time intervals
            tau_values = list(n_tau_o.keys())
            tau_counts = np.array([n_tau_o[tau] for tau in tau_values])
            tau_probs = tau_counts / tau_counts.sum()

            # Draw next time interval
            delta_hat_tau = np.random.choice(tau_values, p=tau_probs)
            delta_hat_tau = int(delta_hat_tau)  # Ensure integer

            # Update time
            hat_tau += timedelta(seconds=delta_hat_tau)

            # Update origin
            hat_O = hat_D
```

```
            # Get location of new origin_id
            if hat_O not in origin_id_to_location.index:
                logger.warning(f"Cannot find location for origin {hat_O}")
                # Cannot find location, stop prediction
                break

            location = origin_id_to_location.loc[hat_O]
            predicted_lat = location['origin_lat']
            predicted_lon = location['origin_lon']

            # Record predicted movement
            predicted_movements.append({
                'user_id': user_id,
                'predicted_time': hat_tau,
                'predicted_lat': predicted_lat,
                'predicted_lon': predicted_lon
            })

    # Convert predictions to DataFrame
    predicted_df = pd.DataFrame(predicted_movements)
    return predicted_df
```

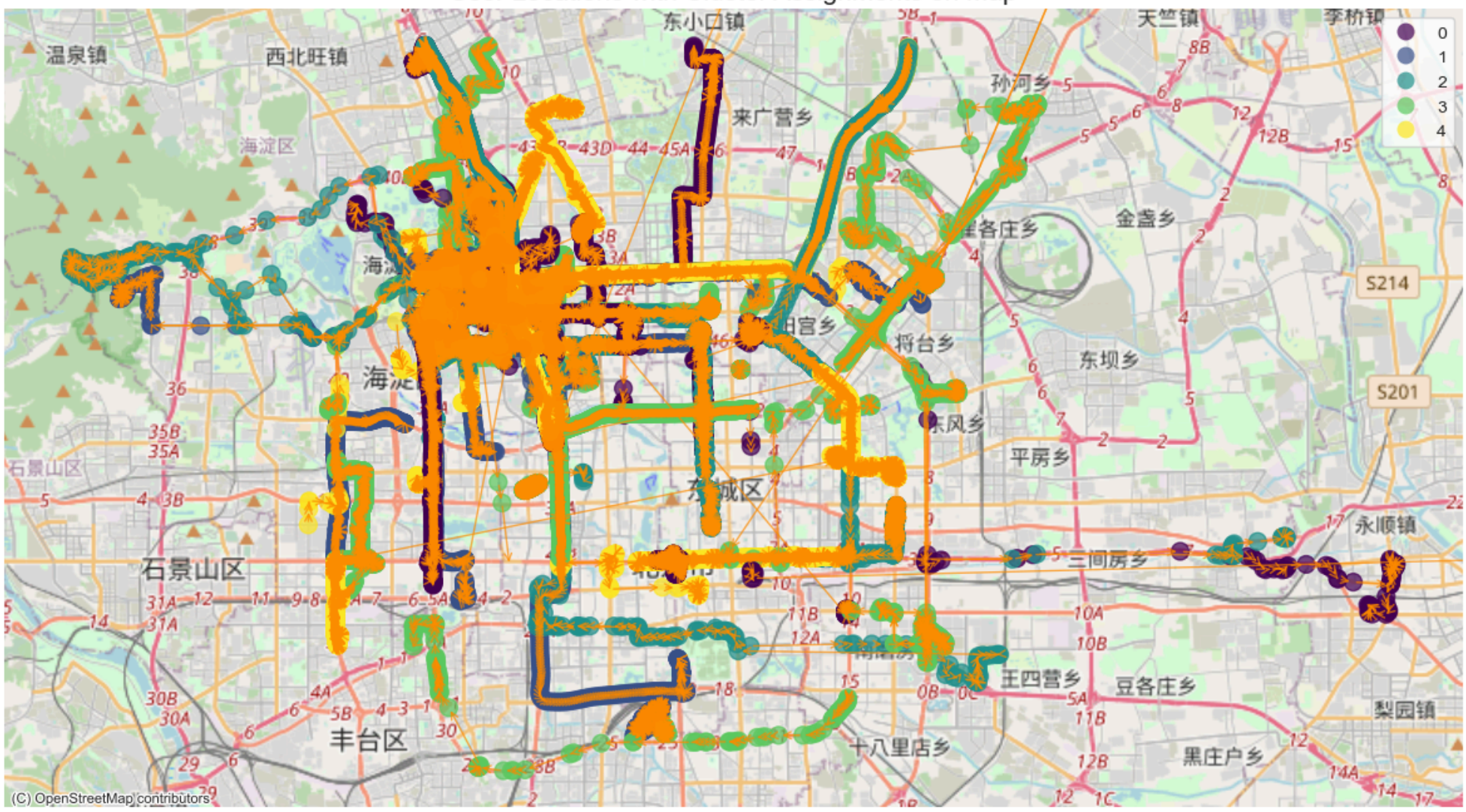
Running clustering phase:

In [ ]:
```
"""
# Example parameters
speed_thresholds = {'low': 10.8, 'medium': 43.2}
num_clusters = 5
num_iterations = 1000
gamma = 1
alpha = 0.01
beta = 1000
delta_time_hours = 1  # Time interval for Gibbs sampling
prediction_delta_hours = 36  # Prediction horizon



# Perform Gibbs sampling over time
cluster_assignments, processd_df,L_list, clusters2,log_lik, log_p, clusters_over_time, aris = gibbs_sampling_over_time(
    locations_df_new2_assigned_subset,#locations_df_new2[:10000],
    num_clusters=num_clusters,
    num_iterations=num_iterations,
    gamma=gamma,
    alpha=alpha,
    beta=beta,
    speed_thresholds=speed_thresholds,
    delta_time_hours=delta_time_hours,
    decimal_places=4
)
"""
```

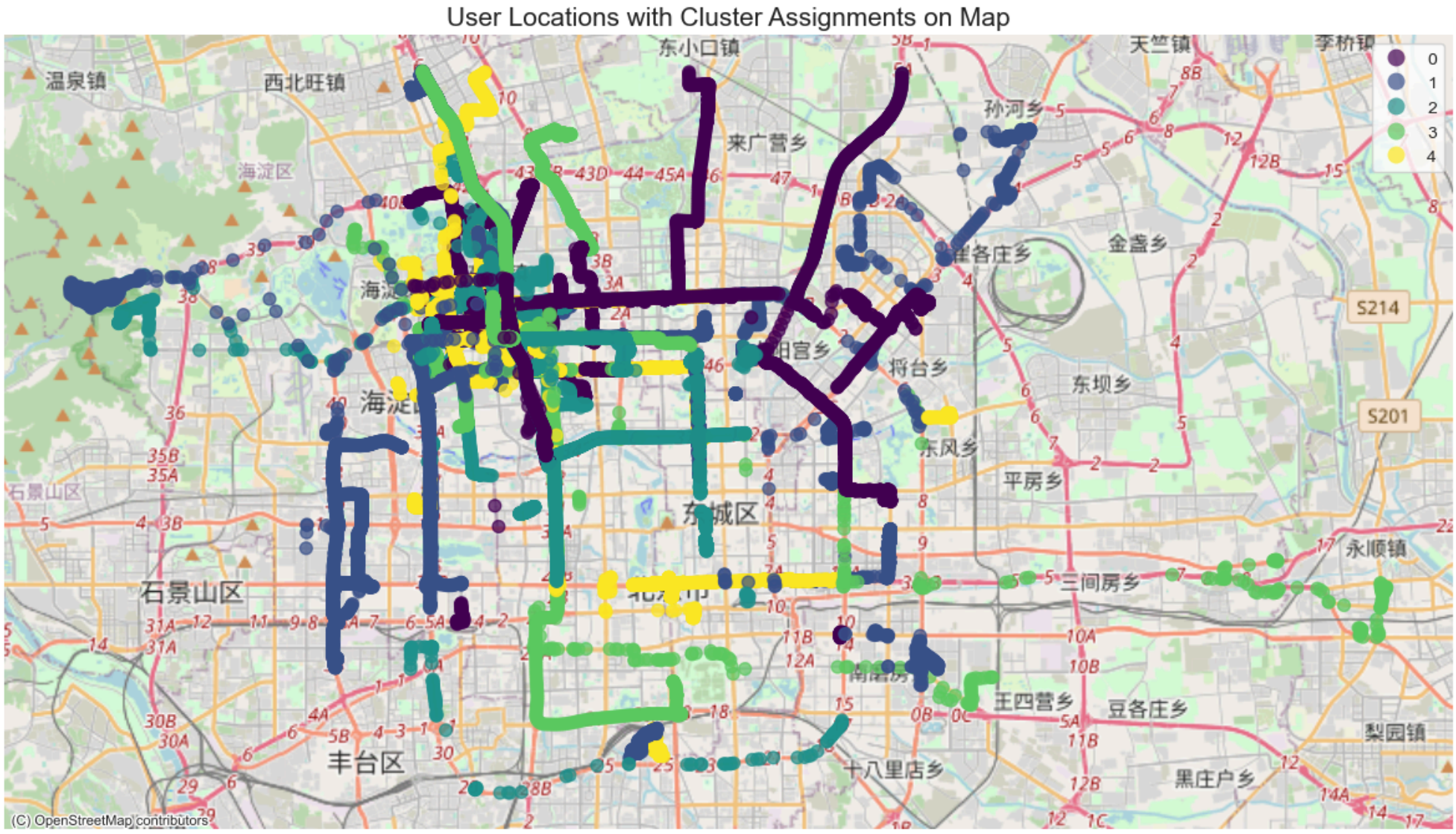
Running prediction phase:

In [ ]:
```
#clusters2_cleaned_new= clean_clusters(clusters2,cluster_assignments,processd_df)
```

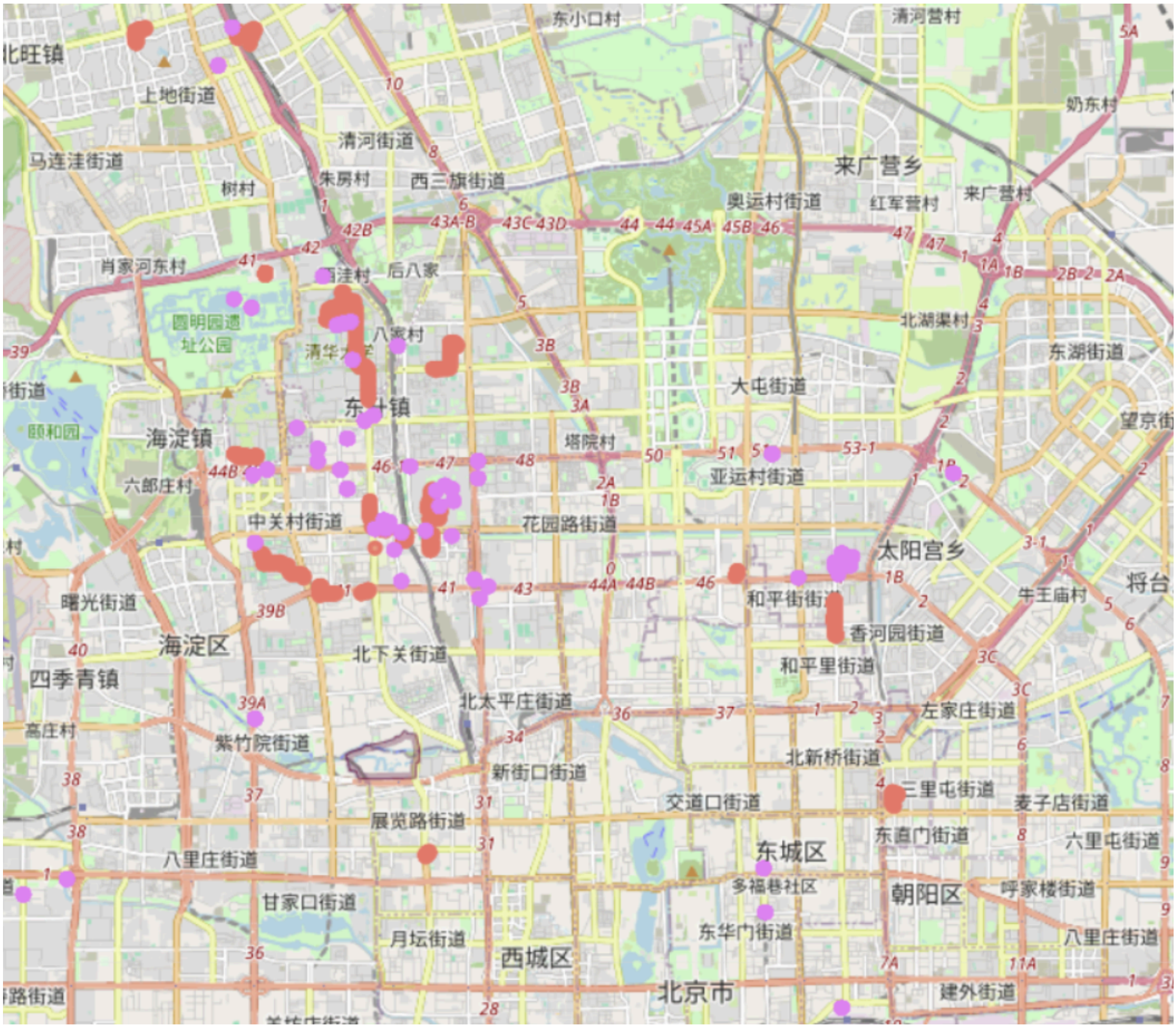
In [ ]:
```
"""
# Define geographical boundaries and time window
area_bounds = (39.0, 41.0, 115.0, 117.0)  # (min_lat, max_lat, min_lon, max_lon)
prediction_start_time = pd.to_datetime('2008-01-01 00:00:00')
prediction_end_time = pd.to_datetime('2008-12-29 19:00:00')

# Call the prediction function with constraints
predicted_df = predict_future_movements_binned(
    processd_df,
    cluster_assignments,
    clusters2_cleaned_new,
    delta_t_hours=1,
    decimal_places=4,
    area_bounds=area_bounds,
    prediction_start_time=prediction_start_time,
    prediction_end_time=prediction_end_time
)
"""
```



Plot 1: Clusters assignment after Gibbs sampling with arrows, demonstrating displacements. 5 clusters, 3 iterations, alpha = 0.01, gamma = 1, beta = 1000



Plot 2: Clusters assignment after Gibbs sampling. 5 clusters, 100 iterations, alpha = 0.01, gamma = 1, beta = 1000



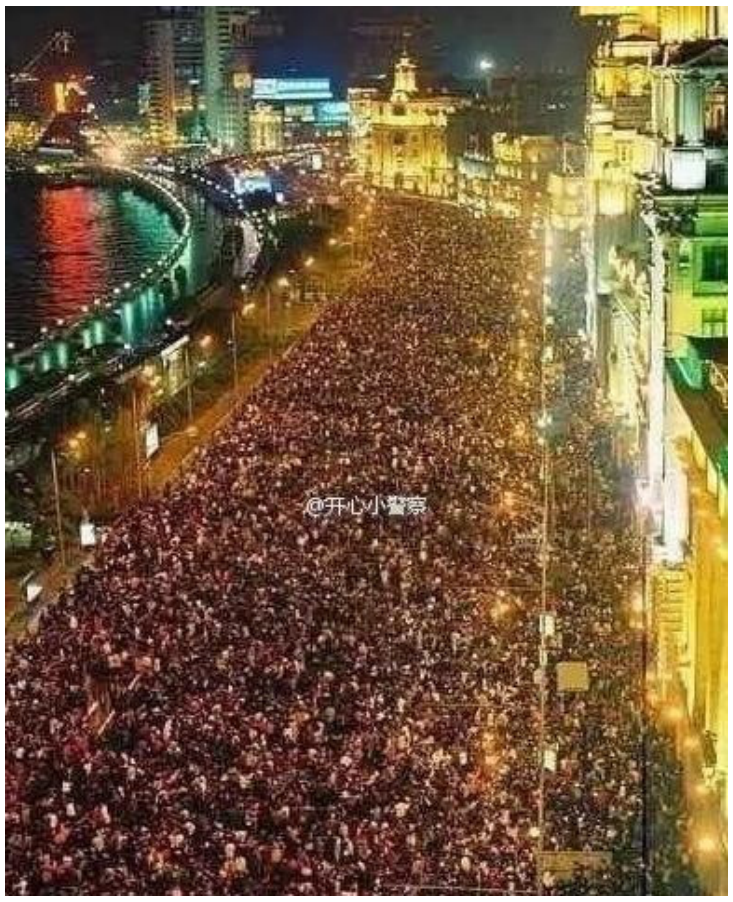
Plot 3: Plotting predicted points for future destinations in year 2008 (pink) vs real destinations in 2008 (red)

Plot 4: Plotting predicted points for future destinations in year 23.10.2008 until 14:10 (pink) vs real destinations in 23.10.2008 until 14:10 (red)

Convergence of gibbs sampling Cluster Assignments series:

Top to bottom: number of users in clusters for iterations 300-400, iterations 0-100

It seems we have a tendency for convergence (from the top plot) but it's not that stable.



Shanghai, New Year's Eve, 2014