Leveraging Twitter Data for Cellular Network Planning

Shubham Tiwari ⟨*shubham.tiwari@partner.samsung.com*⟩

ABSTRACT

This report explores the possibility of using streamed Twitter data to estimate the amount of cellular network resources that must be allocated at an area where the density of cellular network users has unexpectedly increased due to events such as protests or natural disasters.

1. Introduction

Social media platforms allow people to share their thoughts and feelings about a particular event in real time. Twitter, a micro-blogging website, is one such platform. Twitter allows users to optionally provide their geographical location along with the tweet (called Geo-tagged tweets). Twitter also provides an API to stream the tweet data in real time. By streaming geo-tagged tweets, it maybe possible to estimate the crowds in a location in real-time, which can further be used to estimate the amount of cellular network resources that must be provisioned in order to support the QoS demands of the users in the crowded areas. This report discusses the initial work done in these lines.

2. Related Work

Authors in [1] propose a Machine Learning based network planning tool. Data is collected from user-equipment (UE), which may be located at any point in the network, and QoS is estimated based on data through Supervised ML algorithms (SVM, k-NN, NN, DT). The proposed tool is evaluated for two use-cases: (1) small cell deployment in a dense indoor network (2) timely face a detected fault in a macro-cell network. Authors in [2] show that Twitter data can be used to reliably estimate spatial traffic density distribution, and propose a method based on `Voronoi Estimation` [3] to estimate the spatial density, and a queueing theoretic model [4] to estimate the Key Performance Indicators (KPIs). Authors in [5] leverage LDA for topic modelling of Tweets and hierarchical clustering (unsupervised ML), `text2vec` (supervised ML) for predicting the polarity (sentiment) of tweets on the timeline of Mobile Network Operators (MNOs). Article [6] presents a survey of event detection techniques using Twitter data stream. Events are classified into two types: Unspecified events and specified events. In case of detecting unspecified events, no prior information about the event is available. Therefore detection of events in this case requires detecting similarity in trends and grouping them together. In case of specified event detection, information/features about the event are provided by the user. The type of event detection can also be divided into two types: `RED` (Retrospective Event Detection) and `NED` (New Event Detection). `RED` involves detecting previously unidentified events using historical tweet data. `NED` involves detection of new events using Twitter stream in real time. Our use-case can be classified as a `specified NED`. Since we know the bounds of the geographical region that we are interested in, the event is `specified`, and since we are detecting the event in real time using Twitter stream, the type of event detection is `NED`. The survey describes a process of a detecting local geo-social events, by modelling regularities in crowd behavior estimated over a period of time. The geographical area is divided into Regions of Interest (ROIs) using k-means clustering. Irregularities in crowd behavior are detected by comparing the real time Tweet data stream with the estimated behavior.

3. Current Work

Our work focuses on developing an ML based network planning tool taking inspiration from [1], leveraging Twitter data for estimating crowded areas and provisioning cellular resources in those areas. Historical twitter data for the geographical area of interest is used to estimate the crowd behavior in the region. Using density based clustering, the crowd density can be estimated in a specific region for a regular day. Thresholds for a crowd being an outlier is then calculated. If the number of people in the region is above the threshold value, then the cluster is considered to be an irregular crowd.

4. Streaming Twitter data

`Twitter API` allows streaming of tweets in real time, searching recent tweets, and downloading batch historical tweets. Streaming of tweets in real time can be done with a developer account. However, the API allows allows only a sample of the tweets to be streamed. Actual percentage of the tweets streamed depends on the users request and amount of traffic. Streaming almost 100% of the tweets requires subscribing to `Twitter Firehose`, which is an enterprise feature. The huge quantity of streamed twitter data obtained through `Twitter Firehose` also requires the end user to have sufficient resources to process and store the data. By searching through the `Twitter API`, some of the most recent tweets matching the search criteria can be obtained. However, that may not be sufficient for research purposes. Downloading of batch historical tweets is also an enterprise feature.

Numerous streamed Twitter datasets are available online. For the current work, we use a 12 day stream of geo-located tweets within Tokyo [7].

5. Methodology

We follow the density based clustering described in [8]. A circular area with `Tokyo Station` ($x_0 = (35.681308, 139.767127)$) as the center and radius $d = 69752.35$ meters is chosen as the area under consideration for detecting crowds. For each hour, ε is estimated by Eq. 1:

$$\varepsilon = \frac{1}{n}\sum_{i=1}^{n} dist_i \tag{1}$$

where n is the number of geo-tagged tweets in the hour and $dist_i$ is the minimum distance of point x_i from all other points in that hour. Using ε estimated from Eq. 1, $MinPts$ is estimated is estimated by Eq. 2:

$$MinPts = \frac{1}{n}\sum_{i=1}^{n} NumPts_i \tag{2}$$

where $NumPts_i$ is the number of points inside a radius of ε from each point x_i. The estimated ε and $MinPts$ are then chosen as arguments to the `DBSCAN` clustering algorithm.

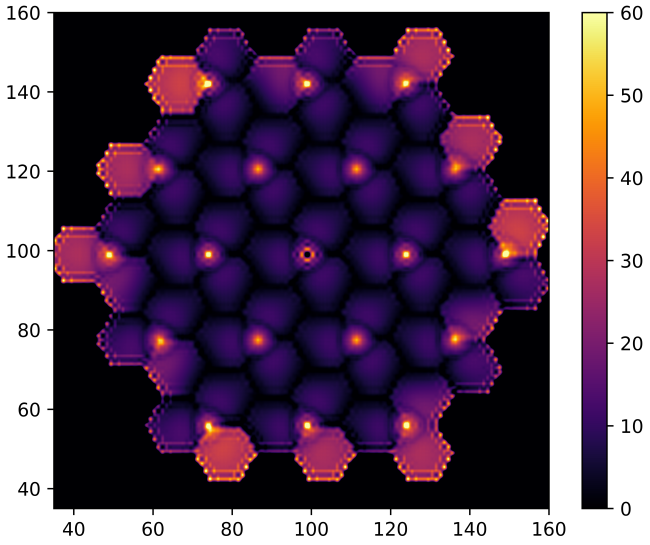
The twitter dataset in [7], is supposed to consist of 200,000 geo-tagged tweets for 10 days. However a lot of the tweets lack geographical data. Table 1 shows the number of actual geo-tagged tweets on each day. Days `2016-04-16` and `2016-04-17` clearly have a larger number of tweets as compared to the other days. This indicates that those days were of special significance in Tokyo, and it is expected that there would be outlier crowds on these days. Hence, days `2016-04-16` and `2016-04-17` consist of the test set. Day `2016-04-12` has a very low number of tweets due to very short period of streaming of tweets, and is therefore dropped from the dataset. Rest of the days consist of the training set.

Day	# Tweets
2016-04-21	252
2016-04-20	299
2016-04-19	353
2016-04-18	346
2016-04-17	456
2016-04-16	513
2016-04-15	293
2016-04-14	370
2016-04-13	359
2016-04-12	26

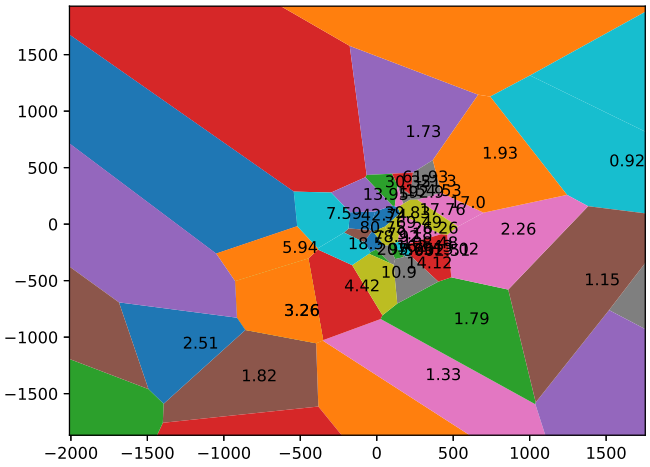
Table 1. # Tweets on each day

6. Results

The results of `DBSCAN` clustering are shown in Figure 3. (a) - (x) and Figure 4. The scripts used to produce the results can be found in Section 8.



(a) SINR distribution for the urban macro cell setup



(b) User density obtained through voronoi tessellation

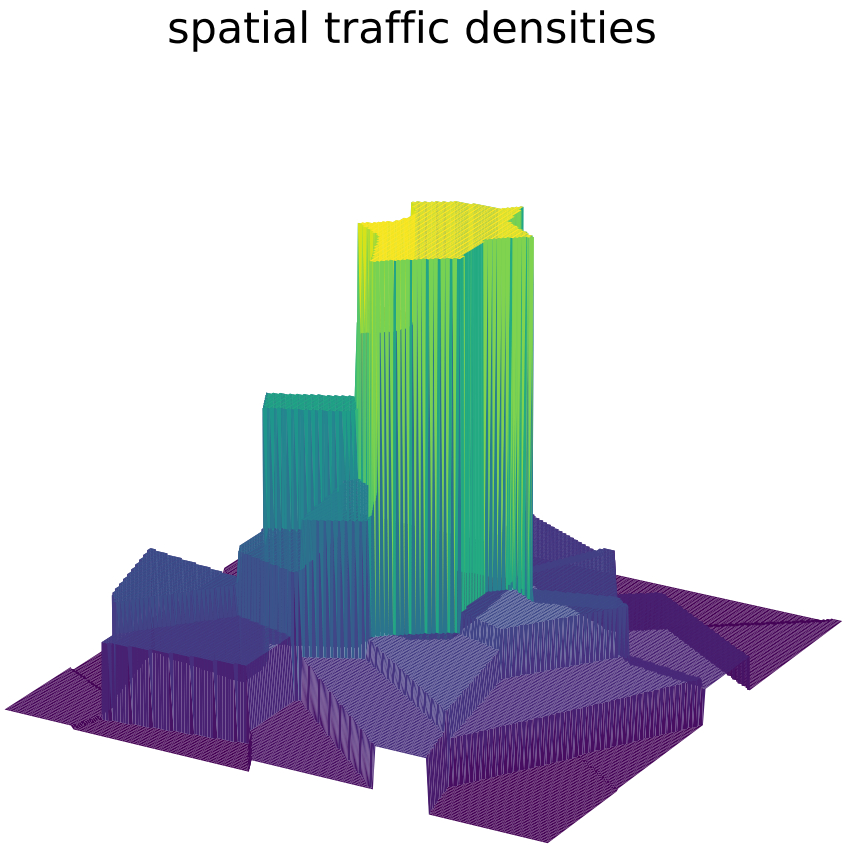


Figure 2. User density distribution in the region

7. Future Work

Future work will involve using the test dataset to evaluate the the density based crowd detection using `DBSCAN`. Allocation of cellular network resources in crowded area will be evaluated through simulations.

References

[1] J. Moysen, L. Giupponi, and J. Mangues-Bafalluy, "A mobile network planning tool based on data analytics," *Mobile Information Systems*, vol. 2017, 2017.

[2] H. Klessig, H. Kuntzschmann, L. Scheuvens, B. Almeroth, P. Schulz, and G. Fettweis, "Twitter as a source for spatial traffic information in big data-enabled self-organizing networks," in *2017 IEEE Wireless Communications and Networking Conference (WCNC)*, 2017, pp. 1–5.

[3] C. D. Barr and F. P. Schoenberg, "On the voronoi estimator for the intensity of an inhomogeneous planar poisson process," *Biometrika*, vol. 97, no. 4, pp. 977–984, 2010.

[4] H. Klessig, D. Öhmann, A. J. Fehske, and G. P. Fettweis, "A performance evaluation framework for interference-coupled cellular data networks," *IEEE Transactions on Wireless Communications*, vol. 15, no. 2, pp. 938–950, 2016.

[5] K. A. Ogudo and D. M. J. Nestor, "Sentiment analysis application and natural language processing for mobile network operators' support on social media," in *2019 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*, 2019, pp. 1–10.

[6] F. Atefeh and W. Khreich, "A survey of techniques for event detection in twitter," *Comput. Intell.*, vol. 31, no. 1, p. 132–164, Feb. 2015. [Online]. Available: https://doi.org/10.1111/coin.12017

[7] "200,000 tokyo geolocated tweets. free twitter dataset," http://www.followthehashtag.com/datasets/200000-tokyo-geolocated-tweets-free-twitter-dataset/, accessed: 2020-06-12.

[8] M. B. Khalifa, R. P. Díaz Redondo, A. F. Vilas, and S. S. Rodríguez, "Identifying urban crowds using geo-located social media data: A twitter experiment in new york city," *J. Intell. Inf. Syst.*, vol. 48, no. 2, p. 287–308, Apr. 2017. [Online]. Available: https://doi.org/10.1007/s10844-016-0411-x

8. Scripts

8.1. Twitter Stream

```
import tweepy
import pprint

pp = pprint.PrettyPrinter(indent=4)

consumer_key = "###"
consumer_secret = "###"

access_token = "###"
access_token_secret = "###"

#override tweepy.StreamListener to add logic to on_status
class MyStreamListener(tweepy.StreamListener):
        def on_status(self, tweet):
                if not ((tweet._json['place'] is None) and (tweet._json['geo'] is None) and (
                        pp.pprint(tweet._json)
                        print(f"{tweet.user.name} -> {tweet.text}")


#auth = tweepy.AppAuthHandler(consumer_key, consumer_secret)
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)

api = tweepy.API(auth, wait_on_rate_limit=True,
    wait_on_rate_limit_notify=True)
```

```
myStreamListener = MyStreamListener()
myStream = tweepy.Stream(auth = api.auth, listener=myStreamListener)

myStream.filter(track=['COVID19'])
```

8.2. Clustering using DBSCAN

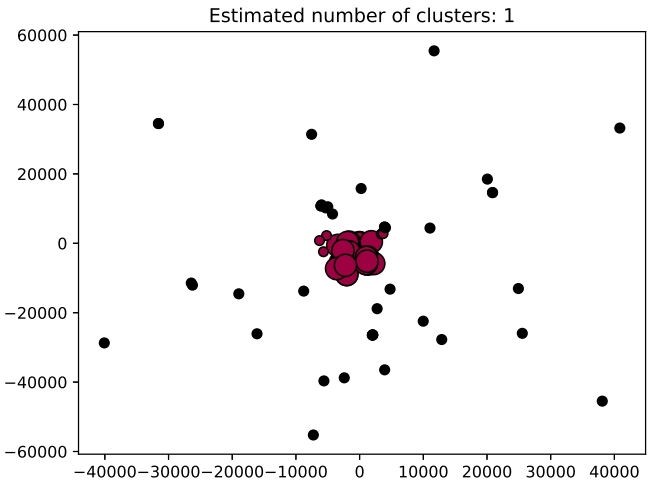
```
from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.metrics.pairwise import pairwise_distances

import pandas as pd
import numpy as np
import sys
import os
from datetime import datetime, timedelta
import geopy.distance
import matplotlib.pyplot as plt

from joblib import dump, load
import pickle

data = pd.read_csv('tokyo.csv', sep=',',encoding = 'utf8',lineterminator='\n',low_memory=Fals
data = data.dropna()

# Maximum number of tweets are on 2016-04-16 and 2016-04-17, so drop all rows related to that
# They will act as the test data
data = data.drop(data[data['Date']=='2016-04-16'].index)
data = data.drop(data[data['Date']=='2016-04-17'].index)
# Very low number of tweets on 2016-04-12, so drop it
data = data.drop(data[data['Date']=='2016-04-12'].index)
#-------------------------------------------------------------------------------#

# get date time object
#data['time'] = datetime.strptime(data['Date']+' '+data['Hour'], '%y-%m-%d %H:%M')
data['time'] =  (data['Hour']).apply(lambda x: datetime.strptime(x, '%H:%M'))
data = data.set_index('time')
print('Columns in dataset:')
print(list(data.columns.values))

# printing dataframe rows where date is equal to date we want
max = 0
date_m = None
print('Numbers of tweets in each day:')
for date in data['Date'].unique():
        n = len(data[data['Date']==date])
        print('{} on {}'.format(n, date))
        if n > max:
                max = n
                date_m = date

print('Max number of tweets in a day: {} on {}'.format(max, date_m))
#print(data['Date'][data['Date']=='2016-04-12'])
```

```
#print('Times in a day: ')
#print(data['Hour'][data['Date']=='2016-04-16'].unique())

# Tokyo station: 35.681308, 139.767127
x = 35.681308
y = 139.767127
origin = (x,y)
print('Tokyo station: x={}, y={}'.format(x,y))

#coords_1 = (52.2296756, 21.0122287)
#coords_2 = (52.406374, 16.9251681)

#print(geopy.distance.vincenty(coords_1, coords_2).m)

# find the maximum distance d in the dataset
d = 0
for index, row in data.iterrows():
        coord = (row['Latitude'], row['Longitude'])
        curr = geopy.distance.vincenty(origin, coord).m
        if curr > d:
                d = curr

print('Maximum distance of a geo-tagged tweet from the origin is: {} meters'.format(d))

#quit()

def calc_min(origin, hdf):
        min = float('inf')
        for index, row in hdf.iterrows():
                if origin[0]==row['Latitude'] and origin[1]==row['Longitude']:
                        continue
                coord = (row['Latitude'], row['Longitude'])
                dist = geopy.distance.vincenty(origin, coord).m
                if dist < min:
                        min = dist
        return min

def calc_min_pts(origin, hdf, eps):
        sum = 0
        for index, row in hdf.iterrows():
                if origin[0]==row['Latitude'] and origin[1]==row['Longitude']:
                        continue
                coord = (row['Latitude'], row['Longitude'])
                dist = geopy.distance.vincenty(origin, coord).m
                if dist < eps:
                        sum = sum + 1
        return sum

def distance_in_meters(x, y):
    return geopy.distance.vincenty((x[0], x[1]), (y[0], y[1])).m

df = {}
eps_dict = {}
min_pts_dict = {}
cluster = {}
```

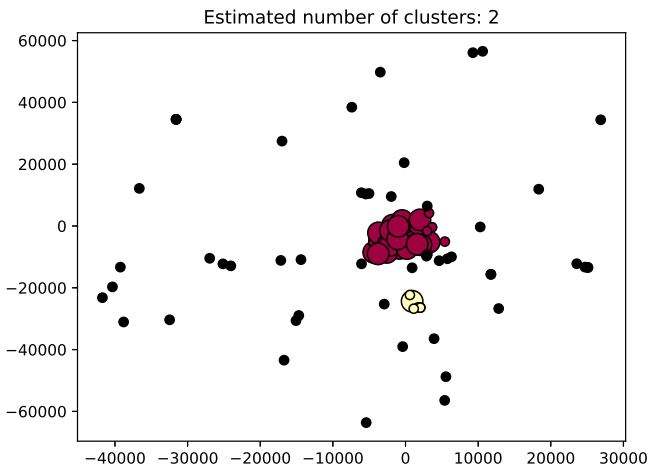
```
n_clusters = pd.DataFrame(columns=['time', 'n_clusters'])
# sort by dates
#for date in data['Date'].unique():
#        df[date] = data[data['Date']==date]

data = data.groupby(pd.Grouper(freq='60Min'))
for key, item in data:
    #print(key, "\n\n") # data.get_group(key) to get the group values
    hdf = data.get_group(key)
    sum = 0
    for index, r in hdf.iterrows():
        sum = sum + calc_min((r['Latitude'], r['Longitude']), hdf)
    eps = sum/len(hdf.index)
    #print('key: {}, eps={}'.format(key, eps))

    pts = 0
    for index, r in hdf.iterrows():
        pts = pts + calc_min_pts((r['Latitude'], r['Longitude']), hdf, eps)
    min_pts = pts/len(hdf.index)
    print('key: {}, eps={} min_pts={}'.format(key, eps, min_pts))
    eps_dict[key] = eps
    min_pts_dict[key] = min_pts

    # Apply fit transform for the current hour
    X = hdf[['Latitude','Longitude']]
    distance_matrix = pairwise_distances(X, metric=distance_in_meters)
    #print(distance_matrix)
    db = DBSCAN(eps=eps, min_samples=int(min_pts), metric='precomputed')
    db.fit(distance_matrix)
    cluster[key] = db
    # save the trained model on the disk for future use
    dump(db, 'clusters/trains/train' + key.strftime("%H") + '.joblib')
    X.to_pickle('clusters/data/data' + key.strftime("%H") + '.pkl')
    ### Output ###
    core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
    core_samples_mask[db.core_sample_indices_] = True
    labels = db.labels_

    # Number of clusters in labels, ignoring noise if present.
    n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
    n_noise_ = list(labels).count(-1)

    n_clusters = n_clusters.append(pd.DataFrame([[key.strftime("%H:%M"), n_clusters_]], colum

    print('Estimated number of clusters: %d' % n_clusters_)
    print('Estimated number of noise points: %d' % n_noise_)
#     print("Homogeneity: %0.3f" % metrics.homogeneity_score(labels_true, labels))
#     print("Completeness: %0.3f" % metrics.completeness_score(labels_true, labels))
#     print("V-measure: %0.3f" % metrics.v_measure_score(labels_true, labels))
#     print("Adjusted Rand Index: %0.3f"
#           % metrics.adjusted_rand_score(labels_true, labels))
#     print("Adjusted Mutual Information: %0.3f"
#           % metrics.adjusted_mutual_info_score(labels_true, labels))
    print("Silhouette Coefficient: %0.3f"
          % metrics.silhouette_score(X, labels))
```

```
    # ##############################################################################
    # Plot result
     #converting points with reference to x and y
    X = X.astype(float)
    #print('-- after conversion --')
    #print(X['Latitude'])
    X['Latitude'] = X['Latitude'].apply(lambda t: np.sign(t - float(x))*(geopy.distance.vince
    X['Longitude'] = X['Longitude'].apply(lambda t: np.sign(t - float(y))*(geopy.distance.vin
    #X['Latitude'] = X['Latitude'] - x
    #X['Longitude'] = X['Longitude'] - y

    # Black removed and is used for noise instead.
    unique_labels = set(labels)
    colors = [plt.cm.Spectral(each)
              for each in np.linspace(0, 1, len(unique_labels))]
    for k, col in zip(unique_labels, colors):
        if k == -1:
            # Black used for noise.
            col = [0, 0, 0, 1]

        class_member_mask = (labels == k)

        xy = X[class_member_mask & core_samples_mask]
        plt.plot(xy.iloc[:, 0].values, xy.iloc[:, 1].values, 'o', markerfacecolor=tuple(col),
                 markeredgecolor='k', markersize=14)

        xy = X[class_member_mask & ~core_samples_mask]
        plt.plot(xy.iloc[:, 0].values, xy.iloc[:, 1].values, 'o', markerfacecolor=tuple(col),
                 markeredgecolor='k', markersize=6)

    plt.title('Estimated number of clusters: %d' % n_clusters_)
#     plt.savefig('clusters/plot_'+ key.strftime("%H-%M") +'.png', bbox_inches='tight')
    loc = 'clusters/png/plot_'+ key.strftime("%H-%M") +'.png'
    plt.savefig(loc, dpi=1000, bbox_inches='tight')

    loc = 'clusters/eps/plot_'+ key.strftime("%H-%M") +'.eps'
    plt.savefig(loc, format='eps', dpi=1000, bbox_inches='tight')

    # Clear the plot
    plt.clf()
    plt.cla()
    plt.close()


# save the cluster data
n_clusters.to_pickle('clusters/nclusterdata.pkl')
# Plot number of clusters as a function of the hour of the day
plt.plot(n_clusters['time'], n_clusters['n_clusters'])
plt.title('Number of clusters in each time of day')
plt.xlabel('hour of the day')
plt.ylabel('number of clusters')
plt.xticks(rotation=45)

#pickle.dump(ax, open("plot.pickle", "wb"))
```

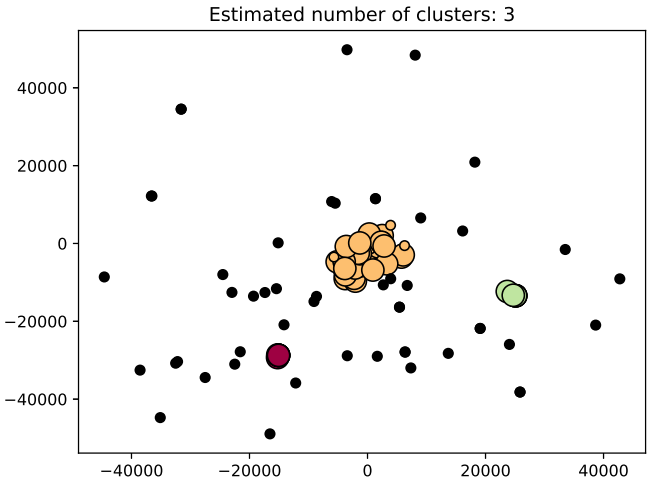
```
loc = 'clusters/nclusters.png'
plt.savefig(loc, dpi=1000, bbox_inches='tight')

loc = 'clusters/nclusters.eps'
plt.savefig(loc, format='eps', dpi=1000, bbox_inches='tight')

# Clear the plot
plt.clf()
plt.cla()
plt.close()
```



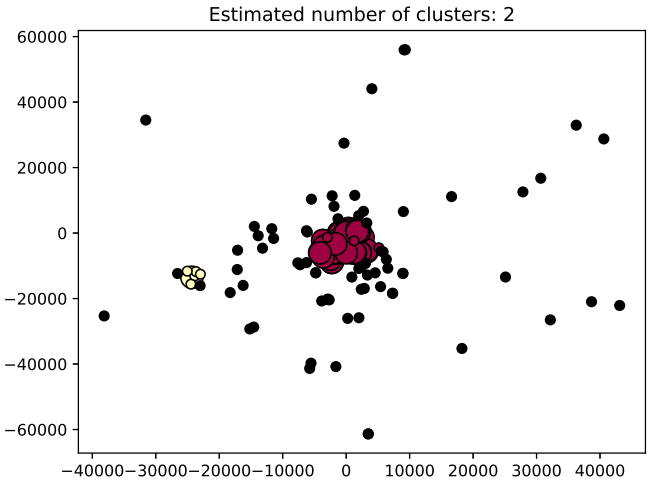
(a) Hour 00:00



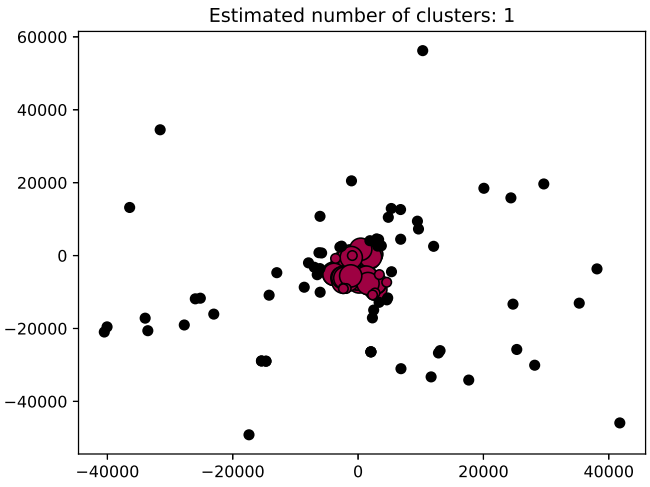
(b) Hour 01:00



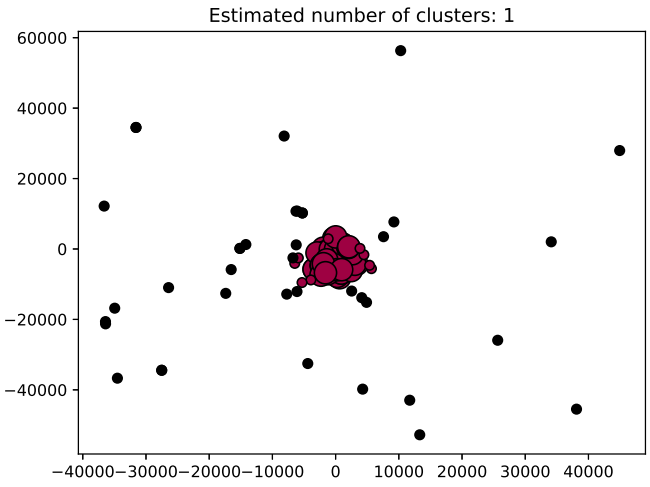
(c) Hour 02:00



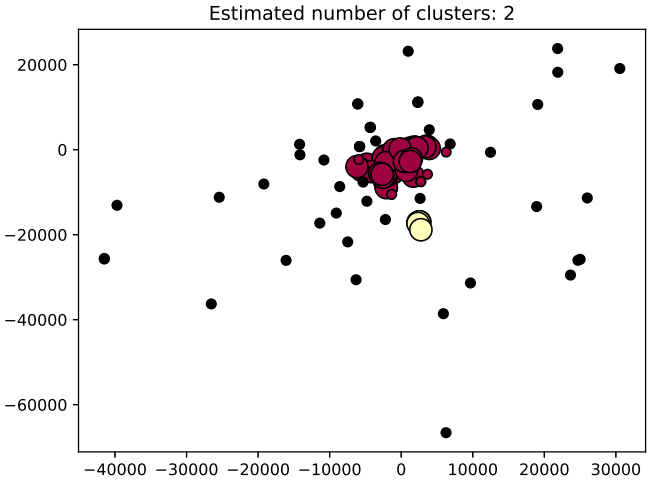
(d) Hour 03:00



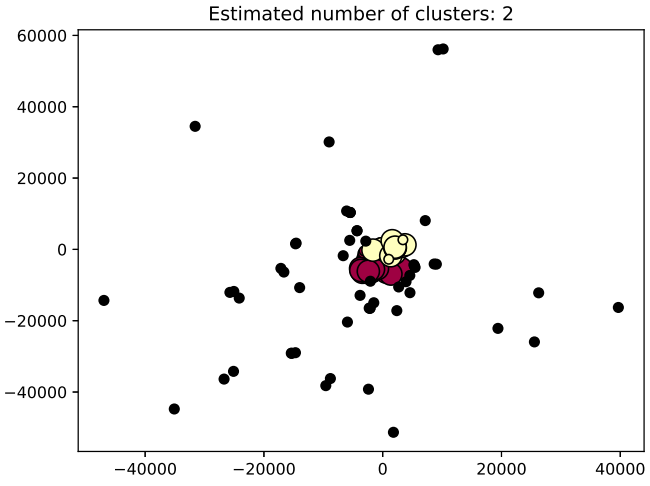
(e) Hour 04:00



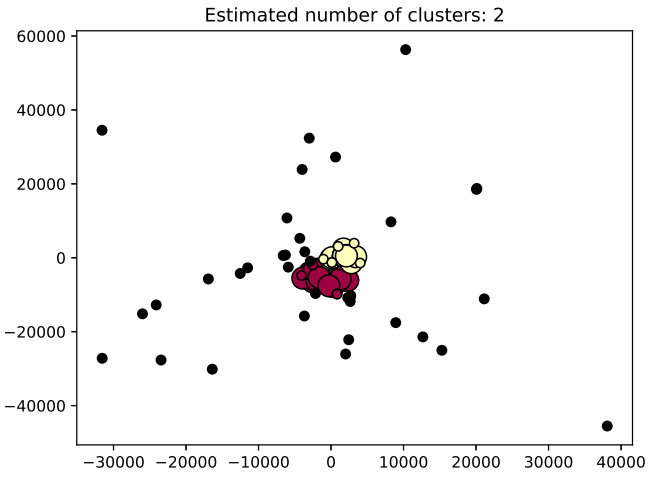
(f) Hour 05:00



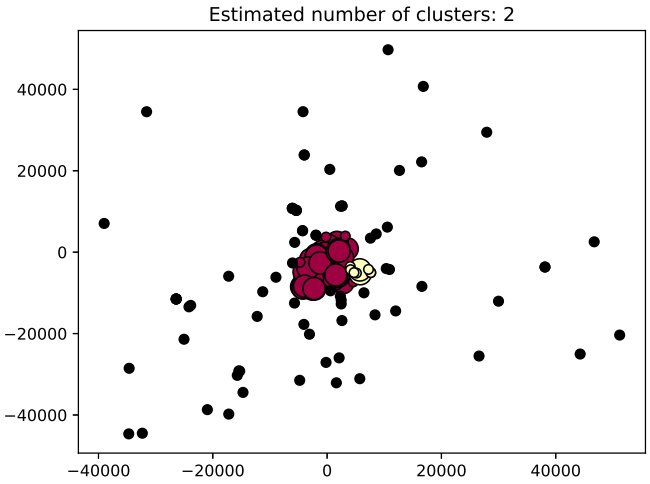
(g) Hour 06:00



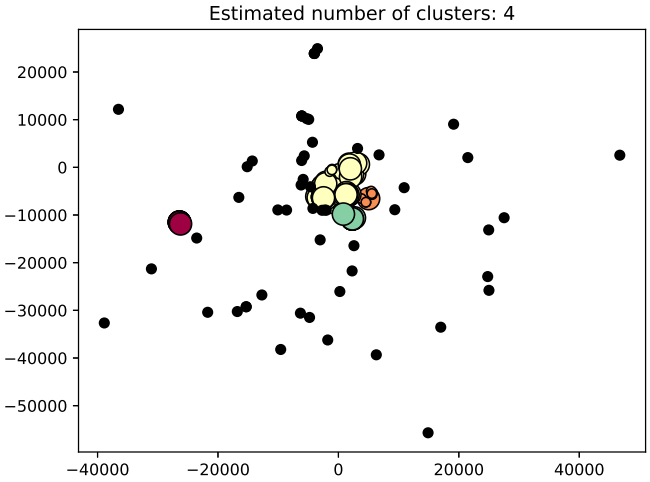
(h) Hour 07:00



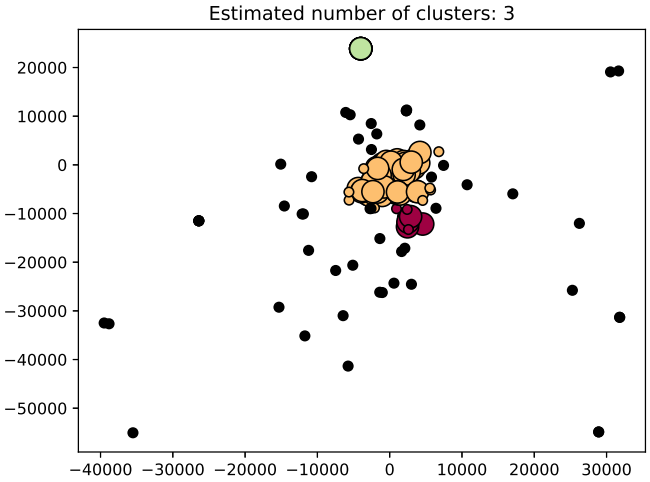
(i) Hour 08:00



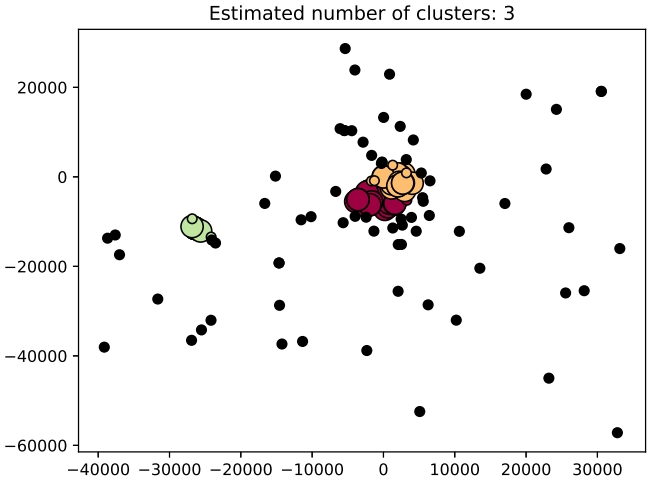
(j) Hour 09:00



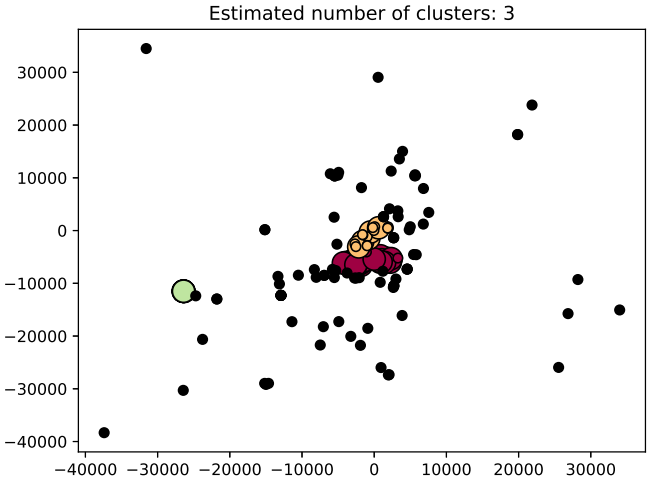
(k) Hour 10:00



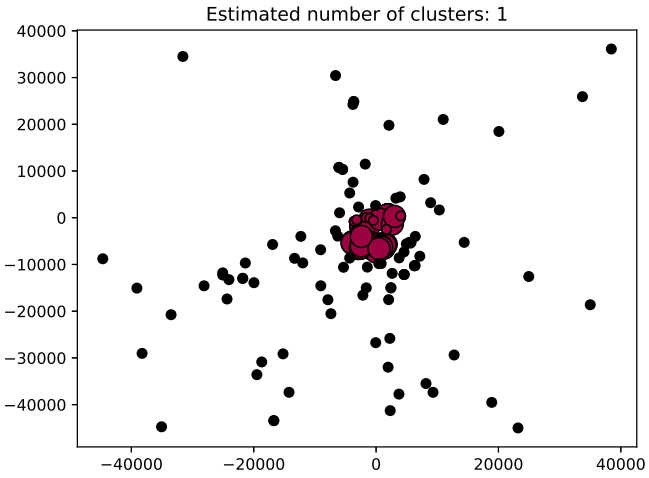
(l) Hour 11:00



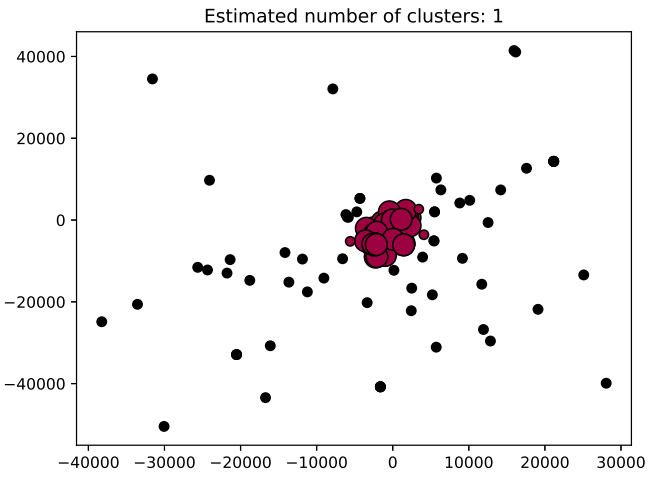
(m) Hour 12:00



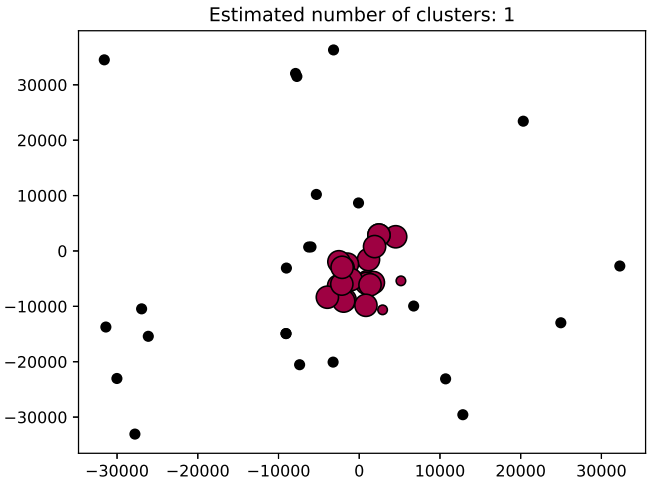
(n) Hour 13:00



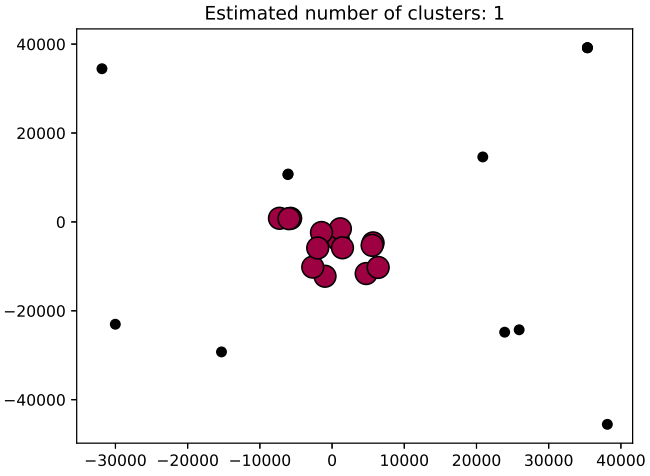
(o) Hour 14:00



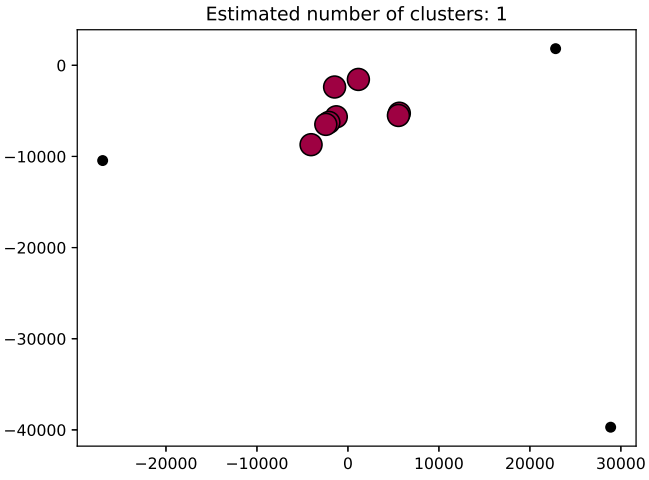
(p) Hour 15:00



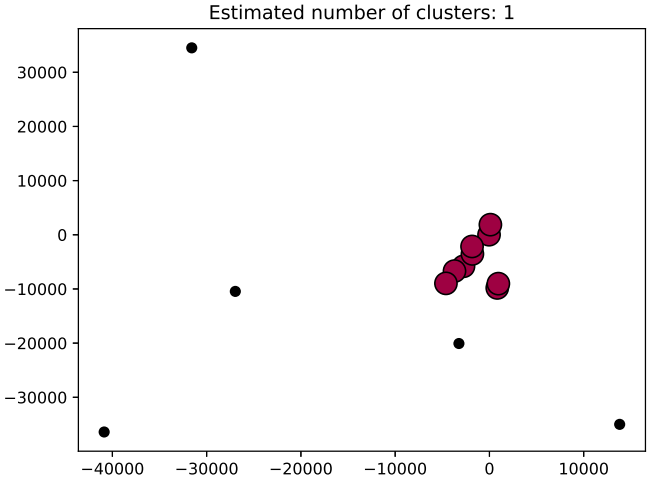
(q) Hour 16:00



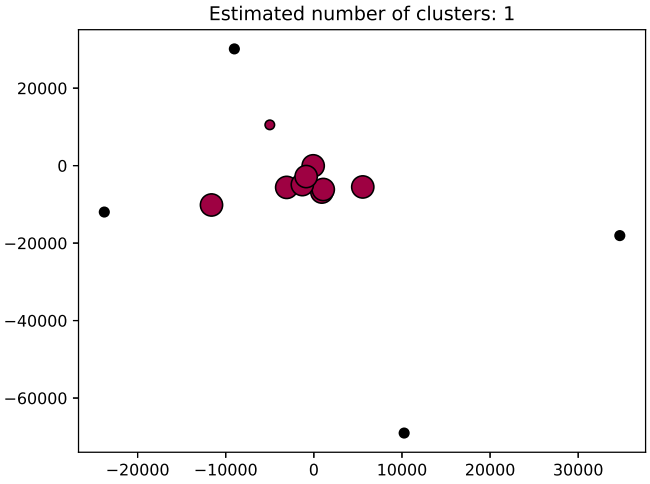
(r) Hour 17:00



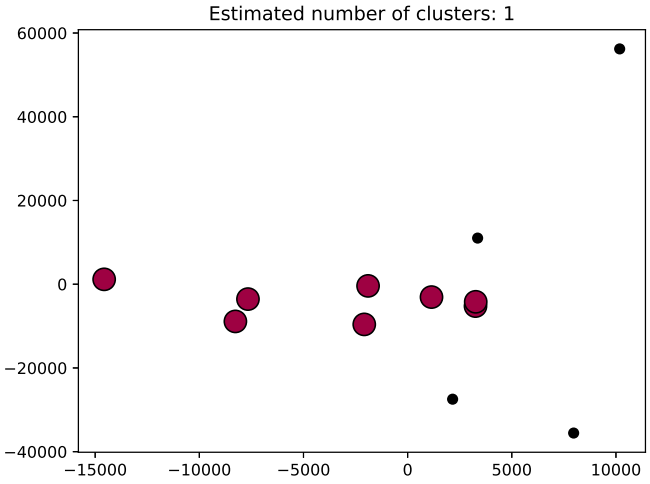
(s) Hour 18:00



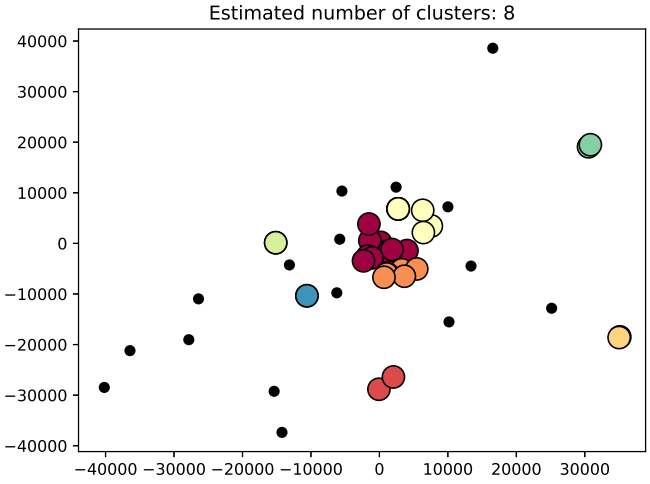
(t) Hour 19:00



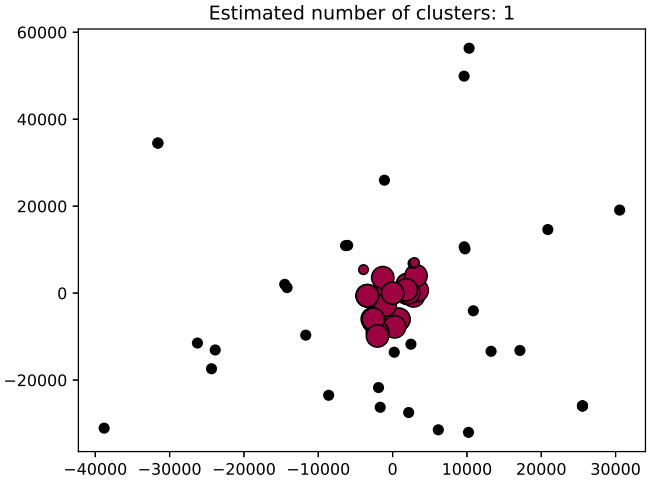
(u) Hour 20:00



(v) Hour 21:00



(w) Hour 22:00



(x) Hour 23:00

Figure 3. Clustering results for each hour.

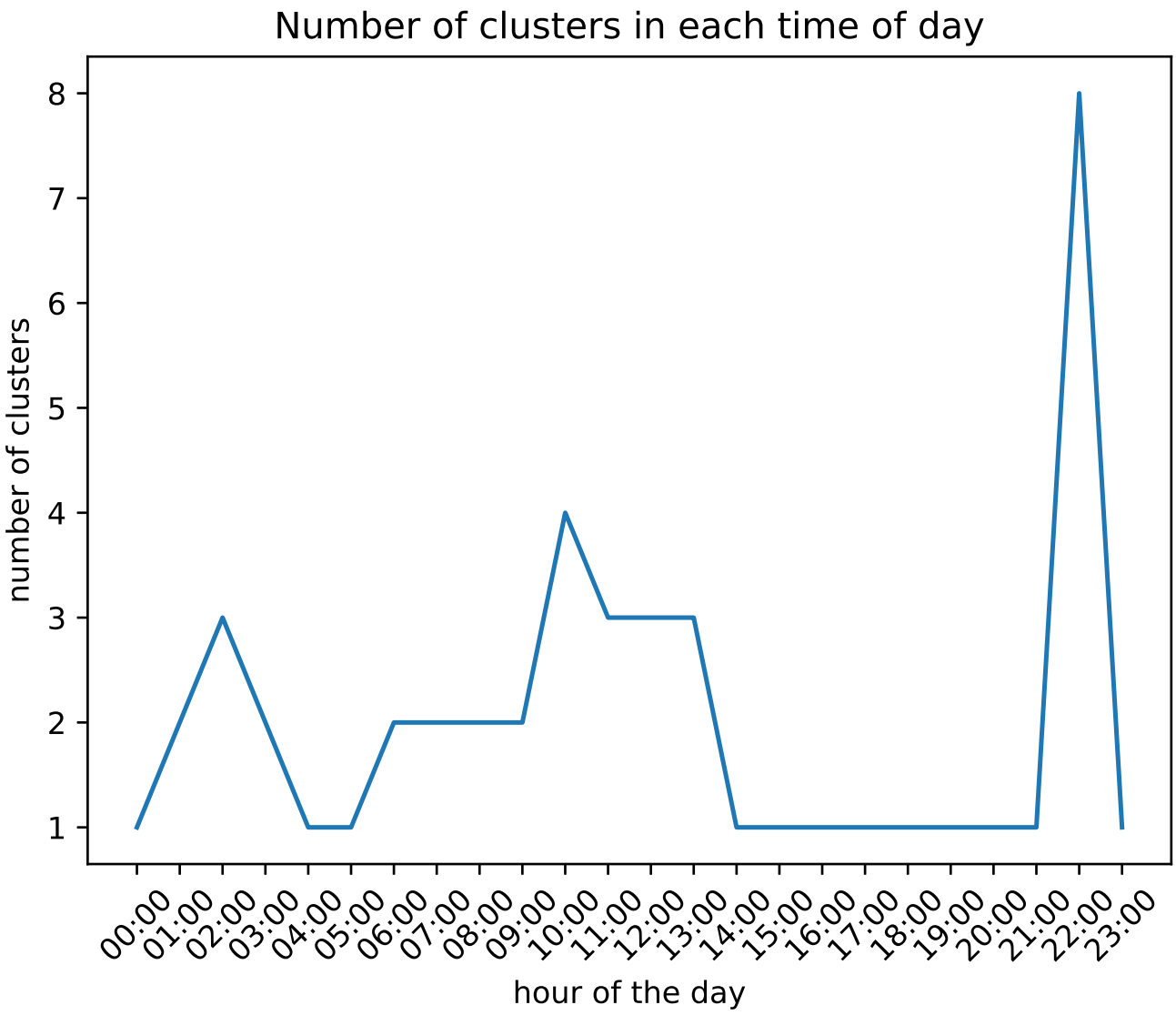


Figure 4. Variation of number of clusters with each hour.