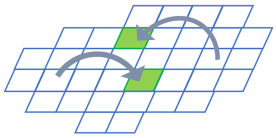


TransBigData

Release 0.5.2

Qing Yu

Jun 28, 2023

INSTALLATION

Main Functions

TransBigData is a Python package developed for transportation spatio-temporal big data processing and analysis. TransBigData provides fast and concise methods for processing common traffic spatio-temporal big data such as Taxi GPS data, bicycle sharing data and bus GPS data. It includes general methods such as rasterization, data quality analysis, data pre-processing, data set counting, trajectory analysis, GIS processing, map base map loading, coordinate and distance calculation, and data visualization.

Technical Features

- Provides different processing methods for different stages of traffic spatio-temporal big data analysis.
- The code with TransBigData is clean, efficient, flexible, and easy to use, allowing complex data tasks to be achieved with concise code.

CHAPTER

ONE

INTRODUCTION

CHAPTER

TWO

QUICK START

Before installing TransBigData, make sure that you have installed the geopandas package:
https://geopandas.org/index.html
If you already have geopandas installed, run the following code directly from the command prompt to install it

```
pip install -U transbigdata
```

The following example shows how to use the TransBigData to quickly extract trip OD from taxi GPS data

```
import transbigdata as tbd
import pandas as pd
data = pd.read_csv('TaxiData-Sample.csv',header = None)
data.columns = ['VehicleNum','time','slon','slat','OpenStatus','Speed']
data
```

	VehicleNum	time	slon	slat	OpenStatus	Speed
0	34745	20:27:43	113.806847	22.623249	1	27
1	34745	20:24:07	113.809898	22.627399	0	0
2	34745	20:24:27	113.809898	22.627399	0	0
3	34745	20:22:07	113.811348	22.628067	0	0
4	34745	20:10:06	113.819885	22.647800	0	54
...	...	...	...	...	...	...
544994	28265	21:35:13	114.321503	22.709499	0	18
544995	28265	09:08:02	114.322701	22.681700	0	0
544996	28265	09:14:31	114.336700	22.690100	0	0
544997	28265	21:19:12	114.352600	22.728399	0	0
544998	28265	19:08:06	114.137703	22.621700	0	0

544999 rows × 6 columns

Use the *tbd.taxigps_to_od* method and pass in the corresponding column name to extract the trip OD:

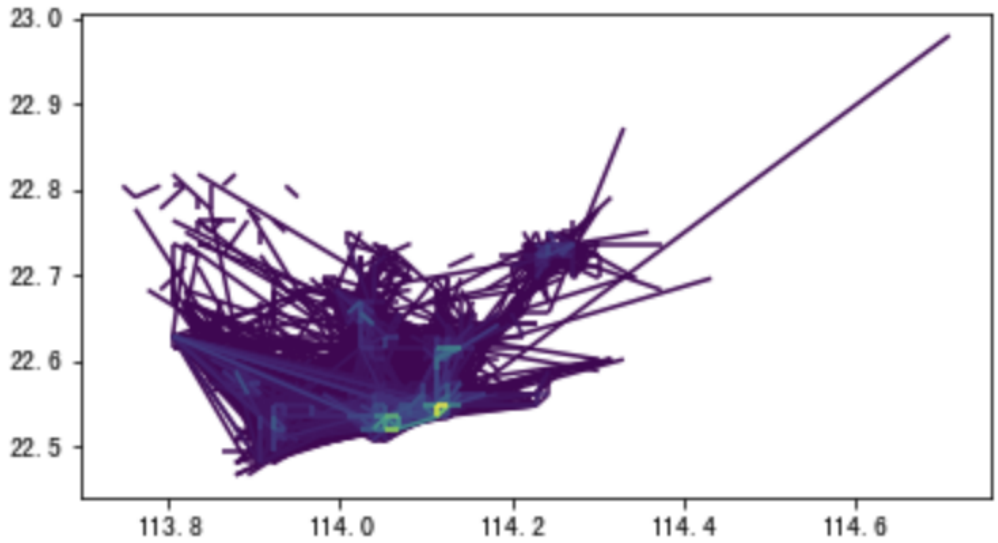
```
#Extract OD from GPS data
oddata = tbd.taxigps_to_od(data,col = ['VehicleNum','time','slon','slat','OpenStatus'])
oddata
```

	VehicleNum	stime	slon	slat	etime	elon	elat
20000	22396	00:19:41	114.013016	22.664818	00:23:01	114.021400	22.663918
20094	22396	00:41:51	114.021767	22.640200	00:43:44	114.026070	22.640266
20217	22396	00:45:44	114.028099	22.645082	00:47:44	114.030380	22.650017
19093	22396	01:08:26	114.034897	22.616301	01:16:34	114.035614	22.646717
19712	22396	01:26:06	114.046021	22.641251	01:34:48	114.066048	22.636183
...	...	...	...	...	...	...	...
183652	36805	22:49:12	114.114365	22.550632	22:50:40	114.115501	22.557983
183037	36805	22:52:07	114.115402	22.558083	23:03:12	114.118484	22.547867
180332	36805	23:03:45	114.118484	22.547867	23:20:09	114.133286	22.617750
181804	36805	23:36:19	114.112968	22.549601	23:43:12	114.089485	22.538918
180899	36805	23:46:14	114.091217	22.540768	23:53:36	114.120354	22.544300

5816 rows × 7 columns

Aggregate OD into grids:

```
#define bounds
bounds = [113.6,22.4,114.8,22.9]
#obtain the gridding parameters
params = tbd.grid_params(bounds = bounds,accuracy = 1500)
#gridding OD data and aggregate
od_gdf = tbd.odagg_grid(oddata,params)
od_gdf.plot(column = 'count')
```



CHAPTER

THREE

INSTALLATION

3.1 Installation

3.1.1 Installation

TransBigData support Python >= 3.6.

TransBigData depends on geopandas, Before installing TransBigData, you need to install geopandas based on this link
If you already have geopandas installed, run the following code directly from the command prompt to install it:

```
pip install -U transbigdata
```

You can also install TransBigData by conda-forge, this will automaticaly solve the dependency, it can be installed with:

```
conda install -c conda-forge transbigdata
```

To import TransBigData, run the following code in Python:

```
import transbigdata as tbd
```

3.1.2 Dependency

TransBigData depends on the following packages

- *pandas*
- *shapely*
- *rtree*
- *geopandas*
- *scipy*
- *matplotlib*
- *networkx* (optional)
- *igraph* (optional)
- *osmnx* (optional)
- *seaborn* (optional)
- *keplergl* (optional)

CHAPTER

FOUR

EXAMPLE GALLERY

4.1 Example gallery

Here are some examples showing what you can do with *TransBigData*. The *ipynb* and *Data* are in This link

4.1.1 Basic

1 Processing & visualizing taxi GPS data

Taxi GPS data processing

In this example, we will introduce how to use the TransBigData package to efficiently process Taxi GPS data. Firstly, import the TransBigData and read the data using pandas

[1]:
```
import transbigdata as tbd
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt

# Read data
data = pd.read_csv('data/TaxiData-Sample.csv', header=None)
data.columns = ['VehicleNum', 'Time', 'Lng', 'Lat', 'OpenStatus', 'Speed']
data.head()
```

[1]:
```
   VehicleNum      Time         Lng        Lat  OpenStatus  Speed
0       34745  20:27:43  113.806847  22.623249           1     27
1       34745  20:24:07  113.809898  22.627399           0      0
2       34745  20:24:27  113.809898  22.627399           0      0
3       34745  20:22:07  113.811348  22.628067           0      0
4       34745  20:10:06  113.819885  22.647800           0     54
```

[2]:
```
# Read the GeoDataFrame of the study area
sz = gpd.read_file(r'data/sz.json')
sz.crs = None
sz.head()
```

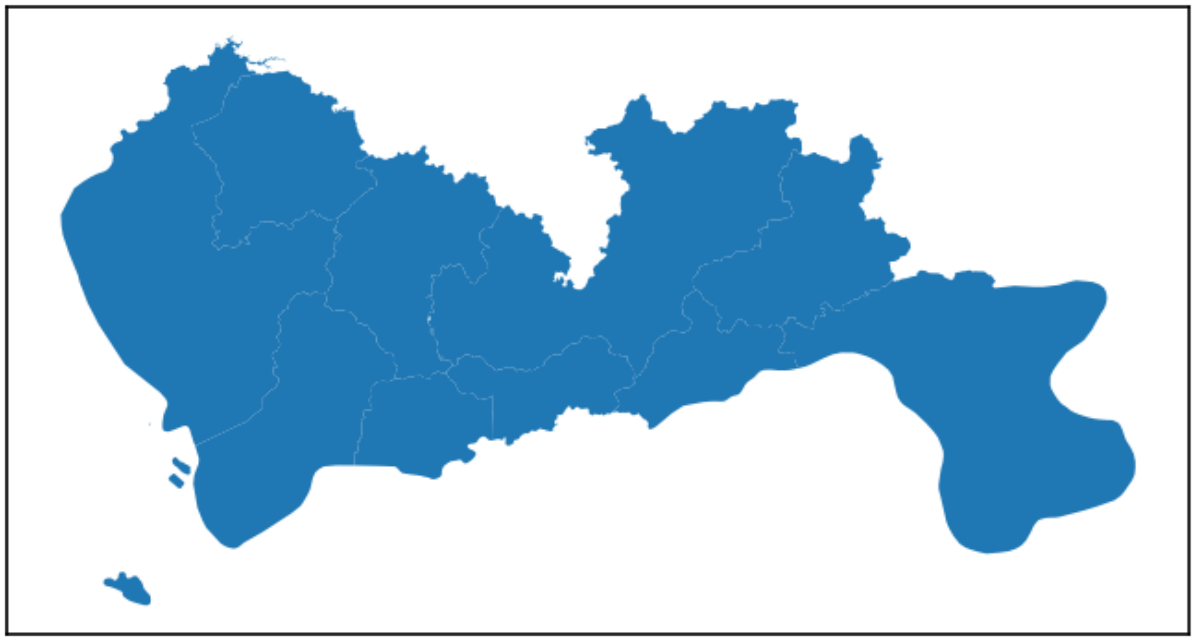
[2]:
```
   centroid_x  centroid_y  qh  \
0  114.143157   22.577605
```

(continues on next page)

(continued from previous page)

```
1  114.041535   22.546180
2  114.270206   22.596432
3  113.851387   22.679120
4  113.926290   22.766157

                                              geometry
0  POLYGON ((114.10006 22.53431, 114.10083 22.534...
1  POLYGON ((113.98578 22.51348, 114.00553 22.513...
2  POLYGON ((114.19799 22.55673, 114.19817 22.556...
3  MULTIPOLYGON (((113.81831 22.54676, 113.81948 ...
4  POLYGON ((113.99768 22.76643, 113.99704 22.766...
```

[3]:
```
fig = plt.figure(1, (8, 3), dpi=150)
ax1 = plt.subplot(111)
sz.plot(ax=ax1)
plt.xticks([], fontsize=10)
plt.yticks([], fontsize=10);
```



Data pre-processing

TransBigData integrates several common methods for data pre-processing. Using `the tbd.clean_outofshape` method, given the data and the GeoDataFrame of the study area, it can delete the data outside the study area. The `tbd.clean_taxi_status` method can filters out the data with instantaneous changes in passenger status(OpenStatus). When using the preprocessing method, the corresponding column names need to be passed in as parameters

[4]:
```
# Data Preprocessing
# Delete the data outside of the study area
data = tbd.clean_outofshape(data, sz, col=['Lng', 'Lat'], accuracy=500)
```

(continues on next page)

(continued from previous page)

```
# Delete the data with instantaneous changes in passenger status
data = tbd.clean_taxi_status(data, col=['VehicleNum', 'Time', 'OpenStatus'])
```

Data Gridding

The most basic way to express the data distribution is in the form of geograpic grids; after the data gridding, each GPS data point is mapped to the corresponding grid. For data gridding, you need to determine the gridding parameters at first(which can be interpreted as defining a grid coordinate system):

[5]:
```
# Data gridding
# Define the bounds and generate gridding parameters
bounds = [113.6, 22.4, 114.8, 22.9]
params = tbd.area_to_params(bounds, accuracy=500)
print(params)
```

```
{'slon': 113.6, 'slat': 22.4, 'deltalon': 0.004872390756896538, 'deltalat': 0.
↪004496605206422906, 'theta': 0, 'method': 'rect', 'gridsize': 500}
```

After obtaining the gridding parameters, the next step is to map the GPS is to their corresponding grids. Using the `tbd.GPS_to_grids`, it will generate the `LONCOL` column and the `LATCOL` column. The two columns together can specify a grid:

[6]:
```
# Mapping GPS data to grids
data['LONCOL'], data['LATCOL'] = tbd.GPS_to_grid(data['Lng'], data['Lat'], params)
data.head()
```

[6]:

	VehicleNum	Time	Lng	Lat	OpenStatus	Speed	LONCOL	LATCOL
0	34745	20:27:43	113.806847	22.623249	1	27	42	50
1	27368	09:08:53	113.805893	22.624996	0	49	42	50
2	22998	10:51:10	113.806931	22.624166	1	54	42	50
3	22998	10:11:50	113.805946	22.625433	0	43	42	50
4	22998	10:12:05	113.806381	22.623833	0	60	42	50

Count the amount of data in each grids:

[7]:
```
# Aggregate data into grids
datatest = data.groupby(['LONCOL', 'LATCOL'])['VehicleNum'].count().reset_index()
datatest.head()
```

[7]:

	LONCOL	LATCOL	VehicleNum
0	36	63	2
1	36	66	1
2	36	67	8
3	37	62	9
4	37	63	8

Generate the geometry of the grids and transform it into a GeoDataFrame:

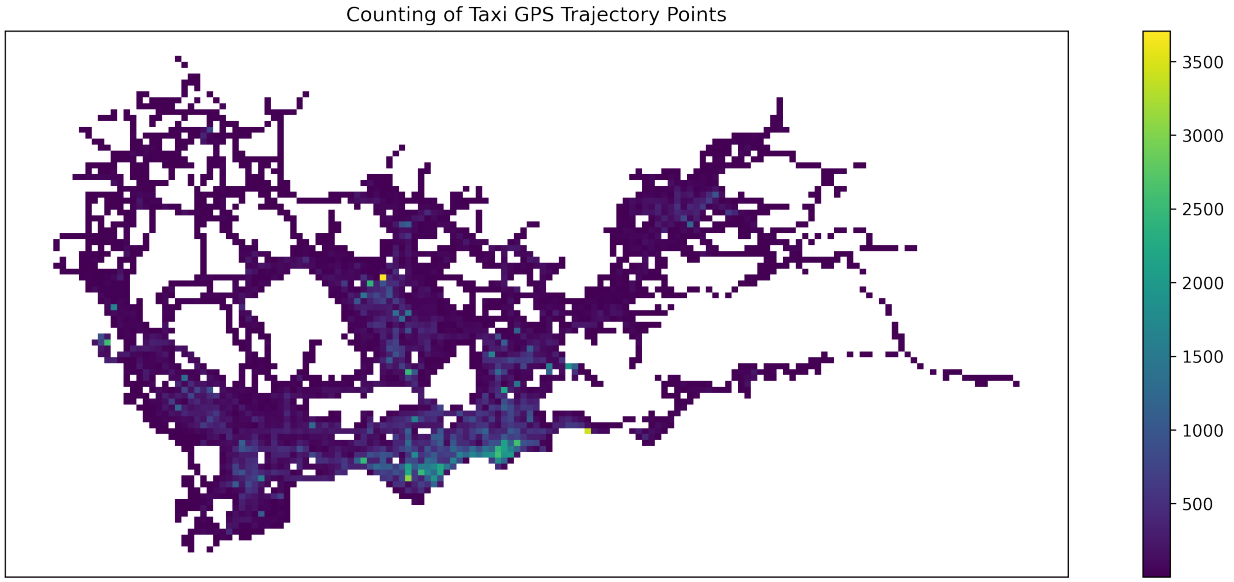
[8]:
```
# Generate the geometry for grids
datatest['geometry'] = tbd.grid_to_polygon([datatest['LONCOL'], datatest['LATCOL']],␣
↪params)

# Change it into GeoDataFrame
# import geopandas as gpd
datatest = gpd.GeoDataFrame(datatest)
datatest.head()
```

[8]:

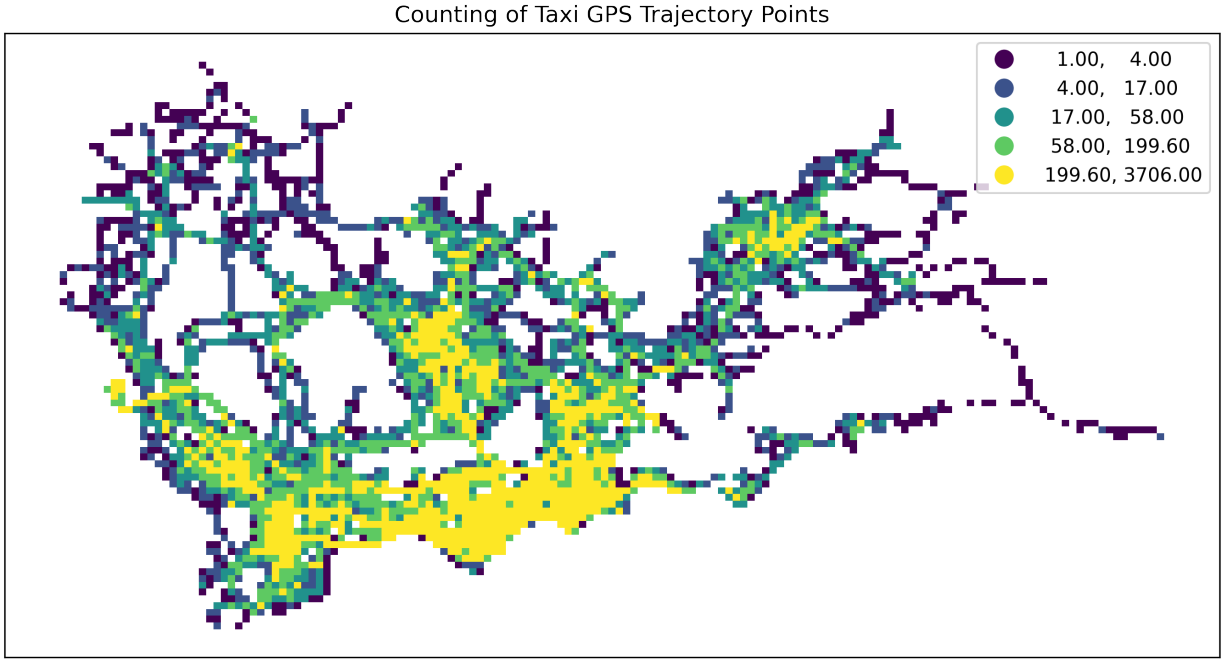
	LONCOL	LATCOL	VehicleNum	geometry
0	36	63	2	POLYGON ((113.77297 22.68104, 113.77784 22.681...
1	36	66	1	POLYGON ((113.77297 22.69453, 113.77784 22.694...
2	36	67	8	POLYGON ((113.77297 22.69902, 113.77784 22.699...
3	37	62	9	POLYGON ((113.77784 22.67654, 113.78271 22.676...
4	37	63	8	POLYGON ((113.77784 22.68104, 113.78271 22.681...

Plot the generated grids:

[9]:
```
# Plot the grids
fig = plt.figure(1, (16, 6), dpi=300)
ax1 = plt.subplot(111)

# tbd.plot_map(plt, bounds, zoom=10, style=4)
datatest.plot(ax=ax1, column='VehicleNum', legend=True)
plt.xticks([], fontsize=10)
plt.yticks([], fontsize=10)
plt.title('Counting of Taxi GPS Trajectory Points', fontsize=12);
```



[10]:
```
# Plot the grids
fig = plt.figure(1, (16, 6), dpi=300) # 68
ax1 = plt.subplot(111)
datatest.plot(ax=ax1, column='VehicleNum', legend=True, scheme='quantiles')
# plt.legend(fontsize=10)
plt.xticks([], fontsize=10)
plt.yticks([], fontsize=10)
plt.title('Counting of Taxi GPS Trajectory Points', fontsize=12);
```

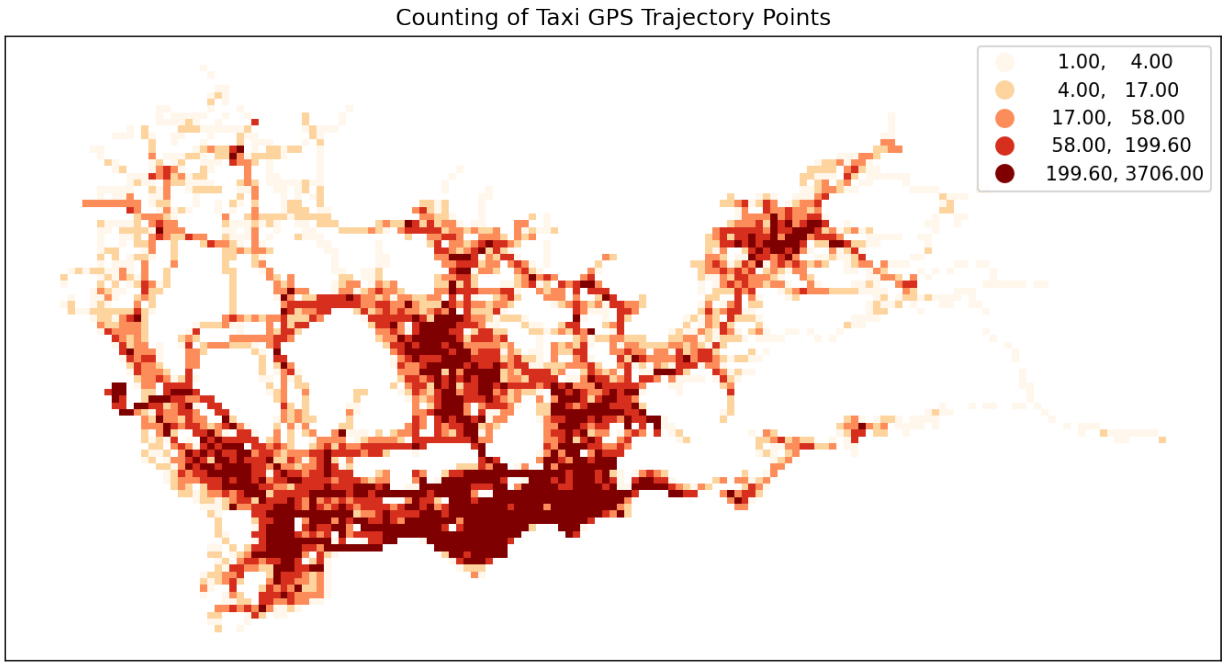


[11]:
```
# Plot the grids
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(1, (16, 6), dpi=150) # 68
ax1 = plt.subplot(111)
datatest.plot(ax=ax1, column='VehicleNum', legend=True, cmap='OrRd', scheme='quantiles')
# plt.legend(fontsize=10)
plt.xticks([], fontsize=10)
plt.yticks([], fontsize=10)
plt.title('Counting of Taxi GPS Trajectory Points', fontsize=12);
```



Origin-destination(OD) Extraction and aggregate taxi trips

Use the `tbd.taxigps_to_od` method and pass in the corresponding column name to extract the taxi trip OD:

[12]:
```
# Extract taxi OD from GPS data
oddata = tbd.taxigps_to_od(data,col = ['VehicleNum', 'Time', 'Lng', 'Lat', 'OpenStatus'])
oddata
```

[12]:

	VehicleNum	stime	slon	slat	etime	elon	\
427075	22396	00:19:41	114.013016	22.664818	00:23:01	114.021400	
131301	22396	00:41:51	114.021767	22.640200	00:43:44	114.026070	
417417	22396	00:45:44	114.028099	22.645082	00:47:44	114.030380	
376160	22396	01:08:26	114.034897	22.616301	01:16:34	114.035614	
21768	22396	01:26:06	114.046021	22.641251	01:34:48	114.066048	
...	...	...	...	...	...	...	
57666	36805	22:37:42	114.113403	22.534767	22:48:01	114.114365	
175519	36805	22:49:12	114.114365	22.550632	22:50:40	114.115501	
212092	36805	22:52:07	114.115402	22.558083	23:03:27	114.118484	
119041	36805	23:03:45	114.118484	22.547867	23:20:09	114.133286	
224103	36805	23:36:19	114.112968	22.549601	23:43:12	114.089485	

(continues on next page)

(continued from previous page)

```
             elat    ID
427075  22.663918     0
131301  22.640266     1
417417  22.650017     2
376160  22.646717     3
21768   22.636183     4
...           ...   ...
57666   22.550632  5332
175519  22.557983  5333
212092  22.547867  5334
119041  22.617750  5335
224103  22.538918  5336

[5337 rows x 8 columns]
```

Aggregate the extracted OD and generate LineString GeoDataFrame

[13]:
```
# Gridding and aggragate data
od_gdf = tbd.odagg_grid(oddata, params)
od_gdf.head()
```

```
/opt/anaconda3/lib/python3.8/site-packages/pandas/core/dtypes/cast.py:91:␣
↪ShapelyDeprecationWarning: The array interface is deprecated and will no longer work␣
↪in Shapely 2.0. Convert the '.coords' to a numpy array instead.
  values = construct_1d_object_array_from_listlike(values)
```

[13]:
```
      SLONCOL  SLATCOL  ELONCOL  ELATCOL  count      SHBLON     SHBLAT  \
0          40       62       45       68      1  113.794896  22.678790
3331      101       36       86       29      1  114.092111  22.561878
3330      101       35      105       30      1  114.092111  22.557381
3329      101       34      109       34      1  114.092111  22.552885
3328      101       34      103       34      1  114.092111  22.552885

          EHBLON     EHBLAT                                           geometry
0     113.819258  22.705769  LINESTRING (113.79490 22.67879, 113.81926 22.7...
3331  114.019026  22.530402  LINESTRING (114.09211 22.56188, 114.01903 22.5...
3330  114.111601  22.534898  LINESTRING (114.09211 22.55738, 114.11160 22.5...
3329  114.131091  22.552885  LINESTRING (114.09211 22.55288, 114.13109 22.5...
3328  114.101856  22.552885  LINESTRING (114.09211 22.55288, 114.10186 22.5...
```

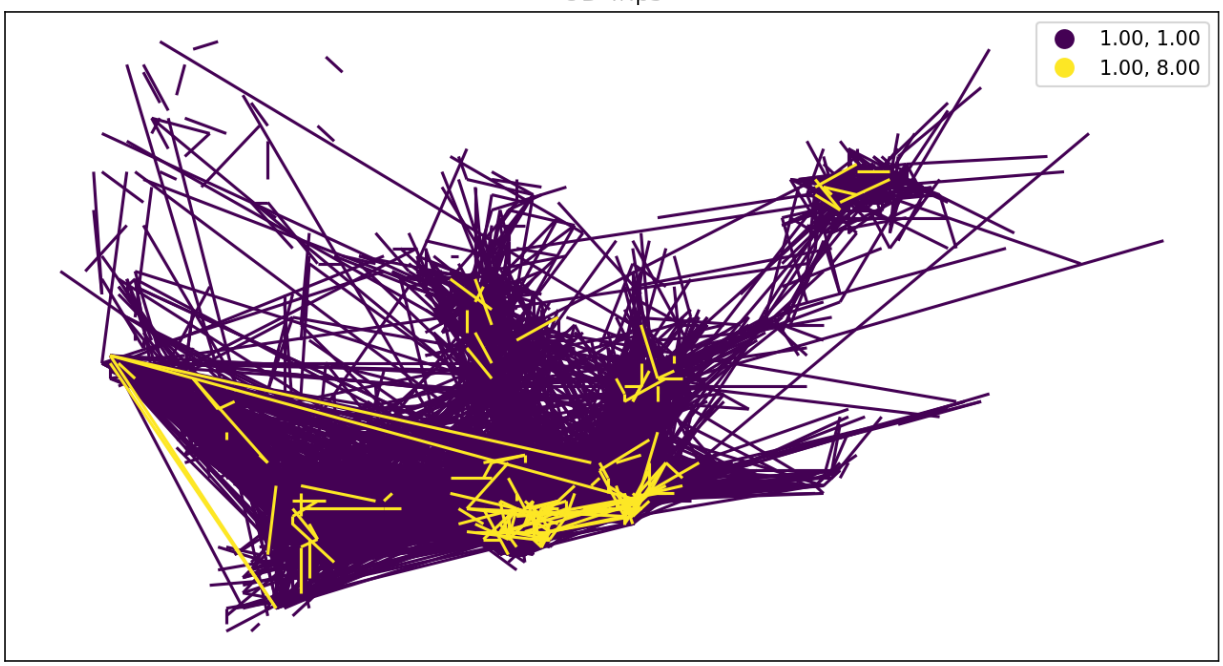
[14]:
```
# Plot the grids
fig = plt.figure(1, (16, 6), dpi=150) # 68
ax1 = plt.subplot(111)
# data_grid_count.plot(ax=ax1, column='VehicleNum', legend=True, cmap='OrRd', scheme=
↪'quantiles')

od_gdf.plot(ax=ax1, column='count', legend=True, scheme='quantiles')
plt.xticks([], fontsize=10)
plt.yticks([], fontsize=10)
plt.title('OD Trips', fontsize=12);
```

```
/opt/anaconda3/lib/python3.8/site-packages/mapclassify/classifiers.py:238: UserWarning:␣
↪Warning: Not enough unique values in array to form k classes
```

(continues on next page)

(continued from previous page)

```
  Warn(
/opt/anaconda3/lib/python3.8/site-packages/mapclassify/classifiers.py:241: UserWarning:␣
↪Warning: setting k to 2
  Warn("Warning: setting k to %d" % k_q, UserWarning)
```

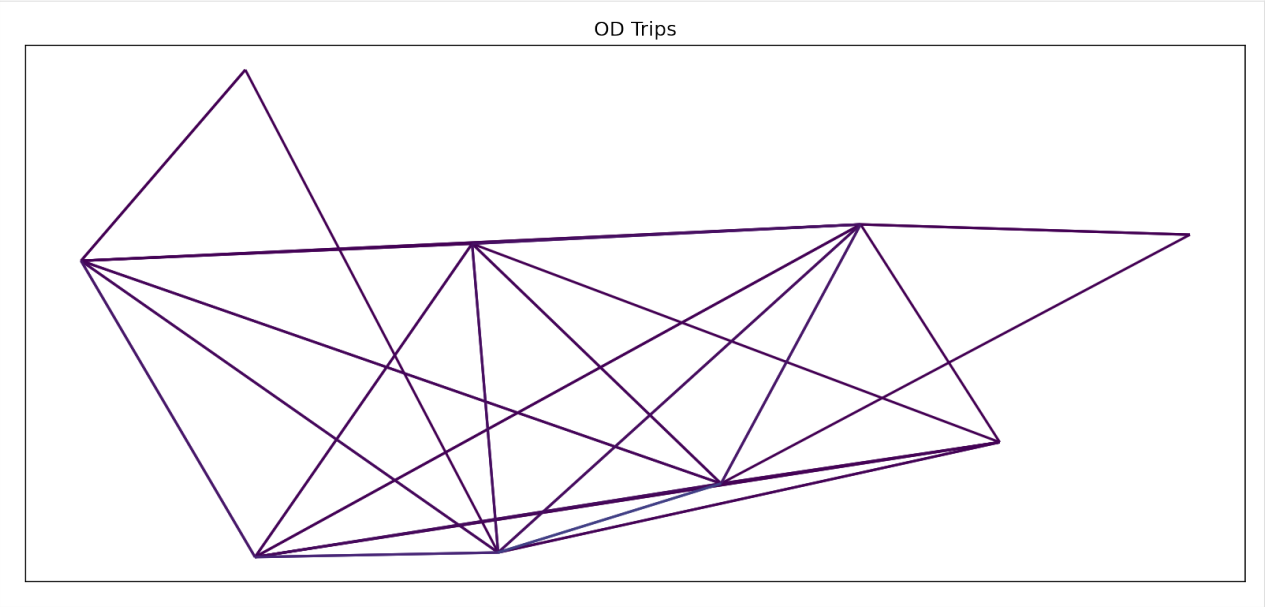


Aggregate OD into polygons

TransBigData also provides the method for aggregating OD into polygons

[15]:
```
# Aggragate OD data to polygons
# without passing gridding parameters, the algorithm will map the data
# to polygons directly using their coordinates
od_gdf = tbd.odagg_shape(oddata, sz, round_accuracy=6)
fig = plt.figure(1, (16, 6), dpi=150) # 68
ax1 = plt.subplot(111)
od_gdf.plot(ax=ax1, column='count')
plt.xticks([], fontsize=10)
plt.yticks([], fontsize=10)
plt.title('OD Trips', fontsize=12);
```

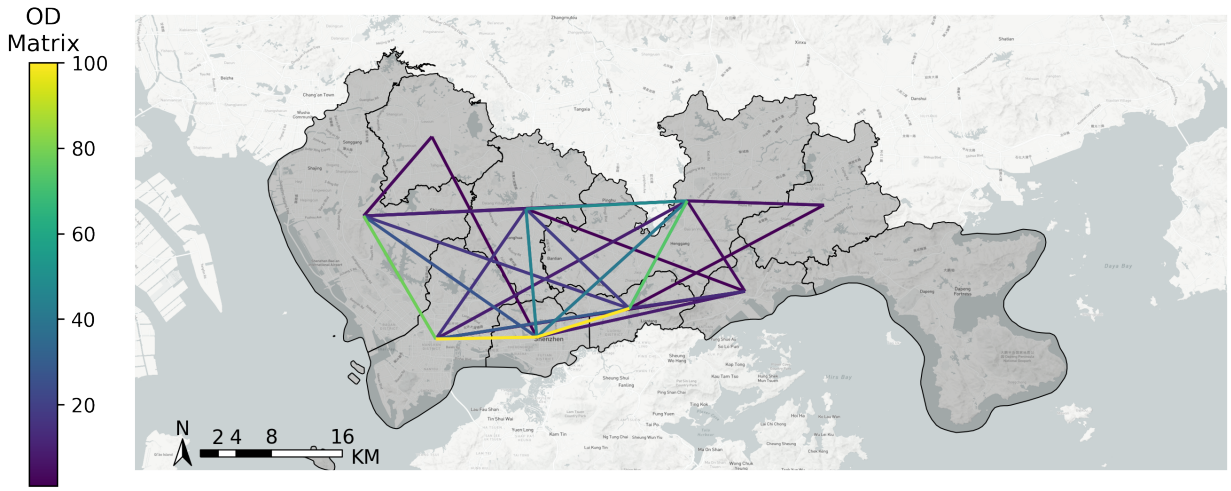
```
/opt/anaconda3/lib/python3.8/site-packages/pandas/core/dtypes/cast.py:91:␣
↪ShapelyDeprecationWarning: The array interface is deprecated and will no longer work␣
↪in Shapely 2.0. Convert the '.coords' to a numpy array instead.
  values = construct_1d_object_array_from_listlike(values)
```



Matplotlib-based map drawing

TransBigData also provide basemap loading in matplotlib. Before using this method, you need to set your mapbox-token and the storage location for the basemap, see: this linktbd.plot_map to add basemap and tbd.plotscale to add scale and compass:

[16]:
```
# Create figure
fig = plt.figure(1, (10, 10), dpi=300)
ax = plt.subplot(111)
plt.sca(ax)

# Load basemap
tbd.plot_map(plt, bounds, zoom=12, style=4)

# Define an ax for colorbar
cax = plt.axes([0.05, 0.33, 0.02, 0.3])
plt.title('OD\nMatrix')
plt.sca(ax)

# Plot the OD
od_gdf.plot(ax=ax, vmax=100, column='count', cax=cax, legend=True)

# Plot the polygons
sz.plot(ax=ax, edgecolor=(0, 0, 0, 1), facecolor=(0, 0, 0, 0.2), linewidths=0.5)

# Add compass and scale
tbd.plotscale(ax, bounds=bounds, textsize=10, compasssize=1, accuracy=2000, rect=[0.06,␣
↪0.03], zorder=10)
plt.axis('off')
plt.xlim(bounds[0], bounds[2])
plt.ylim(bounds[1], bounds[3])
plt.show()
```



Extraction of taxi trajectpries

Using `tbd.taxigps_traj_point` method, inputing GPS data and OD data, trajectory points can be extracted

[17]:
```
data_deliver, data_idle = tbd.taxigps_traj_point(data,oddata,col=['VehicleNum',
                                                                  'Time',
                                                                  'Lng',
                                                                  'Lat',
                                                                  'OpenStatus'])
```

[18]:
```
data_deliver.head()
```

[18]:
```
        VehicleNum      Time         Lng        Lat  OpenStatus  Speed  \
427075       22396  00:19:41  114.013016  22.664818           1   63.0
427085       22396  00:19:49  114.014030  22.665483           1   55.0
416622       22396  00:21:01  114.018898  22.662500           1    1.0
427480       22396  00:21:41  114.019348  22.662300           1    7.0
416623       22396  00:22:21  114.020615  22.663366           1    0.0

        LONCOL  LATCOL   ID  flag
427075    85.0    59.0  0.0   1.0
427085    85.0    59.0  0.0   1.0
416622    86.0    58.0  0.0   1.0
427480    86.0    58.0  0.0   1.0
416623    86.0    59.0  0.0   1.0
```

[19]:
```
data_idle.head()
```

[19]:
```
        VehicleNum      Time         Lng        Lat  OpenStatus  Speed  \
416628       22396  00:23:01  114.021400  22.663918           0   25.0
401744       22396  00:25:01  114.027115  22.662100           0   25.0
394630       22396  00:25:41  114.024551  22.659834           0   21.0
394671       22396  00:26:21  114.022797  22.658367           0    0.0
394672       22396  00:26:29  114.022797  22.658367           0    0.0
```

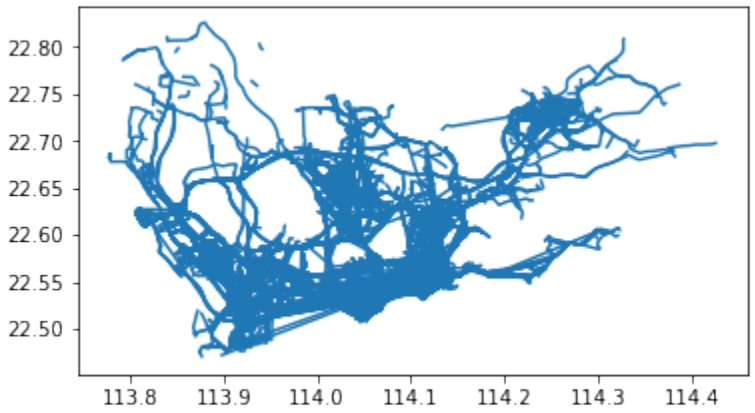
(continues on next page)

(continued from previous page)

```
        LONCOL  LATCOL   ID  flag
416628    86.0    59.0  0.0   0.0
401744    88.0    58.0  0.0   0.0
394630    87.0    58.0  0.0   0.0
394671    87.0    57.0  0.0   0.0
394672    87.0    57.0  0.0   0.0
```

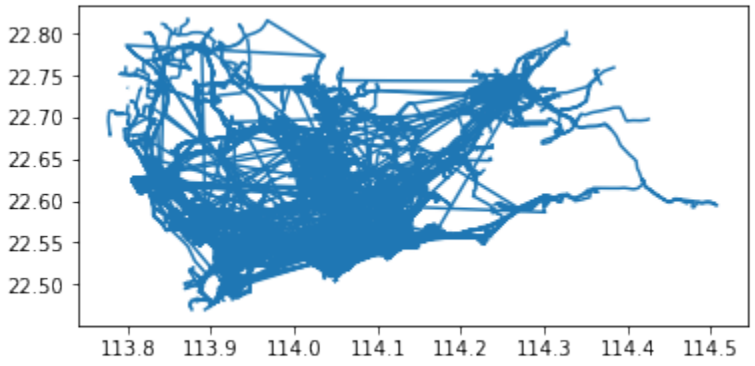
Generate delivery and idle trajectories from trajectory points

[20]:
```
traj_deliver = tbd.points_to_traj(data_deliver)
traj_deliver.plot();
```



[21]:
```
traj_idle = tbd.points_to_traj(data_idle[data_idle['OpenStatus'] == 0])
traj_idle.plot()
```

[21]:
```
<AxesSubplot:>
```



Trajectories visualization

Built-in visualization capabilities of `TransBigData` leverage the visualization package `keplergl` to interactively visualize data on Jupyter notebook with simple code. To use this method, please install the `keplergl` package for python:

```
pip install keplergl
```

Detailed information please see this link

Visualization of trajectory data:

[22]:
```
tbd.visualization_trip(data_deliver)
```

```
Processing trajectory data...
Generate visualization...
User Guide: https://docs.kepler.gl/docs/keplergl-jupyter
```

```
KeplerGl(config={'version': 'v1', 'config': {'visState': {'filters': [], 'layers': [{'id
↪': 'hizm36i', 'type': ...
```

2 Grid-base processing framework of TransBigData

This notebook will introduce the core functions embedded in the Transbigdata package

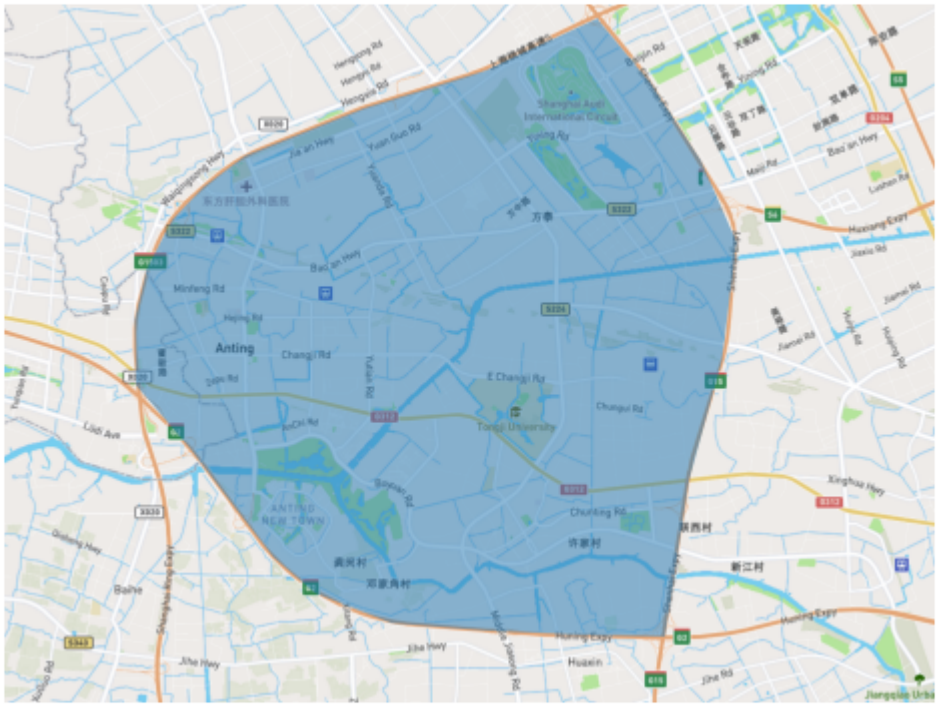
[1]:
```
import transbigdata as tbd
import geopandas as gpd
import pandas as pd
import matplotlib.pyplot as plt
import pprint
import random
```

[2]:
```
# this is a shp file, the sample area is part of Jiading district, Shanghai, China
jiading_polygon = gpd.read_file(r'data/jiading_polygon/jiading_polygon.shp')
jiading_polygon.head()
```

[2]:

	id	geometry
0	1	POLYGON ((121.22538 31.35142, 121.22566 31.350...

[3]:
```
jiading_rec_bound = [121.1318, 31.2484, 121.2553, 31.3535]

fig = plt.figure(1, (6, 6), dpi=100)
ax = plt.subplot(111)
plt.sca(ax)
tbd.plot_map(plt, bounds=jiading_rec_bound, zoom=13, style=2)

jiading_polygon.plot(ax=ax, alpha=0.5)
plt.axis('off');
```



transbigdata.area_to_grid(location, accuracy=500, method='rect', params='auto')

[4]:
```
# generate the default grid
grid_rec, params_rec = tbd.area_to_grid(jiading_polygon)
pprint.pprint(params_rec)
grid_rec.head()
```

```
{'deltalat': 0.004496605206422906,
 'deltalon': 0.005262604989003139,
 'gridsize': 500,
 'method': 'rect',
 'slat': 31.25168182840957,
 'slon': 121.13797109957756,
 'theta': 0}
```

[4]:

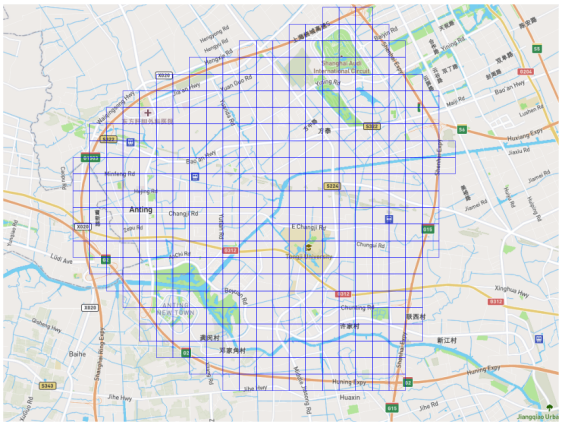
	LONCOL	LATCOL	geometry
171	9	0	POLYGON ((121.18270 31.24943, 121.18797 31.249...
174	10	0	POLYGON ((121.18797 31.24943, 121.19323 31.249...
177	11	0	POLYGON ((121.19323 31.24943, 121.19849 31.249...
180	12	0	POLYGON ((121.19849 31.24943, 121.20375 31.249...
183	13	0	POLYGON ((121.20375 31.24943, 121.20902 31.249...

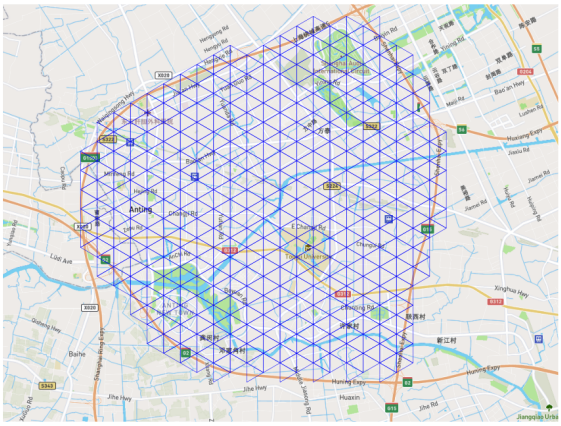
[5]:
```
# generate triangle grid
grid_tri, params_tri = tbd.area_to_grid(jiading_polygon, method='tri')  # to do: bug␣
↪need to be fixed here
pprint.pprint(params_tri)
grid_tri.head()
```

```
{'deltalat': 0.004496605206422906,
 'deltalon': 0.005262604989003139,
 'gridsize': 500,
 'method': 'tri',
 'slat': 31.25168182840957,
 'slon': 121.13797109957756,
 'theta': 0}
```

[5]:
```
    loncol_1  loncol_2  loncol_3  \
22         6         2        -5
24         7         2        -5
27         8         3        -5
28         8         3        -6
30         9         3        -6

                                              geometry
22  POLYGON ((121.17481 31.25947, 121.16955 31.256...
24  POLYGON ((121.17481 31.25428, 121.18007 31.256...
27  POLYGON ((121.18007 31.25168, 121.18533 31.254...
28  POLYGON ((121.18533 31.25947, 121.18007 31.256...
30  POLYGON ((121.18533 31.25428, 121.19060 31.256...
```

[6]:
```
# Visualization
fig = plt.figure(1, (12, 8), dpi=200)
ax1 = plt.subplot(121)
plt.sca(ax1)
tbd.plot_map(plt, bounds=jiading_rec_bound, zoom=13, style=2)
grid_rec.plot(ax=ax1, lw=0.2, edgecolor='blue', facecolor="None")

plt.axis('off');

ax2 = plt.subplot(122)
plt.sca(ax2)
tbd.plot_map(plt, bounds=jiading_rec_bound, zoom=13, style=2)
grid_tri.plot(ax=ax2, lw=0.2, edgecolor='blue', facecolor="None")

plt.axis('off');
```





transbigdata.area_to_params(location, accuracy=500, method='rect')

Sometime, due to data sparisity, we do not need to generate all the grids. In such case, we can use `transbigdata.area_to_params`.

This method only creat a dictionary file for the grid, thus is much faster.

[7]:
```
params = tbd.area_to_params(jiading_polygon)
pprint.pprint(params)
```

```
{'deltalat': 0.004496605206422906,
 'deltalon': 0.005262604989003139,
 'gridsize': 500,
 'method': 'rect',
 'slat': 31.25168182840957,
 'slon': 121.13797109957756,
 'theta': 0}
```

transbigdata.GPS_to_grid(lon, lat, params)

The next common step is to know which grid does each trajectory point belong to.

[8]:
```
# First, we generate some random GPS points (20 points in this case)

lon_list, lat_list = [], []
for i in range(20):
    gps_lon = random.uniform(jiading_rec_bound[0], jiading_rec_bound[2])
    gps_lat = random.uniform(jiading_rec_bound[1], jiading_rec_bound[3])
    lon_list.append(gps_lon)
    lat_list.append(gps_lat)

gps_random = pd.DataFrame({'veh_id': range(20),
                           'lon': lon_list,
                           'lat': lat_list,
                          })

gps_random.head()
```

[8]:

	veh_id	lon	lat
0	0	121.204726	31.266296
1	1	121.168077	31.326952
2	2	121.142706	31.315498
3	3	121.215899	31.339561
4	4	121.217937	31.269540

[9]:
```
# match each point to the rect grid
gps_random['LonCol'], gps_random['LatCol'] = tbd.GPS_to_grid(gps_random['lon'], gps_
↪random['lat'], params_rec)

gps_random.head()
```

[9]:

	veh_id	lon	lat	LonCol	LatCol
0	0	121.204726	31.266296	13	3

(continues on next page)

(continued from previous page)

```
1       1  121.168077  31.326952       6      17
2       2  121.142706  31.315498       1      14
3       3  121.215899  31.339561      15      20
4       4  121.217937  31.269540      15       4
```

transbigdata.grid_to_centre(gridid, params)

The center location of each grid can acquired using `transbigdata.grid_to_centre`

[10]:
```
# Use the matched grid as example
gps_random['LonGridCenter'], gps_random['LatGridCenter'] = \
tbd.grid_to_centre([gps_random['LonCol'], gps_random['LatCol']], params_rec)

# check the matched results
gps_random.head()
```

[10]:
```
   veh_id         lon        lat  LonCol  LatCol  LonGridCenter  LatGridCenter
0       0  121.204726  31.266296      13       3     121.206385      31.265172
1       1  121.168077  31.326952       6      17     121.169547      31.328124
2       2  121.142706  31.315498       1      14     121.143234      31.314634
3       3  121.215899  31.339561      15      20     121.216910      31.341614
4       4  121.217937  31.269540      15       4     121.216910      31.269668
```

transbigdata.grid_to_polygon(gridid, params)

For visualization convenience, grid parameters can be transformed into `geometry` format

[11]:
```
# Use the matched grid as example again
gps_random['grid_geo_polygon'] = tbd.grid_to_polygon([gps_random['LonCol'], gps_random[
↪'LatCol']], params_rec)

# check the matched results
gps_random.head()
```

[11]:
```
   veh_id         lon        lat  LonCol  LatCol  LonGridCenter  \
0       0  121.204726  31.266296      13       3     121.206385
1       1  121.168077  31.326952       6      17     121.169547
2       2  121.142706  31.315498       1      14     121.143234
3       3  121.215899  31.339561      15      20     121.216910
4       4  121.217937  31.269540      15       4     121.216910

   LatGridCenter                                    grid_geo_polygon
0      31.265172  POLYGON ((121.2037536619401 31.262923341425626...
1      31.328124  POLYGON ((121.16691542701707 31.32587581431555...
2      31.314634  POLYGON ((121.14060240207206 31.31238599869628...
3      31.341614  POLYGON ((121.2142788719181 31.339365629934818...
4      31.269668  POLYGON ((121.2142788719181 31.26741994663205,...
```

transbigdata.grid_to_area(data, shape, params, col=['LONCOL', 'LATCOL'])

In addition to grid, there might be several districts. `transbigdata.grid_to_area` can be used to match the information.

In this case, there are only one district in `jiading_polygon`, the matched column is `id`.

[12]:
```
gps_matched = tbd.grid_to_area(gps_random, jiading_polygon, params_rec, col=['LonCol',
↪'LatCol'])

# check the matched results
gps_matched.head()
```

```
/Applications/anaconda3/envs/tbd/lib/python3.9/site-packages/transbigdata/grids.py:421:␣
↪UserWarning: CRS mismatch between the CRS of left geometries and the CRS of right␣
↪geometries.
Use `to_crs()` to reproject one of the input geometries to match the CRS of the other.

Left CRS: None
Right CRS: EPSG:4326

  data1 = gpd.sjoin(data1, shape)
```

[12]:

	veh_id	lon	lat	LonCol	LatCol	LonGridCenter	LatGridCenter	grid_geo_polygon	geometry	index_right	id
0	0	121.204726	31.266296	13	3	121.206385	31.265172	POLYGON ((121.2037536619401 31.262923341425626...	POINT (121.20638 31.26517)	0	1
1	1	121.168077	31.326952	6	17	121.169547	31.328124	POLYGON ((121.16691542701707 31.32587581431555...	POINT (121.16955 31.32812)	0	1
2	2	121.142706	31.315498	1	14	121.143234	31.314634	POLYGON ((121.14060240207206 31.31238599869628...	POINT (121.14323 31.31463)	0	1
3	3	121.215899	31.339561	15	20	121.216910	31.341614	POLYGON ((121.2142788719181 31.339365629934818...	POINT (121.21691 31.34161)	0	1
4	4	121.217937	31.269540	15	4	121.216910	31.269668	POLYGON ((121.2142788719181 31.26741994663205,...	POINT (121.21691 31.26967)	0	1

transbigdata.grid_to_params(grid)

A useful tool to get grid params from grid geometry

[13]:
```
# this is the formal grid geometry
grid_rec.head()
```

[13]:

	LONCOL	LATCOL	geometry
171	9	0	POLYGON ((121.18270 31.24943, 121.18797 31.249...
174	10	0	POLYGON ((121.18797 31.24943, 121.19323 31.249...

(continues on next page)

(continued from previous page)

```
177      11       0  POLYGON ((121.19323 31.24943, 121.19849 31.249...
180      12       0  POLYGON ((121.19849 31.24943, 121.20375 31.249...
183      13       0  POLYGON ((121.20375 31.24943, 121.20902 31.249...
```

[14]:
```
tbd.grid_to_params(grid_rec)
```

[14]:
```
{'slon': 121.13797109957761,
 'slat': 31.25168182840957,
 'deltalon': 0.005262604988999442,
 'deltalat': 0.004496605206419705,
 'theta': 0,
 'method': 'rect'}
```

transbigdata.grid_params_optimize(data, initialparams, col=['uid', 'lon', 'lat'], optmethod='centerdist', printlog=False, sample=0)

Offers several methods to optimize the grids

This method relies on the `scikit-opt` package. To do so, please run following code in cmd:

```
pip install scikit-opt
```

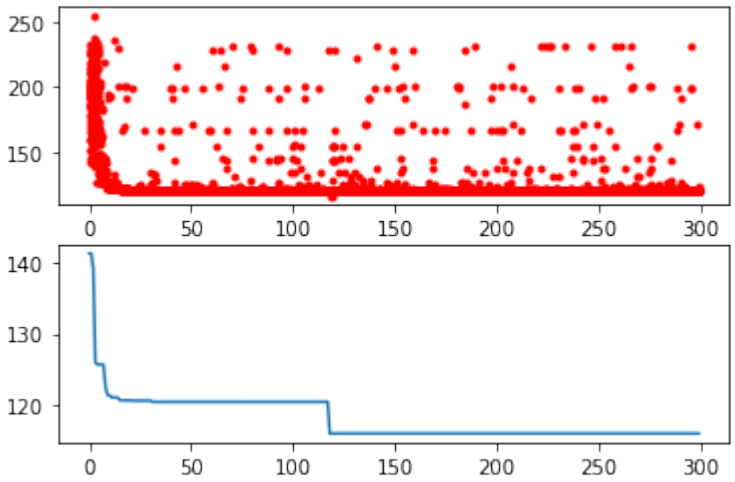
For more details of this method, please refer to this notebook.

[15]:
```
# we use the random generated data again
gps_random.head()
```

[15]:
```
   veh_id         lon        lat  LonCol  LatCol  LonGridCenter  \
0       0  121.204726  31.266296      13       3     121.206385
1       1  121.168077  31.326952       6      17     121.169547
2       2  121.142706  31.315498       1      14     121.143234
3       3  121.215899  31.339561      15      20     121.216910
4       4  121.217937  31.269540      15       4     121.216910

   LatGridCenter                                    grid_geo_polygon
0      31.265172  POLYGON ((121.2037536619401 31.262923341425626...
1      31.328124  POLYGON ((121.16691542701707 31.32587581431555...
2      31.314634  POLYGON ((121.14060240207206 31.31238599869628...
3      31.341614  POLYGON ((121.2142788719181 31.339365629934818...
4      31.269668  POLYGON ((121.2142788719181 31.26741994663205,...
```

[16]:
```
tbd.grid_params_optimize(gps_random, params_rec, col=['veh_id', 'lon', 'lat'],
↪printlog=True)
```

```
Optimized index centerdist: 116.11243965546235
Optimized gridding params: {'slon': 121.14169760115118, 'slat': 31.252579076220087,
↪'deltalon': 0.005262604989003139, 'deltalat': 0.004496605206422906, 'theta': 50.
↪91831009508256, 'method': 'rect'}
Optimizing cost:
```



Result:

[16]:
```
{'slon': 121.14169760115118,
 'slat': 31.252579076220087,
 'deltalon': 0.005262604989003139,
```

(continues on next page)

(continued from previous page)

```
 'deltalat': 0.004496605206422906,
 'theta': 50.91831009508256,
 'method': 'rect'}
```

3 Trajectory processing using TransBigData

For vehicle trajectory data, the TransBigData library provides a comprehensive set of trajectory data processing methods starting from version 0.5.0 and above. These methods include preprocessing and drift correction of trajectory data, segmentation of stops and trips, grid-based representation, visualization, and more. This article will introduce how to use the TransBigData library for processing trajectory data.

[1]:
```
import pandas as pd
import geopandas as gpd
import transbigdata as tbd
# ensure tbd version is above 0.5.0
tbd.__version__
```

[1]:
```
'0.5.0'
```

Trajectory Quality

First, we read the data and observe the basic information to check for any missing values. Using the built-in methods of a DataFrame, we can easily view the basic information of the data, including data types, number of fields, number of rows, and the presence of missing values. The code is as follows:

[2]:
```
# Read the data
data = pd.read_csv('data/TaxiData-Sample.csv', header=None)
data.columns = ['id', 'time', 'lon', 'lat', 'OpenStatus', 'speed']
# Convert the time format
data['time'] = pd.to_datetime(data['time'])
data
```

[2]:
```
           id                time         lon        lat  OpenStatus  speed
0       34745 2023-05-29 20:27:43  113.806847  22.623249           1     27
1       34745 2023-05-29 20:24:07  113.809898  22.627399           0      0
2       34745 2023-05-29 20:24:27  113.809898  22.627399           0      0
3       34745 2023-05-29 20:22:07  113.811348  22.628067           0      0
4       34745 2023-05-29 20:10:06  113.819885  22.647800           0     54
...       ...                 ...         ...        ...         ...    ...
544994  28265 2023-05-29 21:35:13  114.321503  22.709499           0     18
544995  28265 2023-05-29 09:08:02  114.322701  22.681700           0      0
544996  28265 2023-05-29 09:14:31  114.336700  22.690100           0      0
544997  28265 2023-05-29 21:19:12  114.352600  22.728399           0      0
544998  28265 2023-05-29 19:08:06  114.137703  22.621700           0      0

[544999 rows x 6 columns]
```

[3]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 544999 entries, 0 to 544998
```

(continues on next page)

(continued from previous page)

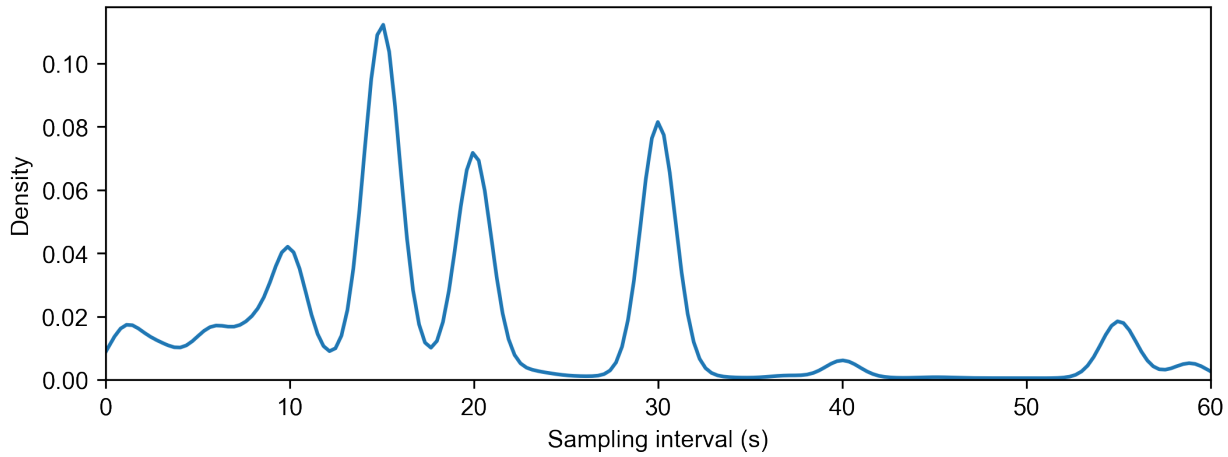
```
Data columns (total 6 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   id          544999 non-null  int64
 1   time        544999 non-null  datetime64[ns]
 2   lon         544999 non-null  float64
 3   lat         544999 non-null  float64
 4   OpenStatus  544999 non-null  int64
 5   speed       544999 non-null  int64
dtypes: datetime64[ns](1), float64(2), int64(3)
memory usage: 24.9 MB
```

In it, the data types of the data fields, the number of non-null values, and the memory usage are listed. In the "Non-Null" column, the number of non-null values for each field is listed. If the number of non-null values for a field is less than the total number of rows, it indicates the presence of missing values in that field.

Next, we will use TransBigData to generate a data quality report and observe the number of vehicles in the data, the observation time period, and the sampling interval:

[4]:
```
# Generate data quality report
tbd.data_summary(data,col=['id', 'time'],show_sample_duration=True)
```

```
Amount of data
-----------------
Total number of data items:  544999
Total number of individuals:  180
Data volume of individuals(Mean):  3027.7722
Data volume of individuals(Upper quartile):  4056.25
Data volume of individuals(Median):  2600.5
Data volume of individuals(Lower quartile):  1595.75

Data time period
-----------------
Start time:  2023-05-29 00:00:00
End time:  2023-05-29 23:59:59

Sampling interval
-----------------
Mean:  27.995 s
Upper quartile:  30.0 s
Median:  20.0 s
Lower quartile:  15.0 s
```



Redundancy Elimination

Redundancy elimination is an important step in cleaning trajectory data. It reduces the data volume and improves data processing efficiency without affecting the information contained in the data. In practical trajectory data processing, you may encounter the following two types of redundancy:

1. **Redundancy of Duplicate Data at the Same Moment**:

In a trajectory dataset, there may be multiple trajectory data entries for the same vehicle at the same moment. This can happen when the sampling interval is too short compared to the precision of the time field in the dataset. For example, if the sampling interval is 1 second, but the time field in the dataset has a precision of 1 minute, it can result in multiple trajectory data entries for the same vehicle within the same minute. Not removing these redundant data can cause difficulties in subsequent processing. The method to eliminate this redundancy is straightforward: keep only one trajectory data entry for the same vehicle at the same moment.

2. **Redundancy of Vehicle Stops**:

In vehicle trajectory data, the sampling interval is usually very short, such as collecting data every few seconds. This means that data is continuously generated for vehicles, whether they are moving or stationary. In practical applications, the focus is often on the trajectory of vehicles during trips, rather than during stops. For each instance of a vehicle stopping, we only need to know the start and end times of the stop. The data generated during the middle part of the stop, at the same location, is redundant and can be removed to reduce the overall data size. For a sequence of consecutive n data entries (n>=3) with the same location, we only need to keep the first and last entries, as the intermediate data is redundant. In the code, it is sufficient to compare the vehicle ID and the latitude-longitude of each data entry with the previous and next trajectory entries. If they are the same, the data can be removed.

However, the redundancy elimination method mentioned above for vehicle stops does not consider the information carried by fields other than the vehicle ID, time, and latitude-longitude. For example, in the case of taxi vehicles, passengers may board the taxi during a stop, changing the status from “vacant” to “occupied.” In such cases, this information needs to be preserved.

TransBigData provides a function, `tbd.traj_clean_redundant()`, for trajectory data redundancy elimination. It can handle the mentioned redundancy situations and can also detect redundancy in fields other than the vehicle ID and latitude-longitude. The code is as follows:

[5]:
```
# data volume before Redundancy Elimination
len(data)
```

[5]:
```
544999
```

[6]:
```
# Data redundancy removal to reduce data size and improve computational efficiency in␣
↪subsequent steps
#
data = tbd.traj_clean_redundant(
    data,
    col=['id', 'time', 'lon', 'lat', 'speed']  # Apart from vehicle ID, time, longitude,␣
↪and latitude, consider whether the speed field has redundancy
)
len(data)
```

[6]:
```
421099
```

The code snippet performs data redundancy removal to reduce the data size and improve computational efficiency. The `traj_clean_redundant` function from the `tbd` module is used for this purpose. The function takes the `data` variable as input and specifies the columns (`col`) to consider for redundancy removal, including the vehicle ID, time, longitude, latitude, and optionally the speed field. The result is stored back in the `data` variable, and the length of the updated data is outputted.

Drift Cleaning

In vehicle trajectory data, deviations and errors may occur between the collected vehicle trajectory data and the actual situations due to factors such as errors in data collection devices, environmental interferences, device malfunctions, unstable GPS signals, insufficient satellite coverage, and signal obstructions. This results in a discrepancy between the actual positions and the collected positions of the trajectory data, known as data drift in vehicle trajectory data. In the data, data drift is manifested by significant distances between the trajectory data points and the actual positions, often accompanied by sudden jumps. This drift can affect subsequent spatial analysis and spatial statistics, requiring the cleaning of vehicle trajectory data to ensure data accuracy and usability.

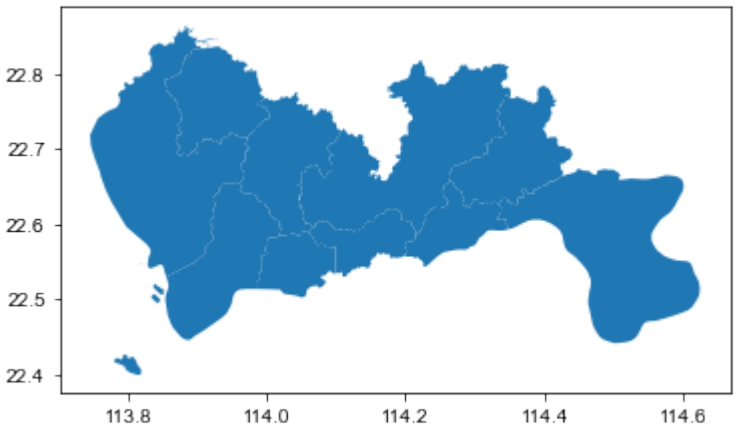
To clean vehicle trajectory data for drift, one approach is to remove trajectory data points outside the defined study area. The study area can be defined in two ways: either by specifying the bottom-left and top-right coordinates to determine the boundary range (bounds) or by using a geographic information file (geojson or shapefile) that represents the study area.

To remove drift data using a geojson or shapefile, you first need to read the geojson or shapefile as a `GeoDataFrame` type in GeoPandas. Then, you can use the `intersects()` method provided by GeoPandas to determine if the trajectory data is within the study area. However, this method requires performing spatial geometry matching for each trajectory data point, which can be time-consuming for large datasets. The `tbd.clean_outofshape()` method provided in the TransBigData package offers a more efficient approach. It first maps the trajectory data to the corresponding grid using built-in grid partitioning methods and then performs spatial matching based on the grid, significantly improving the cleaning efficiency.

Here is an example code snippet using TransBigData's `clean_outofshape()` method for data drift cleaning:

[7]:
```
# Read the research area range
sz = gpd.read_file('Data/sz.json')
sz.plot()
```

[7]:
```
<AxesSubplot:>
```



[8]:
```
# Data drift removal
# Removing data outside the study area
data = tbd.clean_outofshape(data, sz, col=['lon', 'lat'], accuracy=500)
len(data)
```

[8]:
```
419448
```

Cleaning trajectory drift within the study area requires assessing and cleaning based on the continuous changes in trajectories. There are three common approaches for cleaning:

1. Velocity threshold method: If the velocity between the current trajectory data and the previous and subsequent trajectories exceeds a threshold, it is considered drift.
2. Distance threshold method: If the distance between the current trajectory data and the previous and subsequent trajectories exceeds a threshold, it is considered drift.
3. Angle threshold method: If the angle formed by the previous, current, and subsequent trajectories is smaller than a threshold, it is considered drift.

In TransBigData, the `tbd.traj_clean_drift()` method is provided to clean trajectory data for multiple vehicles. This method integrates distance, velocity, and angle thresholds into a single function.

Here is an example code snippet using TransBigData's `traj_clean_drift()` method:

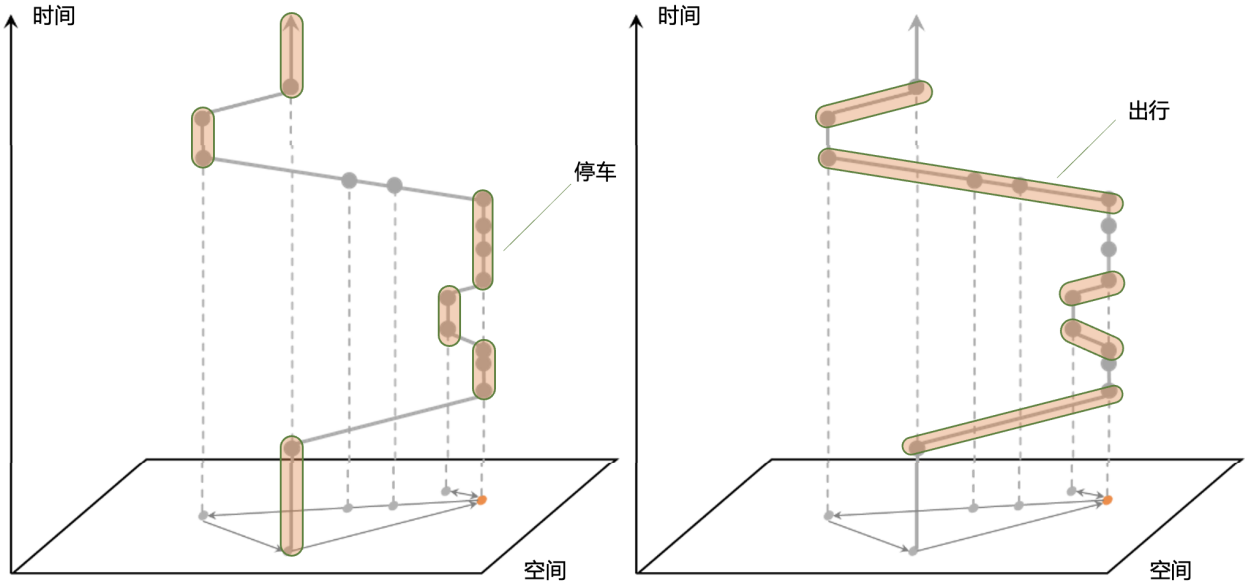
[9]:
```
# Drift cleaning within the study area using speed, distance, and angle as criteria
data = tbd.traj_clean_drift(
    data,  # Trajectory data, can include data for multiple vehicles, distinguished by ID
    col=['id', 'time', 'lon', 'lat'],  # Column names of the trajectory data
    speedlimit=80,  # Speed threshold in km/h, set to None to skip speed-based filtering
    dislimit=4000,  # Distance threshold in meters, set to None to skip distance-based␣
↪filtering
    anglelimit=30)  # Angle threshold in degrees, set to None to skip angle-based␣
↪filtering
len(data)
```

[9]:
```
405286
```

Stops and Trips Extracting

In long-term continuous observations of vehicle trajectory data, a common requirement is to extract the stops and trips from the trajectory data. The stops can be analyzed to determine the duration and location of the vehicle's parking, while the trips can be further analyzed to extract information such as the origin and destination of each trip, the travel path, travel duration, travel distance, and travel speed. In this section, we will explain how to identify stops and trips from vehicle trajectories, extract the trajectory information for each trip, and generate trajectory lines.

In vehicle trajectory data, stops and trips are typically identified using a time threshold method. The approach is as follows: to avoid fluctuations in the trajectory data, we need to predefine a grid in the geographical space. If the duration between two consecutive data points exceeds our set threshold (usually 30 minutes), we consider it as a stop. The time period between two stops is considered a trip, as shown in the figure below.



After cleaning the trajectory data, we need to define a grid coordinate system and convert the trajectory data into grid-based representation to identify stops and trips.

[11]:
```
# Define the grid parameters
bounds = [113.75, 22.4, 114.62, 22.86]
params = tbd.area_to_params(bounds,accuracy = 100)
params
```

[11]:
```
{'slon': 113.75,
 'slat': 22.4,
 'deltalon': 0.0009743362892898221,
 'deltalat': 0.0008993210412845813,
 'theta': 0,
 'method': 'rect',
 'gridsize': 100}
```

The identification of stops and trips has been provided in the TransBigData package through the `tbd.traj_stay_move()` function. Here is the code:

[12]:
```
# Identify stay and move
stay, move = tbd.traj_stay_move(data, params, col=['id', 'time', 'lon', 'lat'],␣
↪activitytime=1800)
len(stay), len(move)
```

[12]:
```
(545, 725)
```

In the code, `data` represents the cleaned trajectory data. The `traj_stay_move()` function is used to identify stops and trips based on the specified stop threshold. It returns two outputs: `stops` and `trips`, which contain the identified stops and trips, respectively. You can utilize these data for further analysis.

Note: The `tbd.traj_stay_move()` function does not remove trips with a duration of 0. This is because some trajectory data may have long sampling intervals, which may not capture the travel process between two stops. As a result, the calculated duration for those trips would be 0.

[13]:
```
stay
```

[13]:
```
           id               stime  LONCOL  LATCOL               etime  \
23320   22396 2023-05-29 06:10:09     262     284 2023-05-29 07:56:32
51327   22396 2023-05-29 12:51:23      50     429 2023-05-29 16:44:57
54670   22413 2023-05-29 00:00:09     145     431 2023-05-29 01:22:41
54722   22413 2023-05-29 01:25:13     145     431 2023-05-29 02:01:44
54741   22413 2023-05-29 02:18:13     145     431 2023-05-29 02:48:46
...       ...                 ...     ...     ...                 ...
151140  36686 2023-05-29 01:04:37     323     172 2023-05-29 02:33:37
154762  36686 2023-05-29 03:03:07     307     185 2023-05-29 03:51:07
154924  36686 2023-05-29 04:21:07     307     185 2023-05-29 05:06:07
307421  36805 2023-05-29 03:15:54     324     168 2023-05-29 04:14:40
337431  36805 2023-05-29 04:23:40     325     132 2023-05-29 05:13:33

                lon        lat  duration  stayid
23320   114.005547  22.655800    6383.0       0
51327   113.798630  22.786167   14014.0       1
54670   113.891403  22.787300    4952.0       2
54722   113.891701  22.787399    2191.0       3
54741   113.891502  22.787300    1833.0       4
...            ...        ...       ...     ...
151140  114.065193  22.554672    5340.0     540
154762  114.048721  22.566692    2880.0     541
154924  114.048676  22.566605    2700.0     542
307421  114.065552  22.551100    3526.0     543
337431  114.066521  22.519133    2993.0     544

[545 rows x 9 columns]
```

[14]:
```
move
```

[14]:
```
           id  SLONCOL  SLATCOL               stime        slon       slat  \
0       22396      253      326 2023-05-29 00:00:29  113.996719  22.693333
23320   22396      262      284 2023-05-29 07:56:32  114.005547  22.655800
51327   22396       50      429 2023-05-29 16:44:57  113.798630  22.786167
54670   22413      145      431 2023-05-29 00:00:09  113.891403  22.787300
54670   22413      145      431 2023-05-29 01:22:41  113.891403  22.787300
...       ...      ...      ...                 ...         ...        ...
154762  36686      307      185 2023-05-29 03:51:07  114.048721  22.566692
154924  36686      307      185 2023-05-29 05:06:07  114.048676  22.566605
135725  36805      328      147 2023-05-29 00:00:03  114.070030  22.531967
307421  36805      324      168 2023-05-29 04:14:40  114.065552  22.551100
337431  36805      325      132 2023-05-29 05:13:33  114.066521  22.519133
```

(continues on next page)

(continued from previous page)

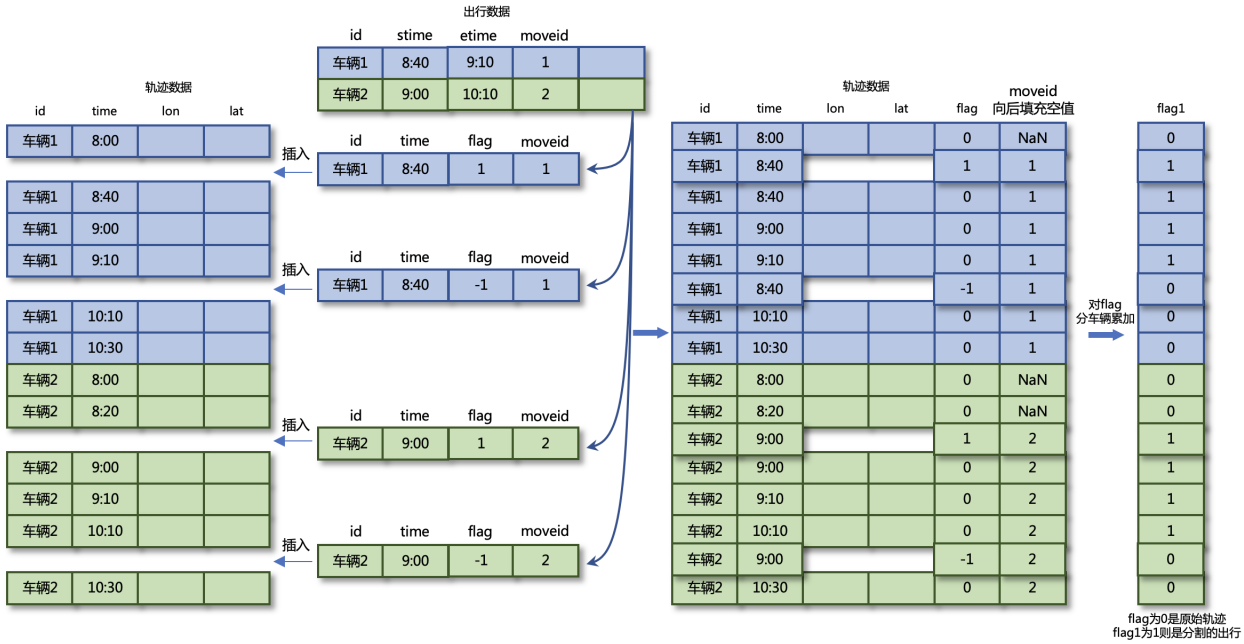
```
                    etime        elon       elat  ELONCOL  ELATCOL  duration  \
0      2023-05-29 06:10:09  114.005547  22.655800    262.0    284.0   22180.0
23320  2023-05-29 12:51:23  113.798630  22.786167     50.0    429.0   17691.0
51327  2023-05-29 23:59:55  114.025253  22.654900    283.0    283.0   26098.0
54670  2023-05-29 00:00:09  113.891403  22.787300    145.0    431.0       0.0
54670  2023-05-29 01:25:13  113.891701  22.787399    145.0    431.0     152.0
...                    ...         ...        ...      ...      ...       ...
154762 2023-05-29 04:21:07  114.048676  22.566605    307.0    185.0    1800.0
154924 2023-05-29 23:53:46  114.124580  22.571978    384.0    191.0   67659.0
135725 2023-05-29 03:15:54  114.065552  22.551100    324.0    168.0   11751.0
307421 2023-05-29 04:23:40  114.066521  22.519133    325.0    132.0     540.0
337431 2023-05-29 23:53:51  114.120354  22.544300    380.0    160.0   67218.0

        moveid
0            0
23320        1
51327        2
54670        3
54670        4
...        ...
154762     720
154924     721
135725     722
307421     723
337431     724

[725 rows x 13 columns]
```

Based on the stops and trips information, you can slice the trajectory data to extract the trajectory points during stops and trips. Since you have already performed redundancy elimination earlier, the redundant trajectory points during stops have been removed. As a result, the trajectory data during stops will have significantly fewer data points compared to the trajectory data during trips.

Trajectory slicing

By the preceding code, parking and travel information has been successfully extracted from the data. However, in the obtained travel information, only the time, latitude, and longitude information of the starting and ending points of each trip are included, without the travel trajectory information. In order to further analyze the travel trajectories of vehicles, it is necessary to extract the trajectory data from the time periods of each trip, that is, slice the trajectory dataset based on the travel information. In the previously calculated travel information, each travel record has a travel ID, start time, and end time columns. The result of trajectory slicing is to extract the trajectory points during the trip and assign a travel ID label to each trajectory point.

The idea of trajectory slicing is illustrated in the diagram below. A flag column is created for the trajectory data to mark whether each row is within the desired time period for slicing. Then, each travel record in the travel data is decomposed into a start time record (flag label of 1) and an end time record (flag label of -1), which are inserted into the trajectory data. Next, the flag column is grouped and summed by vehicle to obtain the flag1 column. In the result, the rows where the flag1 column value is 1 (travel) and the flag column value is 0 (non-temporarily inserted data) are the desired trajectory data.



The code for slicing the trajectory data can be performed by using `tbd.traj_slice`, the code example is as follows:

[18]:
```
# Extract trajectory points during parking
stay_points = tbd.traj_slice(data, stay, traj_col=['id', 'time'], slice_col=[
                             'id', 'stime', 'etime', 'stayid'])
stay_points
```

[18]:
```
          id                time         lon        lat  OpenStatus  speed  \
23320   22396 2023-05-29 06:10:09  114.005547  22.655800         0.0    1.0
23321   22396 2023-05-29 06:20:42  114.005547  22.655800         0.0    0.0
23322   22396 2023-05-29 06:25:09  114.005417  22.655767         0.0    0.0
23323   22396 2023-05-29 06:25:17  114.005417  22.655767         0.0    0.0
23324   22396 2023-05-29 06:30:09  114.005402  22.655767         0.0    0.0
...       ...                 ...         ...        ...         ...    ...
307425  36805 2023-05-29 04:14:40  114.066452  22.550966         0.0    9.0
337431  36805 2023-05-29 04:23:40  114.066521  22.519133         0.0   20.0
337432  36805 2023-05-29 04:23:55  114.066521  22.519133         0.0    0.0
337433  36805 2023-05-29 05:13:27  114.066521  22.519133         1.0    0.0
337434  36805 2023-05-29 05:13:33  114.067230  22.519751         1.0    3.0

        stayid
23320      0.0
23321      0.0
23322      0.0
23323      0.0
23324      0.0
...        ...
307425   543.0
337431   544.0
337432   544.0
337433   544.0
337434   544.0
```

(continues on next page)

(continued from previous page)

```
[8428 rows x 7 columns]
```

[19]:
```
# Extract trajectory points during travel
move_points = tbd.traj_slice(data, move, traj_col=['id', 'time'], slice_col=[
                             'id', 'stime', 'etime', 'moveid'])
move_points
```

[19]:
```
           id                time         lon        lat  OpenStatus  speed  \
0       22396 2023-05-29 00:00:29  113.996719  22.693333         1.0   20.0
27      22396 2023-05-29 00:01:01  113.995514  22.695032         1.0   34.0
28      22396 2023-05-29 00:01:09  113.995430  22.695766         1.0   41.0
29      22396 2023-05-29 00:01:41  113.995369  22.696484         1.0    0.0
30      22396 2023-05-29 00:02:21  113.995430  22.696650         1.0   17.0
...       ...                 ...         ...        ...         ...    ...
132774  36805 2023-05-29 23:53:03  114.120354  22.544333         1.0    3.0
132775  36805 2023-05-29 23:53:09  114.120354  22.544300         1.0    2.0
132776  36805 2023-05-29 23:53:15  114.120354  22.544300         1.0    1.0
132777  36805 2023-05-29 23:53:21  114.120354  22.544300         1.0    0.0
132778  36805 2023-05-29 23:53:51  114.120354  22.544300         0.0    0.0

        moveid
0          0.0
27         0.0
28         0.0
29         0.0
30         0.0
...        ...
132774   724.0
132775   724.0
132776   724.0
132777   724.0
132778   724.0

[397892 rows x 7 columns]
```

Trajectory densification and sparsification

To facilitate subsequent tasks such as matching the travel paths in the trajectory data with the road network, we perform densification or sparsification of the trajectory points during the travel process. At this stage, we specify the ID column as the travel ID (moveid) column, meaning that when performing the densification or sparsification operation on the trajectory points, we consider each travel separately. The code for this operation is as follows:

[59]:
```
# Trajectory densification
move_points_densified = tbd.traj_densify(
    move_points, col=['moveid', 'time', 'lon', 'lat'], timegap=15)
move_points_densified
```

[59]:
```
             id                time         lon        lat  OpenStatus  \
0       22396.0 2023-05-29 00:00:29  113.996719  22.693333         1.0
2           NaN 2023-05-29 00:00:30  113.996681  22.693386         NaN
3           NaN 2023-05-29 00:00:45  113.996116  22.694183         NaN
```

(continues on next page)

(continued from previous page)

```
4            NaN 2023-05-29 00:01:00  113.995552  22.694979         NaN
1        22396.0 2023-05-29 00:01:01  113.995514  22.695032         1.0
...          ...                 ...         ...        ...         ...
397889   36805.0 2023-05-29 23:53:15  114.120354  22.544300         1.0
397890   36805.0 2023-05-29 23:53:21  114.120354  22.544300         1.0
4175974      NaN 2023-05-29 23:53:30  114.120354  22.544300         NaN
4175975      NaN 2023-05-29 23:53:45  114.120354  22.544300         NaN
397891   36805.0 2023-05-29 23:53:51  114.120354  22.544300         0.0

         speed  moveid
0         20.0     0.0
2          NaN     0.0
3          NaN     0.0
4          NaN     0.0
1         34.0     0.0
...        ...     ...
397889     1.0   724.0
397890     0.0   724.0
4175974    NaN   724.0
4175975    NaN   724.0
397891     0.0   724.0

[1211070 rows x 7 columns]
```

[60]:
```
# Trajectory sparsification
move_points_sparsified = tbd.traj_sparsify(
    move_points, col=['moveid', 'time', 'lon', 'lat'],  timegap=120)
move_points_sparsified
```

[60]:
```
           id                time         lon        lat  OpenStatus  speed  \
0       22396 2023-05-29 00:00:29  113.996719  22.693333         1.0   20.0
4       22396 2023-05-29 00:02:21  113.995430  22.696650         1.0   17.0
7       22396 2023-05-29 00:04:21  113.992348  22.696733         0.0   36.0
10      22396 2023-05-29 00:06:21  113.986366  22.691000         0.0   48.0
12      22396 2023-05-29 00:08:21  113.989586  22.681749         0.0   43.0
...       ...                 ...         ...        ...         ...    ...
397822  36805 2023-05-29 23:45:27  114.091217  22.540768         0.0   11.0
397835  36805 2023-05-29 23:47:21  114.093002  22.543383         1.0    0.0
397855  36805 2023-05-29 23:49:21  114.105850  22.545250         1.0   58.0
397875  36805 2023-05-29 23:51:21  114.119514  22.547033         1.0   24.0
397890  36805 2023-05-29 23:53:21  114.120354  22.544300         1.0    0.0

        moveid
0          0.0
4          0.0
7          0.0
10         0.0
12         0.0
...        ...
397822   724.0
397835   724.0
397855   724.0
```

(continues on next page)

(continued from previous page)

```
397875   724.0
397890   724.0

[90172 rows x 7 columns]
```

[ ]:
```
# define a function to plot the trajectory
def plot_traj(traj):
    import folium
    # 1. Create a map with the center at the average coordinates of the trajectory
    m = folium.Map(location=[traj['lat'].mean(), traj['lon'].mean()],  # Map center
                zoom_start=14,  # Map zoom level
                tiles='cartodbpositron')  # Map style
    # 2. Add the trajectory
    folium.PolyLine(
        traj[['lat', 'lon']].values.tolist(),  # Trajectory coordinates
        color='red',  # Trajectory color
        weight=2.5,  # Trajectory width
        opacity=1).add_to(m)  # Trajectory opacity, add to the map after creating the␣
↪trajectory
    # 3. Add trajectory points
    for i in range(len(traj)):
        folium.CircleMarker(
            location=[traj['lat'].iloc[i], traj['lon'].iloc[i]],  # Trajectory point␣
↪coordinates
            radius=3,  # Trajectory point radius
            color='red',  # Trajectory point color
        ).add_to(m)  # Fill opacity, add to the map after creating the trajectory point
    # 4. Add start and end markers
    folium.Marker([traj['lat'].iloc[0], traj['lon'].iloc[0]],  # Start point coordinates
                popup='Start',  # Start marker label
                icon=folium.Icon(color='green')).add_to(m)  # Start marker color
    folium.Marker([traj['lat'].iloc[-1], traj['lon'].iloc[-1]],
                popup='End',
                icon=folium.Icon(color='red')).add_to(m)
    # 5. Display the map, directly in Jupyter Notebook
    return m
```

[65]:
```
moveid = 51
# Original trajectory
traj = move_points[move_points['moveid']==moveid]
plot_traj(traj)
```

[65]:
```
<folium.folium.Map at 0x2ae9ffa90>
```

[66]:
```
# Densified trajectory
traj = move_points_densified[move_points_densified['moveid']==moveid]
plot_traj(traj)
```

[66]:
```
<folium.folium.Map at 0x2af4fc970>
```

[67]:
```
# Sparsified trajectory
traj = move_points_sparsified[move_points_sparsified['moveid']==moveid]
plot_traj(traj)
```

[67]:
```
<folium.folium.Map at 0x2ae4a8f40>
```

Trajectory smoothing

When processing vehicle trajectory data, the trajectory points represent "observations" of the actual "state" of the vehicle. Due to errors, the observed data may deviate from the actual state of the vehicle.

How can we obtain a more accurate estimation of the actual state of the vehicle? Consider the method mentioned in the previous section for detecting trajectory drift, which involves comparing the position of a trajectory point with the position of previous trajectory points to check for significant and unreasonable jumps. This approach is essentially based on predicting the possible future positions of the vehicle based on its previous trajectory. If the next recorded trajectory point deviates significantly from the expected position, it can be determined that the trajectory is abnormal.

This method shares similarities with the concept of Kalman filtering. Kalman filtering is a linear quadratic estimation algorithm used for state estimation in linear dynamic systems. It combines the previous state estimation (i.e., the predicted position of the current trajectory point) with the current observation data (recorded position of the current trajectory point) to obtain an optimal estimate of the current state.

The implementation of Kalman filtering involves predicting the current value using the previous optimal result and then correcting the current value using the observed value to obtain the optimal result. This method effectively reduces the impact of noise, allowing for a more accurate estimation of the actual state of the vehicle.

[84]:
```
move_id =51
traj = move_points[move_points['moveid'] == move_id].copy()
traj_smoothed = tbd.traj_smooth(traj,col = ['id','time','lon', 'lat'],proj=False,process_
↪noise_std = 0.01, measurement_noise_std = 1)
```

[85]:
```
# plot the trajectory
import folium
m = folium.Map(location=[traj['lat'].mean(), traj['lon'].mean()],
               zoom_start=14,
               tiles='cartodbpositron')
# original trajectory
folium.PolyLine(
    traj[['lat', 'lon']].values.tolist(),
    color='red',
    weight=2.5,
    opacity=1).add_to(m)
# smoothed trajectory
folium.PolyLine(
    traj_smoothed[['lat', 'lon']].values.tolist(),
    color='blue',
    weight=2.5,
    opacity=1).add_to(m)
m
```

[85]:
```
<folium.folium.Map at 0x2af969340>
```

The goal of the Kalman filter is to optimize the observed values by estimating the system state while considering the uncertainties of both the observation noise and the system dynamics. It has advantages in smoothing trajectory data by

reducing noise effects and minimizing fluctuations within a small range. However, the Kalman filter cannot completely eliminate all noise or handle trajectory drift.

The Kalman filter is suitable for cases where the noise in the trajectory data is relatively stable, meaning the noise variance remains constant. It is particularly effective in smoothing small-scale fluctuations caused by measurement errors in trajectory data.

However, the effectiveness of the Kalman filter is limited when significant drift occurs in the trajectory. Drift points are treated as observations and have a significant impact on state estimation, which the Kalman filter cannot directly handle.

Furthermore, the Kalman filter requires the specification of covariance matrices for process and observation errors, and these parameter settings affect the smoothing effect. Improper covariance matrix settings can result in significant deviations in the smoothed trajectory data, especially when dealing with trajectory data that may deviate from the road network.

When processing trajectory data, a common approach is to first remove drift, then perform smoothing, and finally conduct road network matching. The rationale behind this approach is as follows:

- The drift removal step eliminates obvious drift points in the data, which are large noise components. The presence of drift points can significantly interfere with subsequent processing steps. Removing or correcting drift points ensures the accuracy and reliability of subsequent processing.
- After drift removal, the trajectory data may still contain some noise and fluctuations. To reduce the impact of these noise and fluctuations, smoothing can be applied to further process the trajectory data, making it smoother, more continuous, and maintaining the overall trend of the trajectory.
- Finally, the smoothed trajectory is more stable and better suited for road network matching. It reduces errors caused by noise and fluctuations, thereby improving the accuracy and reliability of road network matching.

4 Generate and rotate triangle and hexagon grids

In this example, we will introduce more options for `TransBigData` grid processing, including: - Adding rotation angle. - Triangle and hexagon grids.

Rotate the grids

[1]:
```
#Read taxi gps data
import transbigdata as tbd
import pandas as pd
data = pd.read_csv('data/TaxiData-Sample.csv',header = None)
data.columns = ['VehicleNum','time','lon','lat','OpenStatus','Speed']
#Define the study area
bounds = [113.75, 22.4, 114.62, 22.86]
#Delete the data out of the study area
data = tbd.clean_outofbounds(data,bounds = bounds,col = ['lon','lat'])
```

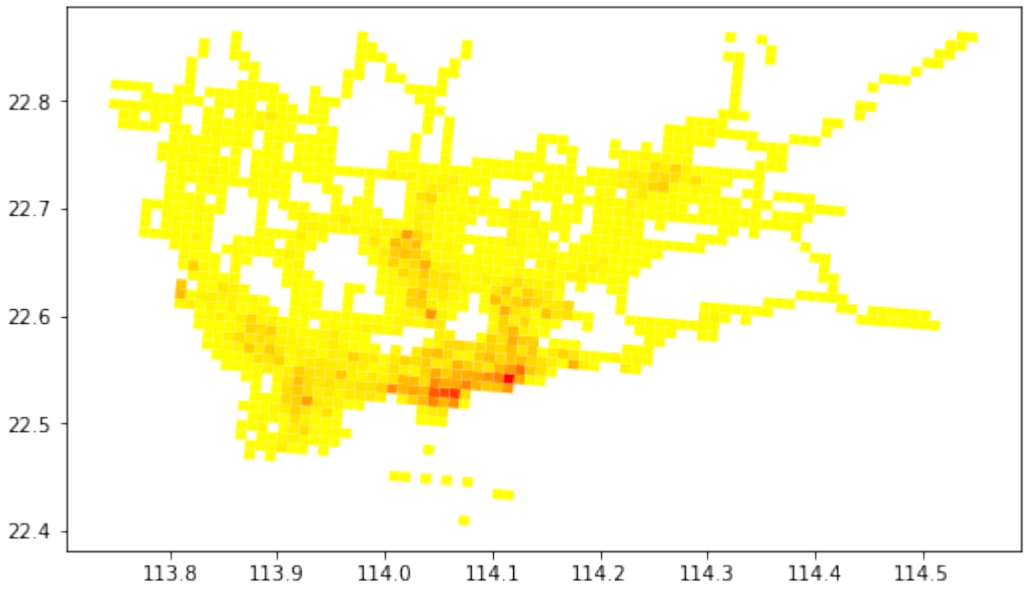
The grid coordinates system offer by `TransBigData` also support adding rotation angle for the grids.
you can also specify a rotation angle for the grids by adding the `theta` into gridding params:

[2]:
```
#Obtain the gridding parameters
params = tbd.area_to_params(bounds,accuracy = 1000)
#Add a rotation angle
params['theta'] = 5
print(params)
```

```
{'slon': 113.75, 'slat': 22.4, 'deltalon': 0.009743362892898 22, 'deltalat': 0.
↪008993210412845813, 'theta': 5, 'method': 'rect', 'gridsize': 1000}
```

[3]:
```
#Map the GPS data to grids
data['LONCOL'],data['LATCOL'] = tbd.GPS_to_grid(data['lon'],data['lat'],params)
#Aggregate data into grids
grid_agg = data.groupby(['LONCOL','LATCOL'])['VehicleNum'].count().reset_index()
#Generate grid geometry
grid_agg['geometry'] = tbd.grid_to_polygon([grid_agg['LONCOL'],grid_agg['LATCOL']],
↪params)
#Change the type into GeoDataFrame
import geopandas as gpd
grid_agg = gpd.GeoDataFrame(grid_agg)
#Plot the grids
grid_agg.plot(column = 'VehicleNum',cmap = 'autumn_r',figsize=(10,5))
```

[3]:
```
<AxesSubplot:>
```

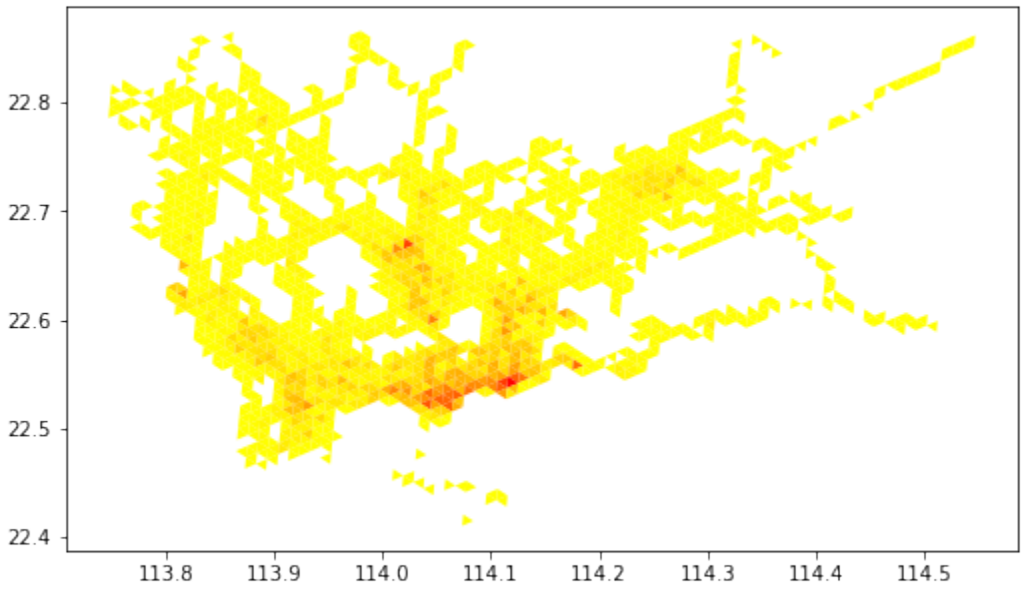


Triangle and Hexagon grids

[4]:
```
#Triangle grids
params['method'] = 'tri'
```

[5]:
```
#Map the GPS data to grids
data['loncol_1'],data['loncol_2'],data['loncol_3'] = tbd.GPS_to_grid(data['lon'],data[
↪'lat'],params)
#Aggregate data into grids
grid_agg = data.groupby(['loncol_1','loncol_2','loncol_3'])['VehicleNum'].count().reset_
↪index()
#Generate grid geometry
grid_agg['geometry'] = tbd.grid_to_polygon([grid_agg['loncol_1'],grid_agg['loncol_2'],
↪grid_agg['loncol_3']],params)
#Change the type into GeoDataFrame
import geopandas as gpd
grid_agg = gpd.GeoDataFrame(grid_agg)
#Plot the grids
grid_agg.plot(column = 'VehicleNum',cmap = 'autumn_r',figsize=(10,5))
```

[5]:
```
<AxesSubplot:>
```



[6]:
```
#Hexagon grids
params['method'] = 'hexa'
```

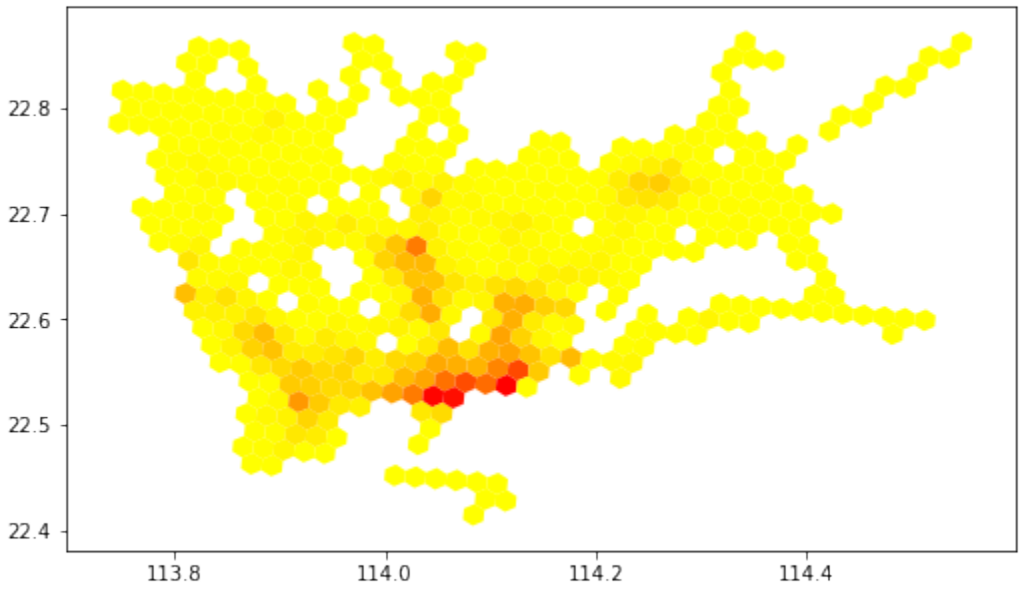
[7]:
```
#Map the GPS data to grids
data['loncol_1'],data['loncol_2'],data['loncol_3'] = tbd.GPS_to_grid(data['lon'],data[
↪'lat'],params)
#Aggregate data into grids
grid_agg = data.groupby(['loncol_1','loncol_2','loncol_3'])['VehicleNum'].count().reset_
↪index()
#Generate grid geometry
grid_agg['geometry'] = tbd.grid_to_polygon([grid_agg['loncol_1'],grid_agg['loncol_2'],
```

(continues on next page)

(continued from previous page)

```
↪grid_agg['loncol_3']],params)
#Change the type into GeoDataFrame
import geopandas as gpd
grid_agg = gpd.GeoDataFrame(grid_agg)
#Plot the grids
grid_agg.plot(column = 'VehicleNum',cmap = 'autumn_r',figsize=(10,5))
```

[7]:
```
<AxesSubplot:>
```



5 Optimize gridding params

Why aggregate data to grids?

Why do we aggregate data to grids?

Discretization

Hard to analyze data in continuous space, but easy to analyze with discretized region. Defining spatial analysing units can discretize the region.

Comparable

All grids are with same size, their attributes are comparable under same standard.

Controllable

Under grid-based framework, aggregation accuracy is controllable. Defining smaller grids will improve the accuracy, but increase computing burden.

Efficient

Using TransBigData, GPS data can match to grids with small computational complexity. High computation speed for the matching between grids and GPS data.

In TransBigData, the gridding framework is determined by the gridding params. Each of the gridding params can define a griding coordinate system. The params are as follows:

```
params=(lonStart,latStart,deltaLon,deltaLat,theta)
```

However, how to choose an appropriate gridding params in our research is the most basic thing, which may have a great impact on the final analysis results.

The selection of the grid depends on the data and the purpose analyzed.

Suppose we want to use the grid system to analyze the vehicle travel trajectory. If the grid boundaries coincide with the road centerline, the vehicle travel through the road section will generate the GPS points along the grid boundry. There will be great differences in the grid sequence generated after matching GPS to grids even if the vehicles are passing through the same road section. In another word, a better grid coordinate system should be that the trajectory travel through the same path should have similar grid sequence.

A good idea is to input the urban road network data and optimize the grid parameters from the road network. However, for a gridding framework like TransBigData, this is not the best solution. The GPS data we want to analyze is not only the vehicle trajectory data and they do not have to follow a given road network. Moreover, the spatial feature of the road network is already included in the vehicle trajectory. Thus, the selection of gridding parameters should depend on the original spatial attributes of the GPS data.

When analysing individual mobility data, the optimal grid selection criteria are also different. Since individuals usually stay more time and generate more data in their activity points, a better gridding should match these data into the same grid. The result should be that few grids occupy more data.

Here, we offer three methods to optimize the griding params: centerdist, gini and gridscount

[1]:
```
import pandas as pd
import geopandas as gpd
import transbigdata as tbd
#Read taxi gps data
tripdata = pd.read_csv(r'data/TaxiData-Sample.csv')
tripdata.columns = ['track_id','time','lon','lat','OpenStatus','Speed']

#Retain the data in given area
area = gpd.read_file(r'data/gis/szarea1.json')
tripdata = tbd.clean_outofshape(tripdata,area,col=['lon','lat'])

#Generate initial griding params
bounds = [113.6,22.4,114.8,22.9]
initialparams = tbd.area_to_params(bounds,accuracy = 500)
```

centerdist: Minimize the distance between grid center and GPS data

When a batch of data with close distance are distributed at the edge of the grid, the deviation of GPS data will cause these data to be matched into different grids. So one of the solution is to minimize the distance between grid center and GPS data.

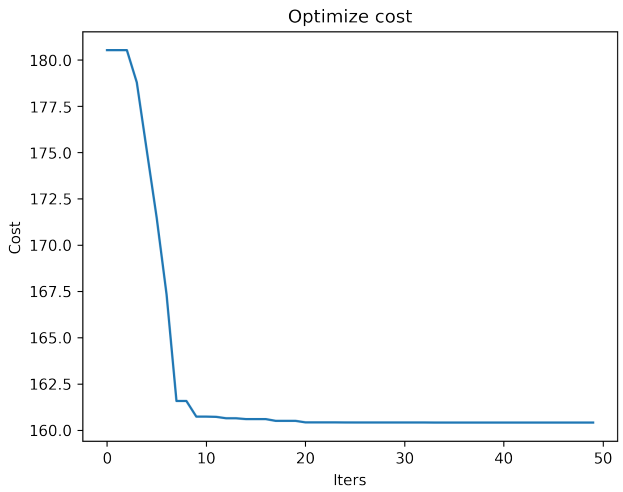
[2]:
```
#Optimize gridding params
params_optimized = tbd.grid_params_optimize(tripdata,
                                            initialparams,
                                            col=['track_id','lon','lat'],
                                            optmethod='centerdist',
```

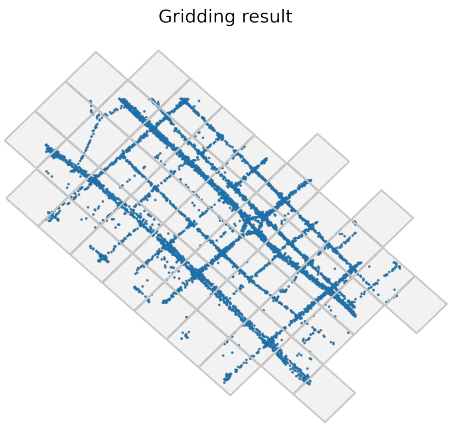
(continues on next page)

(continued from previous page)

```
                                            sample=0, #not sampling
                                            printlog=True)
```

```
Optimized index centerdist: 160.41280636449184
Optimized gridding params: {'slon': 113.60144616975187, 'slat': 22.401543058590295,
↪'deltalon': 0.004872390756896538, 'deltalat': 0.004496605206422906, 'theta': 43.
↪585298279322615, 'method': 'rect'}
```





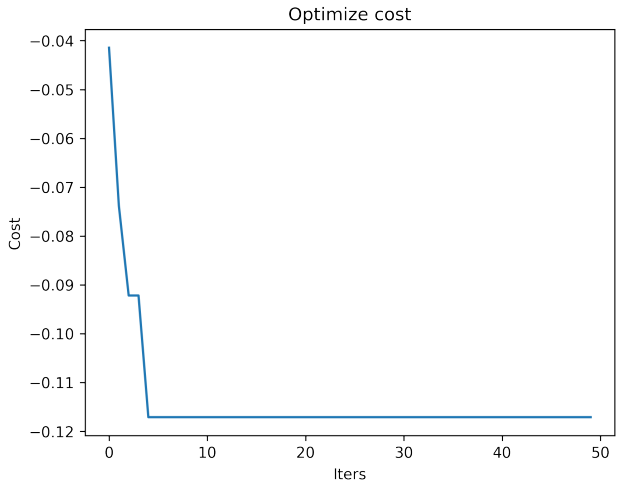
gini: Maximize the gini index

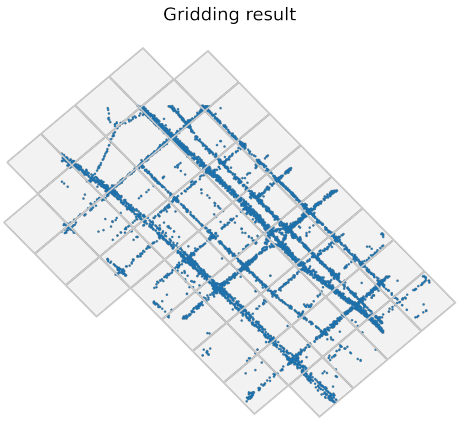
In economics, Gini index is a measure of statistical dispersion intended to represent the income inequality or the wealth inequality within a nation or a social group. Here, we can borrow the concept of Gini index to represent the distribution of GPS data in the grids. The higher of the Gini index represents that the data is more concentrated distribution in the given grids.

The gini index is more helpful in analysing human mobility data.

[3]:
```
#Optimize griding params
params_optimized = tbd.grid_params_optimize(tripdata,
                                            initialparams,
                                            col=['track_id','lon','lat'],
                                            optmethod='gini',
                                            sample=0, #not sampling
                                            printlog=True)
```

```
Optimized index gini: -0.11709661279249717
Optimized gridding params: {'slon': 113.60363252207824, 'slat': 22.40161914185426,
↪'deltalon': 0.004872390756896538, 'deltalat': 0.004496605206422906, 'theta': 47.
↪730990684694575, 'method': 'rect'}
```



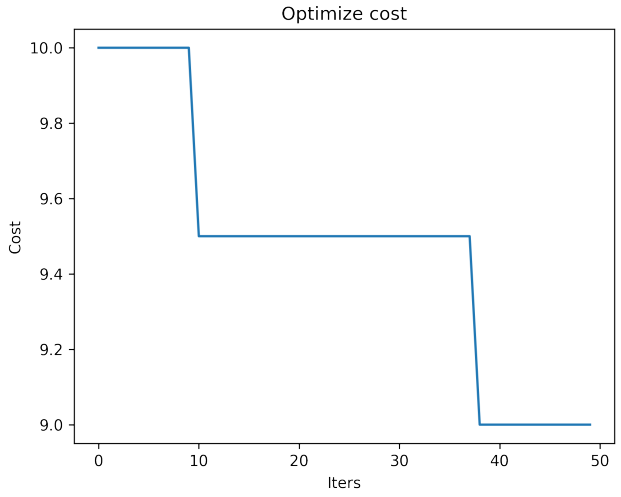


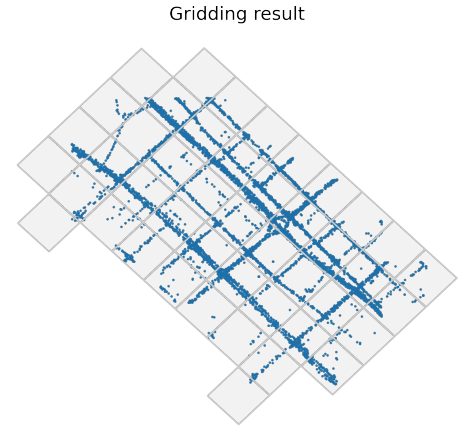
gridscount: Minimize the average count of grids for individuals

Under this standard, each individual should appear in as few grids as possible.

[4]:
```
#Optimize griding params
params_optimized = tbd.grid_params_optimize(tripdata,
                                            initialparams,
                                            col=['track_id','lon','lat'],
                                            optmethod='gridscount',
                                            sample=0, #not sampling
                                            printlog=True)
```

```
Optimized index gridscount: 9.0
Optimized gridding params: {'slon': 113.60372085909265, 'slat': 22.4030027408156666,
↪'deltalon': 0.004872390756896538, 'deltalat': 0.004496605206422906, 'theta': 44.
↪56000665402531, 'method': 'rect'}
```



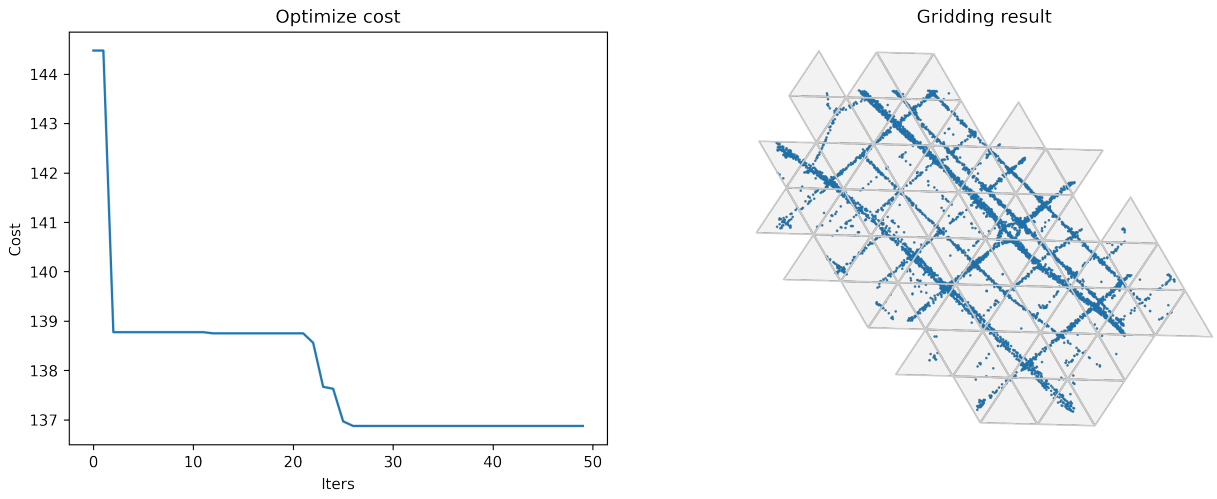


Also support optimizing triangle and hexagon gridding parameters

[5]:
```
initialparams['method'] = 'tri'
```

[6]:
```
#Optimize gridding params
params_optimized = tbd.grid_params_optimize(tripdata,
                                            initialparams,
                                            col=['track_id','lon','lat'],
                                            optmethod='centerdist',
                                            sample=0, #not sampling
                                            printlog=True)
```

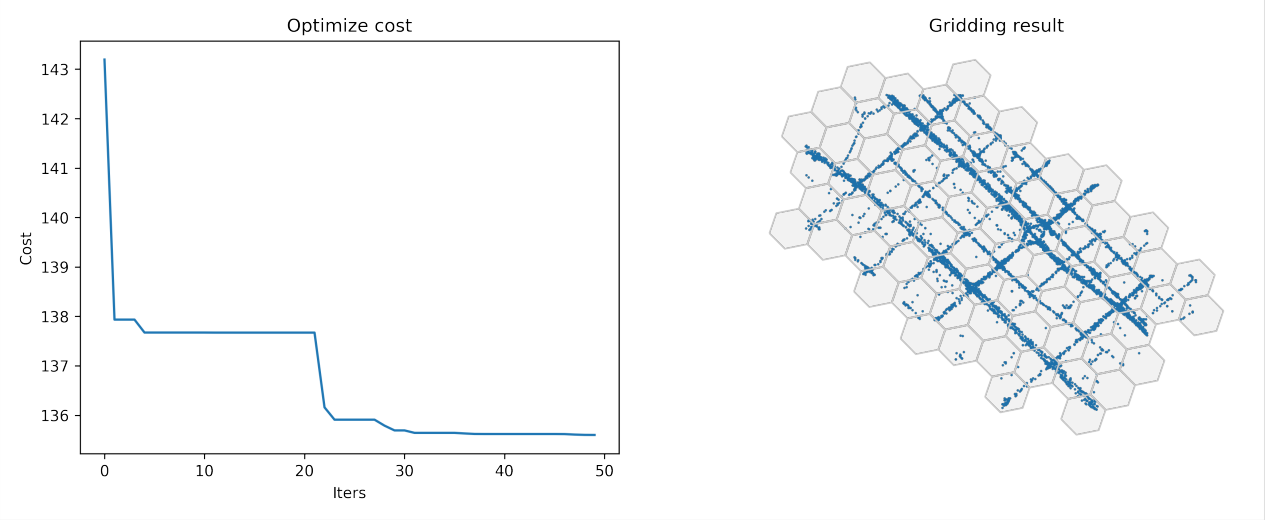
```
Optimized index centerdist: 136.87564489047065
Optimized gridding params: {'slon': 113.60421146982776, 'slat': 22.402738210124514,
↪'deltalon': 0.004872390756896538, 'deltalat': 0.004496605206422906, 'theta': 31.
↪61303640854649, 'method': 'tri'}
```



[7]:
```
initialparams = tbd.area_to_params(bounds,accuracy = 500/(6**0.5))
initialparams['method'] = 'hexa'
```

[8]:
```
#Optimize gridding params
params_optimized = tbd.grid_params_optimize(tripdata,
                                            initialparams,
                                            col=['track_id','lon','lat'],
                                            optmethod='centerdist',
                                            sample=0, #not sampling
                                            printlog=True)
```

```
Optimized index centerdist: 135.60103782128888
Optimized gridding params: {'slon': 113.60043088516572, 'slat': 22.400303375881162,
↪'deltalon': 0.0019891451969749397, 'deltalat': 0.0018357313884130575, 'theta': 17.
↪62535531106509, 'method': 'hexa'}
```



4.1.2 Advanced

6 Mobile phone data processing

In this example, we will introduce how to use the TransBigData to process mobile phone data.
Firstly, import the TransBigData and read the data using pandas

[1]:
```
import pandas as pd
import transbigdata as tbd

data = pd.read_csv(r'data/mobiledata_sample.csv')
#make sure the time column is correct
data['stime'] = pd.to_datetime(data['stime'], format='%Y%m%d%H%M')
data = data.sort_values(by = ['user_id','stime'])
data.head()
```

[1]:
```
                            user_id               stime  longitude  \
78668  00466ab30de56db7efbd04991b680ae1 2018-06-01 00:00:00     121.43
78669  00466ab30de56db7efbd04991b680ae1 2018-06-01 03:35:00     121.43
78670  00466ab30de56db7efbd04991b680ae1 2018-06-01 04:25:00     121.43
78671  00466ab30de56db7efbd04991b680ae1 2018-06-01 05:15:00     121.43
78289  00466ab30de56db7efbd04991b680ae1 2018-06-01 06:05:00     121.43

       latitude      date
78668    30.175  20180601
78669    30.175  20180601
78670    30.175  20180601
78671    30.175  20180601
78289    30.175  20180601
```

Identify stay and move infomation from mobile phone trajectory data

When processing mobile phone data, TransBigData's approach is to first correspond the data to the grids and treat the data within the same grid as being at the same location to avoid data positioning errors that cause the same location to be identified as multiple.

[3]:
```
#Obtain gridding parameters
params = tbd.area_to_params([121.860, 29.295, 121.862, 29.301], accuracy=500)
#Identify stay and move infomation from mobile phone trajectory data
stay,move = tbd.traj_stay_move(data,params,col = ['user_id','stime','longitude',
↪'latitude'])
```

[4]:
```
stay.head()
```

[4]:
```
                              user_id               stime  LONCOL  LATCOL
78668  00466ab30de56db7efbd04991b680ae1 2018-06-01 00:00:00     -83     196  \
78303  00466ab30de56db7efbd04991b680ae1 2018-06-01 07:36:00     -81     191
78364  00466ab30de56db7efbd04991b680ae1 2018-06-01 10:38:00     -81     191
78399  00466ab30de56db7efbd04991b680ae1 2018-06-01 12:20:00     -83     196
78471  00466ab30de56db7efbd04991b680ae1 2018-06-01 14:34:00     -60     189

                    etime      lon     lat  duration  stayid
78668 2018-06-01 07:21:00  121.430  30.175   26460.0       0
78303 2018-06-01 10:38:00  121.444  30.152   10920.0       1
78364 2018-06-01 12:02:00  121.444  30.152    5040.0       2
78399 2018-06-01 13:04:00  121.430  30.175    2640.0       3
78471 2018-06-01 16:06:00  121.551  30.143    5520.0       4
```

[5]:
```
move.head()
```

[5]:
```
                              user_id  SLONCOL  SLATCOL               stime
78668  00466ab30de56db7efbd04991b680ae1      -83      196 2018-06-01 00:00:00  \
78668  00466ab30de56db7efbd04991b680ae1      -83      196 2018-06-01 07:21:00
78303  00466ab30de56db7efbd04991b680ae1      -81      191 2018-06-01 10:38:00
78364  00466ab30de56db7efbd04991b680ae1      -81      191 2018-06-01 12:02:00
78399  00466ab30de56db7efbd04991b680ae1      -83      196 2018-06-01 13:04:00

          slon    slat               etime     elon    elat  ELONCOL  ELATCOL
78668  121.430  30.175 2018-06-01 00:00:00  121.430  30.175    -83.0    196.0  \
78668  121.430  30.175 2018-06-01 07:36:00  121.444  30.152    -81.0    191.0
78303  121.444  30.152 2018-06-01 10:38:00  121.444  30.152    -81.0    191.0
78364  121.444  30.152 2018-06-01 12:20:00  121.430  30.175    -83.0    196.0
78399  121.430  30.175 2018-06-01 14:34:00  121.551  30.143    -60.0    189.0

       duration  moveid
78668       0.0       0
78668     900.0       1
78303       0.0       2
78364    1080.0       3
78399    5400.0       4
```

Home and work place identify

[6]:
```
#Identify home location
home = tbd.mobile_identify_home(stay, col=['user_id','stime', 'etime','LONCOL', 'LATCOL',
↪'lon','lat'], start_hour=8, end_hour=20 )
home.head()
```

[6]:

	user_id	LONCOL	LATCOL	lon	lat
2036	fcc3a9e9df361667e00ee5c16cb08922	-147	292	121.103	30.610
2019	f71e9d7d78e6f5bc9539d141e3a5a1c4	-216	330	120.745	30.778
2001	f6b65495b63574c2eb73c7e63ae38252	-225	-286	120.699	28.011
1982	f1f4224a60da630a0b83b3a231022123	102	157	122.387	30.000
1942	e96739aedb70a8e5c4efe4c488934b43	-223	278	120.708	30.546

[7]:
```
#Identify work location
work = tbd.mobile_identify_work(stay, col=['user_id', 'stime', 'etime', 'LONCOL', 'LATCOL
↪','lon','lat'], minhour=3, start_hour=8, end_hour=20,workdaystart=0, workdayend=4)
work.head()
```

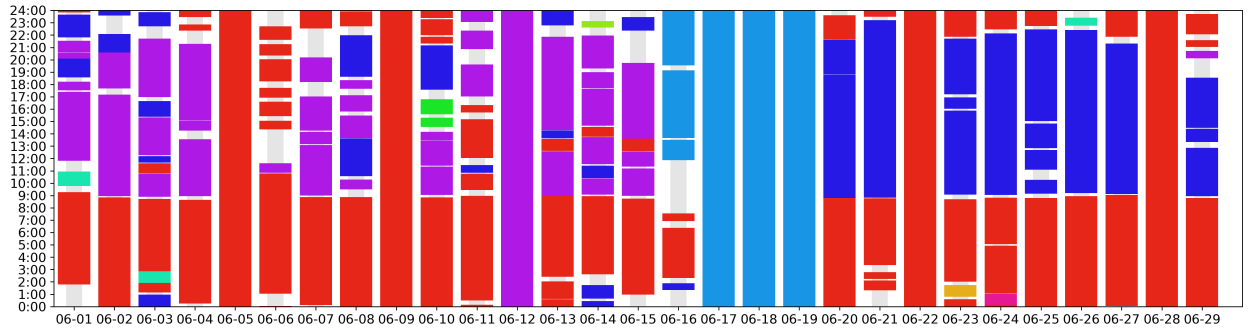
[7]:

	user_id	LONCOL	LATCOL	lon	lat
0	fcc3a9e9df361667e00ee5c16cb08922	-148	292	121.097	30.610
1	f71e9d7d78e6f5bc9539d141e3a5a1c4	-219	325	120.732	30.757
3	f1f4224a60da630a0b83b3a231022123	103	153	122.390	29.982
5	e1a1dfb5a77578c889bd3368ffe1d30f	-62	138	121.540	29.915
6	e0e30d88fc4f4b8a1d649baf9dd1274e	-436	-35	119.614	29.137

[8]:
```
# If you want to filter out the users with work place location from home location
home['flag'] = 1
work = pd.merge(work,home,how='left')
home = home.drop(['flag'],axis = 1)
work = work[work['flag'].isnull()].drop(['flag'],axis = 1)
```

Plot activity

[9]:
```
#Plot the activity of the user, different color represent different location
uid = 'fcc3a9e9df361667e00ee5c16cb08922'
stay['group'] = stay['LONCOL'].astype(str)+','+stay['LATCOL'].astype(str)
tbd.plot_activity(stay[stay['user_id']==uid],figsize = (20, 5))
```



7 Modeling for subway network topology

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

The following example shows how to use TransBigData to download subway lines and to build a topological network model for the subway line network

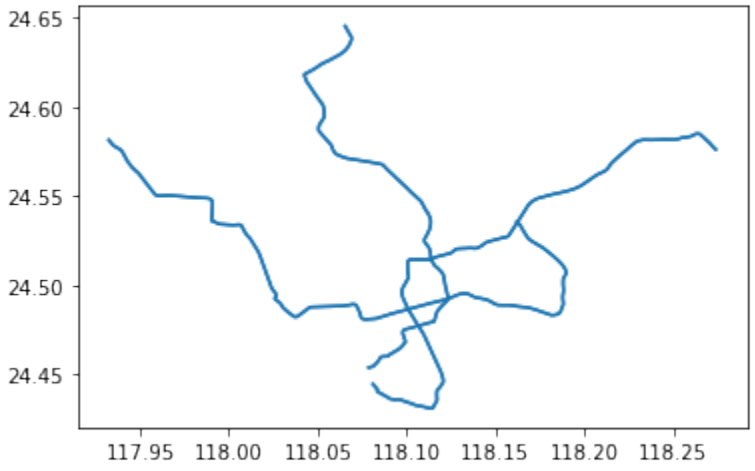
Download subway lines

[2]:
```
import pandas as pd
import numpy as np
import geopandas as gpd
import transbigdata as tbd
line,stop = tbd.getbusdata('',['1','2','3'])
```

```
Obtaining city id: success
1
1(-) success
1(-) success
2
2(-) success
2(-) success
3
3(-) success
3(-) success
3(-) success
3(-) success
```

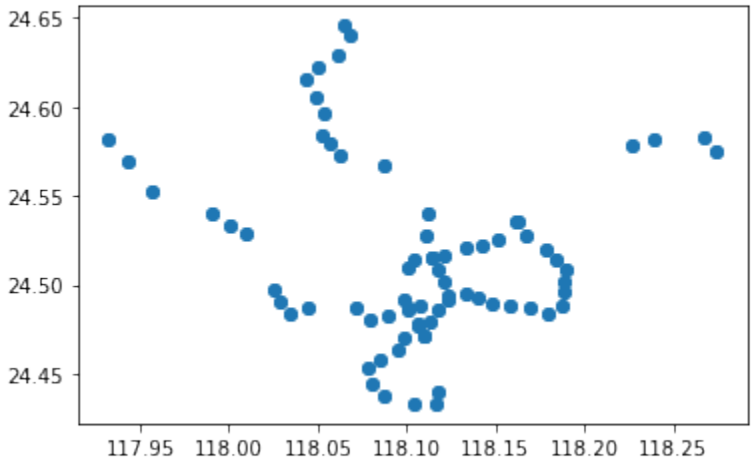
[3]:
```
line.plot()
```

[3]:
```
<AxesSubplot:>
```



[4]:
```
stop.plot()
```

[4]:
```
<AxesSubplot:>
```

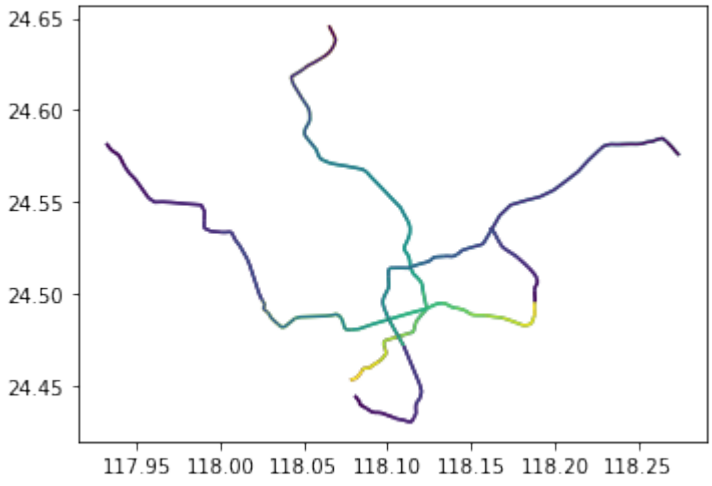


Splitting the metro line into sections

The `tbd.split_subwayline` method can be used to slice the metro line with metro stations to obtain metro section information (This step is useful in subway passenger flow visualization)

[5]:
```
metroline_splited = tbd.split_subwayline(line,stop)
metroline_splited.plot(column = pd.Series(metroline_splited.index))
```

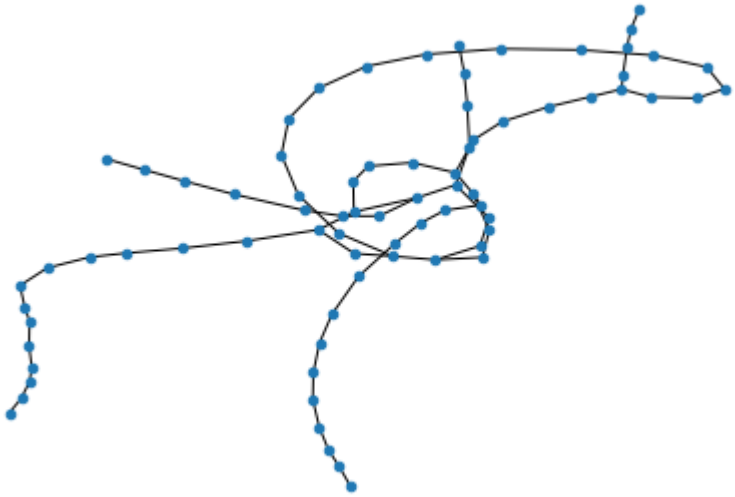
[5]:
```
<AxesSubplot:>
```



Modeling for subway network topology

We can also use the metro station data to build a topology model of the metro network. This step is useful for subsequent identification of metro travel paths. The graph generated relies on NetworkX.

[6]:
```
#Modeling for subway network topology
line['speed'] = 55 #operation speed 55km/h
line['stoptime'] = 0.5 #stop time at each stations 30s
import networkx as nx
G = tbd.metro_network(line,stop, transfertime=5)
nx.draw(G,node_size=20)
```



[7]:
```
path = tbd.get_shortest_path(G,stop,'','')
path
```

[7]:
```
['1',
 '1',
 '1',
 '1',
 '1',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
 '3',
```

(continues on next page)

(continued from previous page)

```
 '3']
```

[8]:
```
tbd.get_path_traveltime(G,path)
```

[8]:
```
49.34680872200037
```

[9]:
```
# k th shortest paths
paths =  tbd.get_k_shortest_paths(G,stop,'','',2)
paths
```

[9]:
```
[['1',
  '1',
  '1',
  '1',
  '1',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3'],
 ['1',
  '1',
  '1',
  '1',
  '1',
  '1',
  '1',
  '1',
  '1',
  '1',
  '1',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3',
  '3']]
```

[10]:
```
tbd.get_path_traveltime(G,paths[1])
```

[10]:
```
49.633890662025024
```

8 Community detection for bicycle-sharing demand

For bicycle sharing demand, each trip of can be seen as a process from the starting loaction to the end loaction. When we regard the start point and the end point as nodes, and the travel between them as edges, a network can be constructed. By analysing this network, we can get information about the spatial connection structure of the city or the macro travel characteristics of the bicycle sharing demand.

Community detection, also called graph partition, helps us to reveal the hidden relations among the nodes in the network. In this example, we will introduce how to integrate TransBigData into the analysis process of community detection from bicycle-sharing data.

To run this example, you may have to install `igraph` and `seaborn`:

pip install igraph
pip install seaborn

Data preprocessing

[1]:
```
# Fristly, import packages.
import pandas as pd
import numpy as np
import geopandas as gpd
import transbigdata as tbd
```

[2]:
```
#Read bicycle sharing data
bikedata = pd.read_csv(r'data/bikedata-sample.csv')
bikedata.head(5)
```

[2]:

	BIKE_ID	DATA_TIME	LOCK_STATUS	LONGITUDE	LATITUDE
0	5	2018-09-01 0:00:36	1	121.363566	31.259615
1	6	2018-09-01 0:00:50	0	121.406226	31.214436
2	6	2018-09-01 0:03:01	1	121.409402	31.215259
3	6	2018-09-01 0:24:53	0	121.409228	31.214427
4	6	2018-09-01 0:26:38	1	121.409771	31.214406

[3]:
```
#Read the polygon of the study area
shanghai_admin = gpd.read_file(r'data/shanghai.json')
#delete the data outside of the study area
bikedata = tbd.clean_outofshape(bikedata, shanghai_admin, col=['LONGITUDE', 'LATITUDE'],␣
↪accuracy=500)
```

Identify Bicycle sharing trip information using `tbd.bikedata_to_od`

[4]:
```
move_data,stop_data = tbd.bikedata_to_od(bikedata,
                    col = ['BIKE_ID','DATA_TIME','LONGITUDE','LATITUDE','LOCK_STATUS'])
move_data.head(5)
```

[4]:
```
      BIKE_ID               stime        slon       slat  \
96          6   2018-09-01 0:00:50  121.406226  31.214436
561         6   2018-09-01 0:24:53  121.409228  31.214427
564         6   2018-09-01 0:50:16  121.409727  31.214403
784         6   2018-09-01 0:53:38  121.413333  31.214951
1028        6  2018-09-01 11:35:01  121.419261  31.213414

                    etime        elon       elat
96     2018-09-01 0:03:01  121.409402  31.215259
561    2018-09-01 0:26:38  121.409771  31.214406
564    2018-09-01 0:52:14  121.412610  31.214905
784    2018-09-01 0:55:38  121.412656  31.217051
1028  2018-09-01 11:35:13  121.419518  31.213657
```

[5]:
```
#Calculate the travel distance
move_data['distance'] = tbd.getdistance(move_data['slon'],move_data['slat'],move_data[
↪'elon'],move_data['elat'])
#Remove too long and too short trips
move_data = move_data[(move_data['distance']>100)&(move_data['distance']<10000)]
```

Perform data gridding:

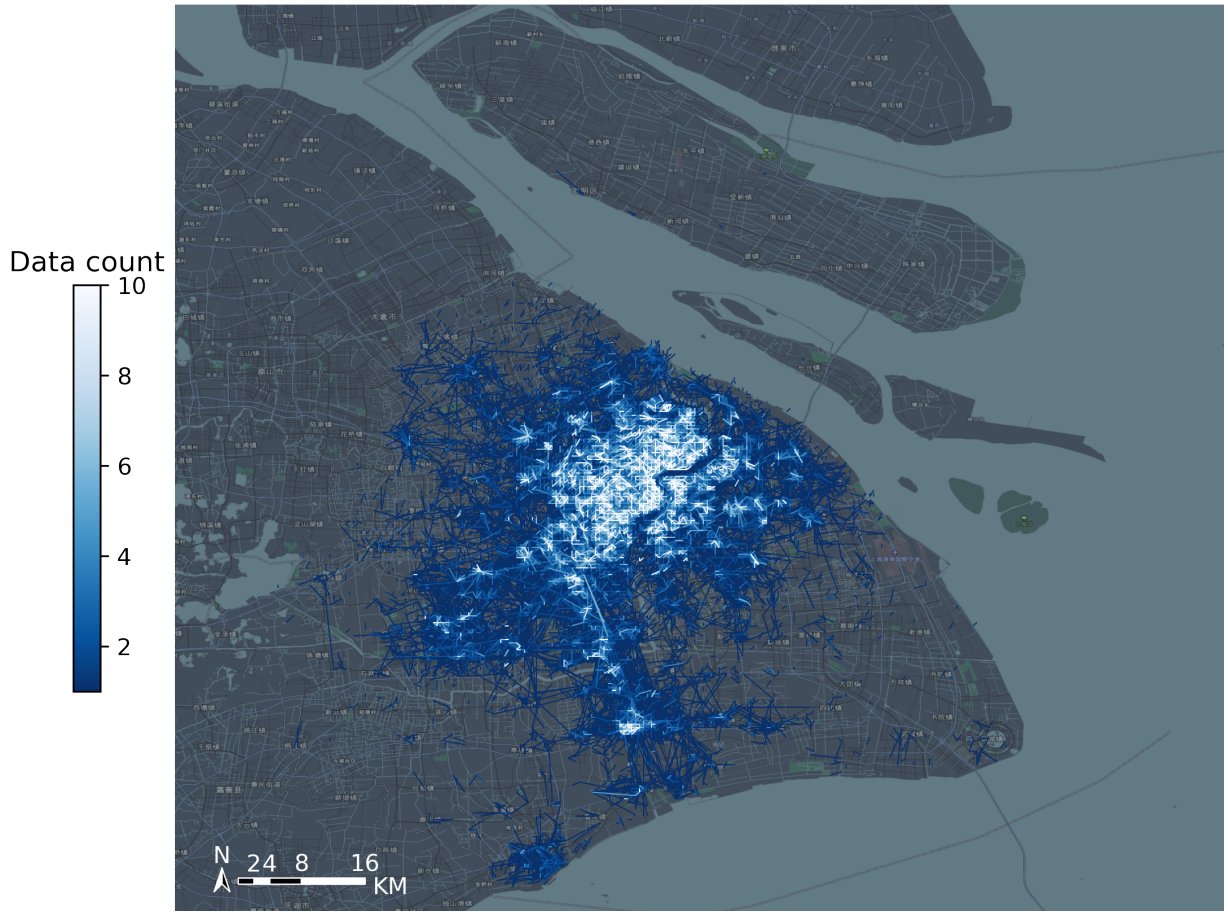
[6]:
```
#obtain gridding params
bounds = (120.85, 30.67, 122.24, 31.87)
params = tbd.grid_params(bounds,accuracy = 500)
#aggregate the travel informations
od_gdf = tbd.odagg_grid(move_data, params, col=['slon', 'slat', 'elon', 'elat'])
od_gdf.head(5)
```

```
/opt/anaconda3/envs/transbigdata/lib/python3.9/site-packages/pandas/core/dtypes/cast.py:
↪122: ShapelyDeprecationWarning: The array interface is deprecated and will no longer␣
↪work in Shapely 2.0. Convert the '.coords' to a numpy array instead.
  arr = construct_1d_object_array_from_listlike(values)
```

[6]:
```
       SLONCOL  SLATCOL  ELONCOL  ELATCOL  count      SHBLON     SHBLAT  \
0           26       95       26       96      1  120.986782  31.097177
40803      117      129      116      127      1  121.465519  31.250062
40807      117      129      117      128      1  121.465519  31.250062
40810      117      129      117      131      1  121.465519  31.250062
40811      117      129      118      126      1  121.465519  31.250062

           EHBLON     EHBLAT  \
0      120.986782  31.101674
40803  121.460258  31.241069
40807  121.465519  31.245565
40810  121.465519  31.259055
40811  121.470780  31.236572

                                                geometry
0      LINESTRING (120.98678 31.09718, 120.98678 31.1...
40803  LINESTRING (121.46552 31.25006, 121.46026 31.2...
40807  LINESTRING (121.46552 31.25006, 121.46552 31.2...
40810  LINESTRING (121.46552 31.25006, 121.46552 31.2...
40811  LINESTRING (121.46552 31.25006, 121.47078 31.2...
```

Visualize the OD data

[7]:
```
#Create figure
import matplotlib.pyplot as plt
fig =plt.figure(1,(8,8),dpi=300)
ax =plt.subplot(111)
plt.sca(ax)

#Load basemap
tbd.plot_map(plt,bounds,zoom = 11,style = 8)

#Create colorbar
cax = plt.axes([0.05, 0.33, 0.02, 0.3])
plt.title('Data count')
plt.sca(ax)

#Plot OD
od_gdf.plot(ax = ax,column = 'count',cmap = 'Blues_r',linewidth = 0.5,vmax = 10,cax =␣
↪cax,legend = True)

#Plot compass and scale
tbd.plotscale(ax,bounds = bounds,textsize = 10,compasssize = 1,textcolor = 'white',
↪accuracy = 2000,rect = [0.06,0.03],zorder = 10)
plt.axis('off')
plt.xlim(bounds[0],bounds[2])
plt.ylim(bounds[1],bounds[3])
plt.show()
```



Create Network

Extract node data

Combine the LONCOL and LATCOL columns into one field and extract node set

[8]:
```
#Combine the ``LONCOL`` and ``LATCOL`` columns into one field
od_gdf['S'] = od_gdf['SLONCOL'].astype(str) + ',' + od_gdf['SLATCOL'].astype(str)
od_gdf['E'] = od_gdf['ELONCOL'].astype(str) + ',' + od_gdf['ELATCOL'].astype(str)
#extract node set
node = set(od_gdf['S'])|set(od_gdf['E'])
node = pd.DataFrame(node)
#reindex the node
node['id'] = range(len(node))
node
```

[8]:

	0	id
0	164,81	0
1	71,125	1
2	102,118	2
3	125,115	3

(continues on next page)

(continued from previous page)

```
4      143,76     4
...       ...   ...
9806   98,167  9806
9807   46,130  9807
9808   118,82  9808
9809   158,57  9809
9810  104,169  9810

[9811 rows x 2 columns]
```

Extract edge data

[9]:
```
#Merge the node information to the OD data to extract edge data.
node.columns = ['S','S_id']
od_gdf = pd.merge(od_gdf,node,on = ['S'])
node.columns = ['E','E_id']
od_gdf = pd.merge(od_gdf,node,on = ['E'])
#Extract edge data
edge = od_gdf[['S_id','E_id','count']]
edge
```

[9]:
```
       S_id  E_id  count
0      6251  4211      1
1      5879  8676      1
2      8432  8676      3
3      5511  8676      1
4      3386  8676      1
...     ...   ...    ...
68468  5663  5835      2
68469  7738  4266      2
68470   360  8003      2
68471  6759   601      3
68472  6081  3107      3

[68473 rows x 3 columns]
```

Create Network

[10]:
```
import igraph
#Create Network
g = igraph.Graph()
#Add node
g.add_vertices(len(node))
#Add edge
g.add_edges(edge[['S_id','E_id']].values)
#Add weight
edge_weights = edge[['count']].values
for i in range(len(edge_weights)):
    g.es[i]['weight'] = edge_weights[i]
```

Community detection

[11]:
```
#Community detection
g_clustered = g.community_multilevel(weights = edge_weights, return_levels=False)
```

[12]:
```
#Modularity
g_clustered.modularity
```

[12]:
```
0.8496074605497185
```

[13]:
```
#Assign the group result to the node
node['group'] = g_clustered.membership
#rename the columns
node.columns = ['grid','node_id','group']
node
```

[13]:
```
         grid  node_id  group
0      164,81        0      0
1      71,125        1      1
2     102,118        2      2
3     125,115        3      3
4      143,76        4      4
...       ...      ...    ...
9806   98,167     9806      9
9807   46,130     9807    555
9808   118,82     9808      6
9809   158,57     9809    132
9810  104,169     9810     86

[9811 rows x 3 columns]
```

Visualize the community

[14]:
```
#Count the number of grids per community
group = node['group'].value_counts()
#Extract communities with more than 10 grids
group = group[group>10]
#Retain only these community grids
node = node[node['group'].apply(lambda r:r in group.index)]

#Get the grid number
node['LONCOL'] = node['grid'].apply(lambda r:r.split(',')[0]).astype(int)
node['LATCOL'] = node['grid'].apply(lambda r:r.split(',')[1]).astype(int)
#Generate the geometry
node['geometry'] = tbd.gridid_to_polygon(node['LONCOL'],node['LATCOL'],params)
#Change it into GeoDataFrame
import geopandas as gpd
node = gpd.GeoDataFrame(node)
node
```

```
/var/folders/b0/q8rx9fj965b5p7yqq8zhvdx80000gn/T/ipykernel_30130/418053260.py:9:␣
↪SettingWithCopyWarning:
```

(continues on next page)

(continued from previous page)

```
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  node['LONCOL'] = node['grid'].apply(lambda r:r.split(',')[0]).astype(int)
/var/folders/b0/q8rx9fj965b5p7yqq8zhvdx80000gn/T/ipykernel_30130/418053260.py:10:␣
↪SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  node['LATCOL'] = node['grid'].apply(lambda r:r.split(',')[1]).astype(int)
/var/folders/b0/q8rx9fj965b5p7yqq8zhvdx80000gn/T/ipykernel_30130/418053260.py:12:␣
↪SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  node['geometry'] = tbd.gridid_to_polygon(node['LONCOL'],node['LATCOL'],params)
```

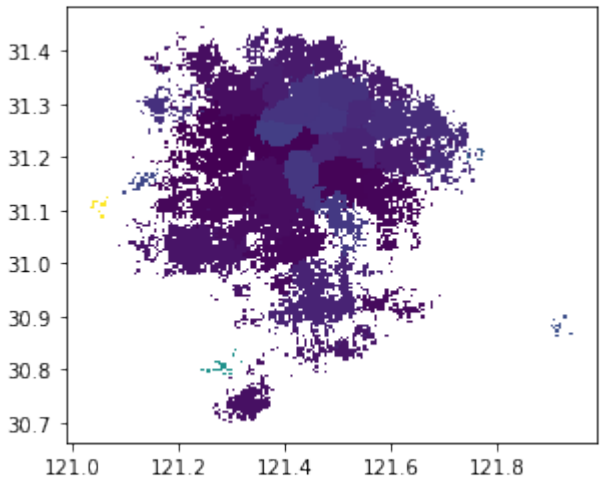
[14]:

	grid	node_id	group	LONCOL	LATCOL	geometry
1	71,125	1	1	71	125	POLYGON ((121.22089 31.22983, 121.22615 31.229...
2	102,118	2	2	102	118	POLYGON ((121.38398 31.19835, 121.38924 31.198...
3	125,115	3	3	125	115	POLYGON ((121.50498 31.18486, 121.51024 31.184...
4	143,76	4	4	143	76	POLYGON ((121.59967 31.00949, 121.60493 31.009...
5	142,87	5	4	142	87	POLYGON ((121.59441 31.05896, 121.59967 31.058...
...	...	...	...	...	...	...
9802	103,103	9802	8	103	103	POLYGON ((121.38924 31.13090, 121.39450 31.130...
9803	162,133	9803	28	162	133	POLYGON ((121.69963 31.26580, 121.70489 31.265...
9804	107,130	9804	41	107	130	POLYGON ((121.41028 31.25231, 121.41554 31.252...
9806	98,167	9806	9	98	167	POLYGON ((121.36293 31.41868, 121.36819 31.418...
9808	118,82	9808	6	118	82	POLYGON ((121.46815 31.03647, 121.47341 31.036...

[8522 rows x 6 columns]

[15]:
```
node.plot('group')
```

[15]:
```
<AxesSubplot:>
```



[16]:
```
#Use the group column to merge polygon
node_community = tbd.merge_polygon(node,'group')
#Input polygon GeoDataFrame data, take the exterior boundary of the polygon to form a␣
↪new polygon
node_community = tbd.polyon_exterior(node_community,minarea = 0.000100)
```

```
/opt/anaconda3/envs/transbigdata/lib/python3.9/site-packages/transbigdata/gisprocess.py:
↪205: ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecated and␣
↪will be removed in Shapely 2.0. Use the `geoms` property to access the constituent␣
↪parts of a multi-part geometry.
  for i in p:
```

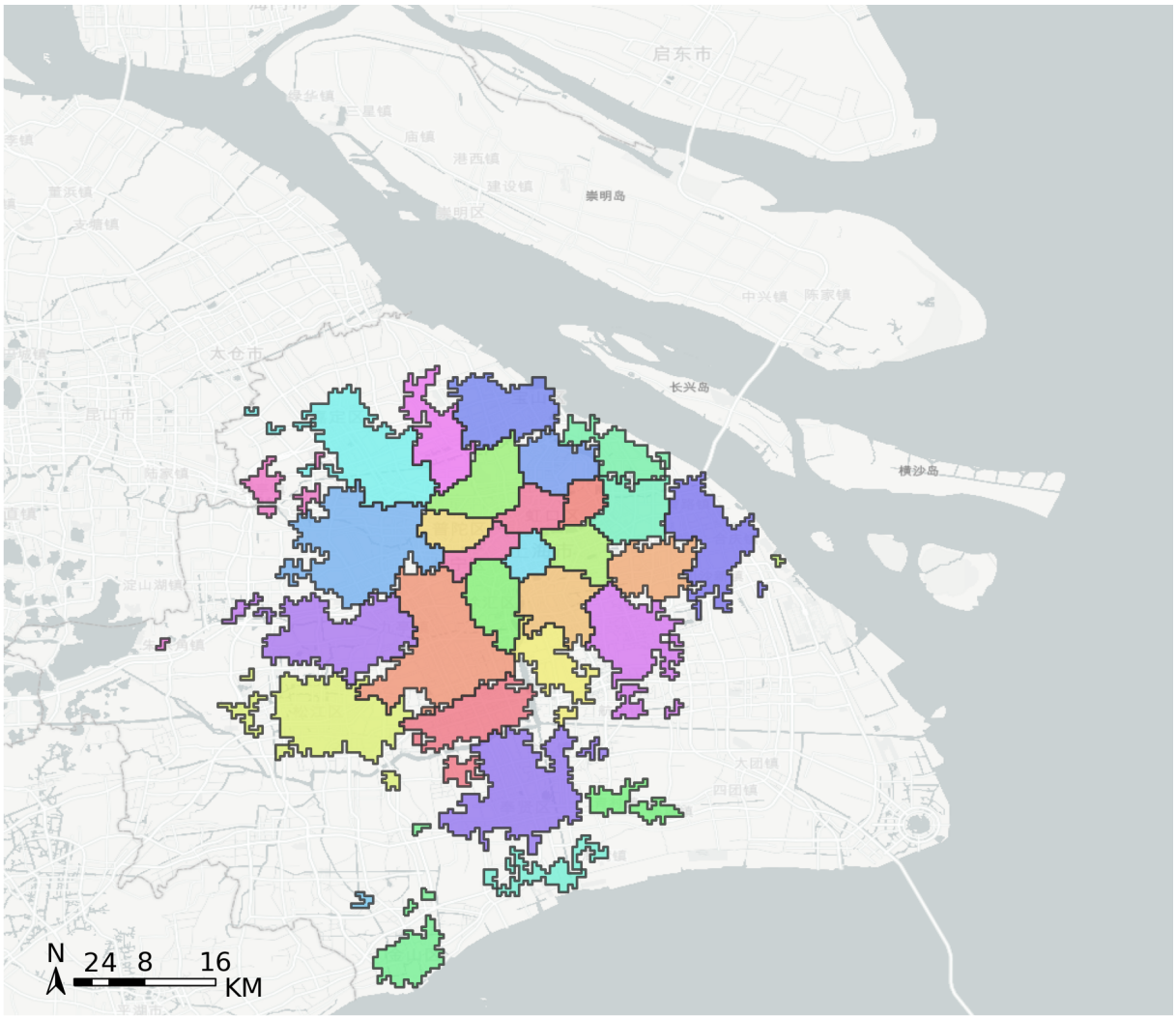
[17]:
```
#Generate palette
import seaborn as sns
## l: Luminance
## s: Saturation
cmap = sns.hls_palette(n_colors=len(node_community), l=.7, s=0.8)
sns.palplot(cmap)
```

[19]:
```
#Create figure
import matplotlib.pyplot as plt
fig =plt.figure(1,(8,8),dpi=300)
ax =plt.subplot(111)
plt.sca(ax)
#Load basemap
tbd.plot_map(plt,bounds,zoom = 10,style = 6)
#Set colormap
from matplotlib.colors import ListedColormap
#Disrupting the order of the community
node_community = node_community.sample(frac=1)
#Plot community
node_community.plot(cmap = ListedColormap(cmap),ax = ax,edgecolor = '#333',alpha = 0.8)
```

(continues on next page)

(continued from previous page)

```
#Add scale
tbd.plotscale(ax,bounds = bounds,textsize = 10,compasssize = 1,textcolor = 'k'
              ,accuracy = 2000,rect = [0.06,0.03],zorder = 10)
plt.axis('off')
plt.xlim(bounds[0],bounds[2])
plt.ylim(bounds[1],bounds[3])
plt.show()
```



9 Identifying arrival and departure information from Bus GPS data

To run this example, you may have to install `seaborn`:

```
pip install seaborn
```

The following example shows how to use TransBigData to process bus GPS data, including identifying bus arrival and departure information, calculate travel time and operating speed for buses.

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
import transbigdata as tbd
import pandas as pd
import geopandas as gpd
```

Read data

Read bus GPS data

[3]:
```
BUS_GPS= pd.read_csv(r'data/busgps.csv',header = None)
BUS_GPS.columns = ['GPSDateTime', 'LineId', 'LineName', 'NextLevel', 'PrevLevel',
       'Strlatlon', 'ToDir', 'VehicleId', 'VehicleNo', 'unknow']
#Convert the time column to datetime type
BUS_GPS['GPSDateTime'] = pd.to_datetime(BUS_GPS['GPSDateTime'])
```

Convert coordinates

[4]:
```
#Slice the latitude and longitude string
BUS_GPS['lon'] = BUS_GPS['Strlatlon'].apply(lambda r:r.split(',')[0])
BUS_GPS['lat'] = BUS_GPS['Strlatlon'].apply(lambda r:r.split(',')[1])
#Convert coordinates
BUS_GPS['lon'],BUS_GPS['lat'] = tbd.gcj02towgs84(BUS_GPS['lon'].astype(float),BUS_GPS[
↪'lat'].astype(float))
BUS_GPS.head(5)
```

[4]:
```
           GPSDateTime  LineId  LineName  NextLevel  PrevLevel  \
0 2019-01-16 23:59:59    7100        71          2          1
1 2019-01-17 00:00:00    7100        71          2          1
2 2019-01-17 00:00:00    7100        71         24         23
3 2019-01-17 00:00:01    7100        71         14         13
4 2019-01-17 00:00:03    7100        71         15         14

              Strlatlon  ToDir VehicleId VehicleNo  unknow         lon  \
0  121.335413,31.173188      1  D-R7103  Z5A-0021       1  121.330858
1  121.334616,31.172271      1  D-R1273  Z5A-0002       1  121.330063
2  121.339955,31.173025      0  D-R5257  Z5A-0020       1  121.335390
3   121.409491,31.20433      0  D-R5192  Z5A-0013       1  121.404843
4  121.398615,31.200253      0  D-T0951  Z5A-0022       1  121.393966

         lat
0  31.175129
1  31.174214
```

(continues on next page)

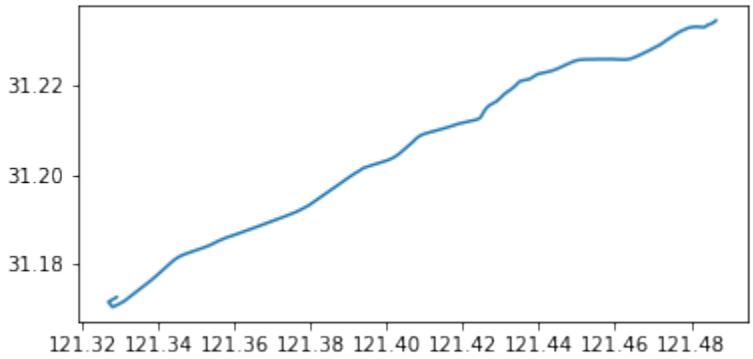
(continued from previous page)

```
2  31.174958
3  31.206179
4  31.202103
```

Read the bus line data

[5]:
```
shp = r'data/busline.json'
linegdf = gpd.GeoDataFrame.from_file(shp,encoding = 'gbk')
line = linegdf.iloc[:1].copy()
line.plot()
```

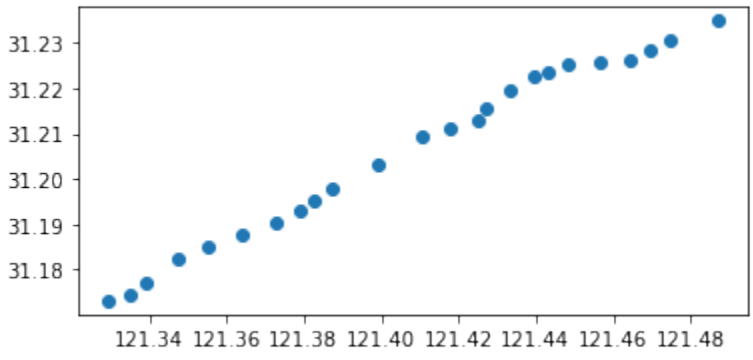
[5]:
```
<AxesSubplot:>
```



Read the bus stop data

[6]:
```
shp = r'data/busstop.json'
stop = gpd.GeoDataFrame.from_file(shp,encoding = 'gbk')
stop = stop[stop['linename'] == '71(-)']
stop.plot()
```

[6]:
```
<AxesSubplot:>
```



Identifying arrival and departure information

[7]:
```
arriveinfo = tbd.busgps_arriveinfo(BUS_GPS,line,stop)
```

```
Cleaning data...
Position matching...
Matching arrival and leaving info...
```

[8]:
```
arriveinfo
```

[8]:
```
            arrivetime           leavetime stopname  VehicleId
0  2019-01-17 07:19:42 2019-01-17 07:31:14             1
1  2019-01-17 09:53:08 2019-01-17 10:09:34             1
0  2019-01-17 07:13:23 2019-01-17 07:15:45               1
1  2019-01-17 07:34:24 2019-01-17 07:35:38               1
2  2019-01-17 09:47:03 2019-01-17 09:50:22               1
..                 ...                 ...       ...        ...
2  2019-01-17 16:35:52 2019-01-17 16:36:49             148
3  2019-01-17 19:21:09 2019-01-17 19:23:44             148
0  2019-01-17 13:36:26 2019-01-17 13:45:04           148
1  2019-01-17 15:52:26 2019-01-17 16:32:46           148
2  2019-01-17 19:24:54 2019-01-17 19:25:55           148

[8984 rows x 4 columns]
```

One-way travel time

Calculate the One-way travel time from `arriveinfo` obtained above. Given start and end stop name of the bus line, `tbd.busgps_onewaytime` can calculate the travel time between the two station.

[9]:
```
onewaytime = tbd.busgps_onewaytime(arriveinfo,
                                   start = '',
                                   end = '',col = ['VehicleId','stopname', 'arrivetime',
↪'leavetime'])
```

[10]:
```
onewaytime
```

[10]:
```
                  time stopname  VehicleId               time1 stopname1  \
0  2019-01-17 07:31:14             1 2019-01-17 08:24:42
1  2019-01-17 10:09:34             1 2019-01-17 11:03:49
0  2019-01-17 13:11:43             2 2019-01-17 14:05:17
1  2019-01-17 15:42:28             2 2019-01-17 16:37:00
0  2019-01-17 18:46:11             3 2019-01-17 19:51:54
..                 ...       ...        ...                   ...        ...
1  2019-01-17 17:11:43           144 2019-01-17 18:13:22
0  2019-01-17 08:15:44           147 2019-01-17 09:14:46
1  2019-01-17 10:51:34           147 2019-01-17 11:50:03
0  2019-01-17 13:45:04           148 2019-01-17 14:44:03
1  2019-01-17 16:32:46           148 2019-01-17 17:31:34

    VehicleId1  duration  shour      direction
0          1.0    3208.0      7  -
```

(continues on next page)

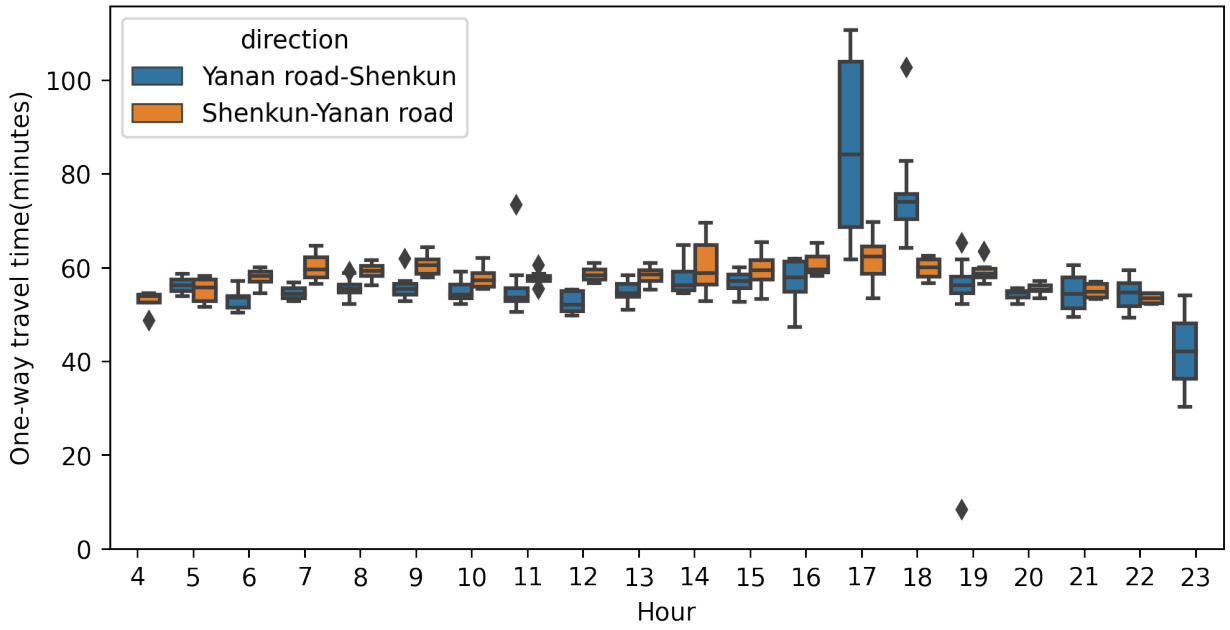
(continued from previous page)

```
1          1.0    3255.0     10  -
0          2.0    3214.0     13  -
1          2.0    3272.0     15  -
0          3.0    3943.0     18  -
..         ...       ...    ...            ...
1        144.0    3699.0     17  -
0        147.0    3542.0      8  -
1        147.0    3509.0     10  -
0        148.0    3539.0     13  -
1        148.0    3528.0     16  -

[375 rows x 9 columns]
```

For English display of the figures, here we will change the station name and direction name into English:

[11]:
```
onewaytime.loc[onewaytime['stopname']=='','stopname']="Yanan road station"
onewaytime.loc[onewaytime['stopname1']=='','stopname1']="Shenkun station"
onewaytime.loc[onewaytime['direction']=='-','direction']="Yanan road-Shenkun"
onewaytime.loc[onewaytime['direction']=='-','direction']="Shenkun-Yanan road"
```

[12]:
```
## Draw box plot for one-way travel time
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
fig     = plt.figure(1,(8,4),dpi = 250)
ax1      = plt.subplot(111)

sns.boxplot(x = 'shour',y = onewaytime['duration']/60,hue = 'direction',data =␣
↪onewaytime)

plt.ylabel('One-way travel time(minutes)')
plt.xlabel('Hour')
plt.ylim(0)
plt.show()
```



Travel speed of the buses

[13]:
```
#Convert coordinate system to projection coordinate system for later calculation of␣
↪distance
line.crs = {'init':'epsg:4326'}
line_2416 = line.to_crs(epsg = 2416)
#Obtain the geometry inside the bus route data
lineshp = line_2416['geometry'].iloc[0]
linename = line_2416['name'].iloc[0]
lineshp
```

[13]:

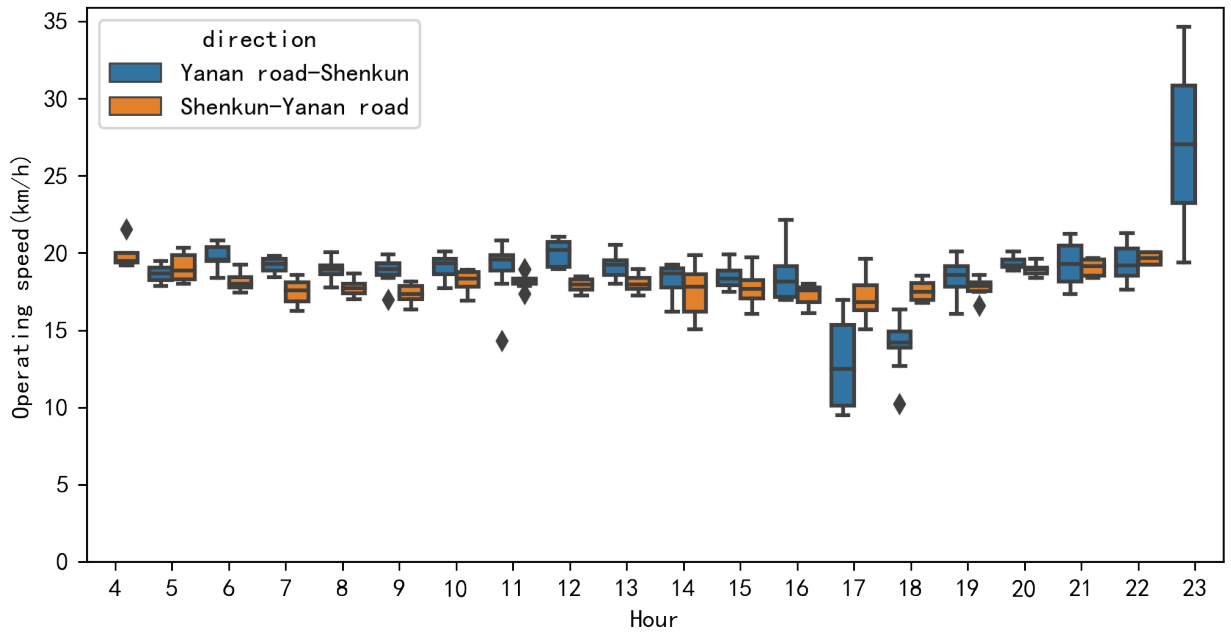
[14]:
```
#Remove the data with abnormal speed
#Vehicle speed units converted to km/h
onewaytime['speed'] = (lineshp.length/onewaytime['duration'])*3.6
onewaytime = onewaytime[onewaytime['speed']<=60]
```

[15]:
```
## Travel speed distribution
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['font.serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus']=False
fig     = plt.figure(1,(8,4),dpi = 250)
ax1      = plt.subplot(111)
sns.boxplot(x = 'shour',y = 'speed',hue = 'direction',data = onewaytime)
plt.ylabel('Operating speed(km/h)')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Hour')
plt.ylim(0)
plt.show()
```



10 Mapmatching of taxi GPS data

This example uses TransBigData+leuvenmapmatching to achieve the road network matching of taxi GPS data

Preparing trajectory data

[1]:
```
import transbigdata as tbd
import pandas as pd

#Read_data
data = pd.read_csv('data/TaxiData-Sample.csv',header = None)
data.columns = ['VehicleNum','Time','Lng','Lat','OpenStatus','Speed']
```

[2]:
```
from leuvenmapmatching.matcher.distance import DistanceMatcher
from leuvenmapmatching.map.inmem import InMemMap
from leuvenmapmatching import visualization as mmviz
```

[3]:
```
#obtain OD from trajectory data
oddata = tbd.taxigps_to_od(data,col = ['VehicleNum','Time','Lng','Lat','OpenStatus'])
#extract deliver and idle trip trajectories
data_deliver,data_idle = tbd.taxigps_traj_point(data,oddata,col=['VehicleNum', 'Time',
↪'Lng', 'Lat', 'OpenStatus'])
```

Modeling road network

You can download the road network from openstreetmap.

[4]:
```
# obtain road network
import osmnx as ox
bounds = [113.75, 22.4, 114.62, 22.86]
north, south, east, west = bounds[3], bounds[1], bounds[2], bounds[0]
G = ox.graph_from_bbox(north, south, east, west, network_type='drive')
```

[5]:
```
#save road network
ox.save_graphml(G,'shenzhen.graphml')
```

If you alread have the road network data…

[6]:
```
# Read the road network
import osmnx as ox
filepath = "shenzhen.graphml"
G = ox.load_graphml(filepath)
```

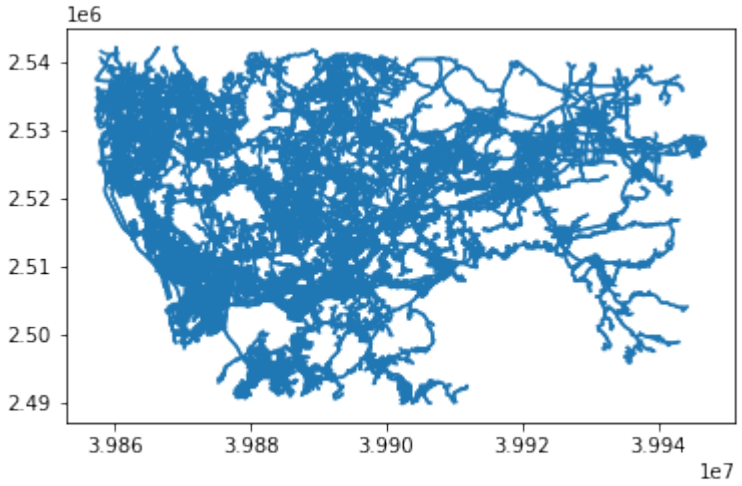
[7]:
```
#Obtain the road GeoDataFrame and road centroid
nodes, edges = ox.graph_to_gdfs(G, nodes=True, edges=True)
edges['lon'] = edges.centroid.x
edges['lat'] = edges.centroid.y
```

```
/var/folders/b0/q8rx9fj965b5p7yqq8zhvdx80000gn/T/ipykernel_80320/1542180003.py:3:␣
↪UserWarning: Geometry is in a geographic CRS. Results from 'centroid' are likely␣
↪incorrect. Use 'GeoSeries.to_crs()' to re-project geometries to a projected CRS before␣
↪this operation.

  edges['lon'] = edges.centroid.x
/var/folders/b0/q8rx9fj965b5p7yqq8zhvdx80000gn/T/ipykernel_80320/1542180003.py:4:␣
↪UserWarning: Geometry is in a geographic CRS. Results from 'centroid' are likely␣
↪incorrect. Use 'GeoSeries.to_crs()' to re-project geometries to a projected CRS before␣
↪this operation.

  edges['lat'] = edges.centroid.y
```

[8]:
```
#convert to projection coordinates
G_p = ox.project_graph(G, to_crs=2416)
nodes_p, edges_p = ox.graph_to_gdfs(G_p, nodes=True, edges=True)
```

[9]:
```
edges_p.plot()
```

[9]:
```
<AxesSubplot:>
```



[11]:
```
# create road network
map_con = InMemMap(name='pNEUMA', use_latlon=False) # , use_rtree=True, index_edges=True)

for node_id, row in nodes_p.iterrows():
    map_con.add_node(node_id, (row['y'], row['x']))
for node_id_1, node_id_2, _ in G_p.edges:
    map_con.add_edge(node_id_1, node_id_2)
```

Mapmatching

[12]:
```
# Extract one of the trajectory using transbigdata
import geopandas as gpd
tmp_gdf = data_deliver[data_deliver['ID'] == 27].sort_values(by='Time')
# trajectory densify
tmp_gdf = tbd.traj_densify(
    tmp_gdf, col=['ID', 'Time', 'Lng', 'Lat'], timegap=15)
# convert coordinate
tmp_gdf['geometry'] = gpd.points_from_xy(tmp_gdf['Lng'], tmp_gdf['Lat'])
tmp_gdf = gpd.GeoDataFrame(tmp_gdf)
tmp_gdf.crs = {'init': 'epsg:4326'}
tmp_gdf = tmp_gdf.to_crs(2416)
# obtain trajectoies
path = list(zip(tmp_gdf.geometry.y, tmp_gdf.geometry.x))
# create mapmatcher
matcher = DistanceMatcher(map_con,
                          max_dist=500,
                          max_dist_init=170,
                          min_prob_norm=0.0001,
                          non_emitting_length_factor=0.95,
                          obs_noise=50,
                          obs_noise_ne=50,
                          dist_noise=50,
                          max_lattice_width=20,
                          non_emitting_states=True)
```

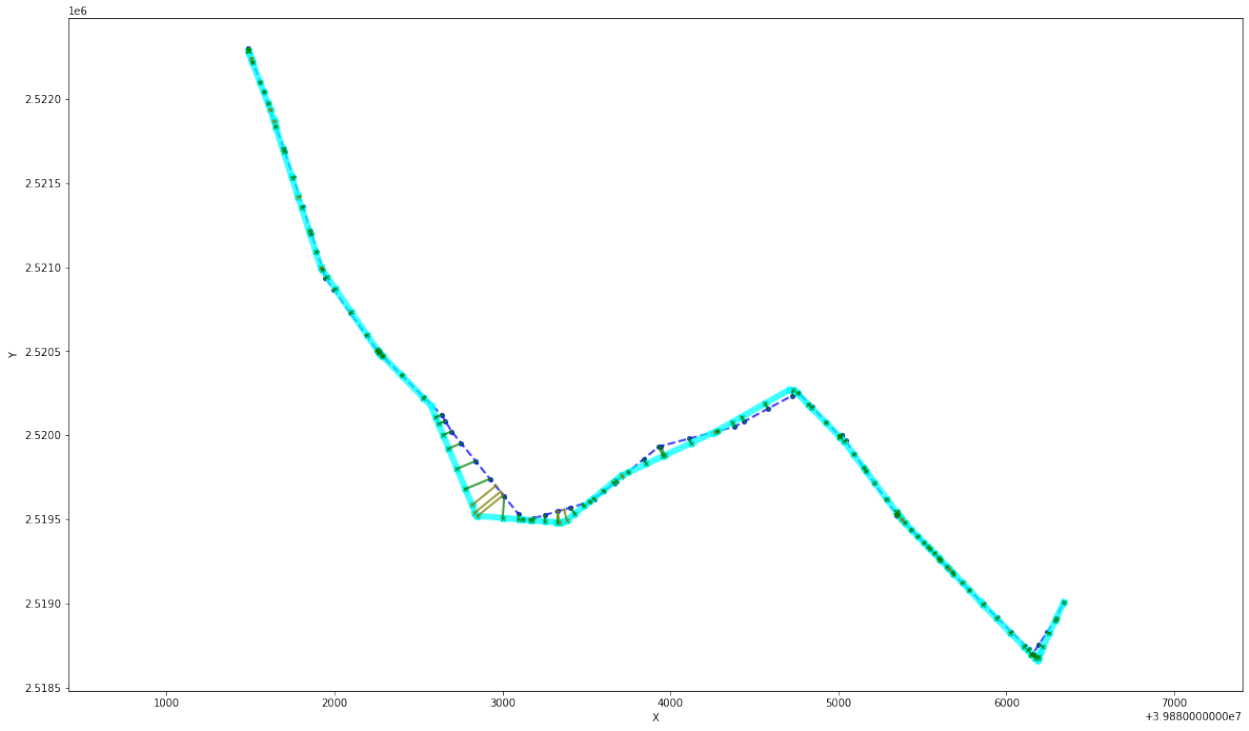
(continues on next page)

(continued from previous page)

```
# mapmatching
states, _ = matcher.match(path, unique=False)
# plot the result
mmviz.plot_map(map_con, matcher=matcher,
               show_labels=False, show_matching=True,  # show_graph=True,
               filename=None)
```

```
/opt/anaconda3/lib/python3.8/site-packages/pyproj/crs/crs.py:131: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  in_crs_string = _prepare_from_proj_string(in_crs_string)
Searching closeby nodes with linear search, use an index and set max_dist
/opt/anaconda3/lib/python3.8/site-packages/leuvenmapmatching/visualization.py:194:␣
↪UserWarning: linestyle is redundantly defined by the 'linestyle' keyword argument and␣
↪the fmt string "o-" (-> linestyle='-'). The keyword argument will take precedence.
  ax.plot(px, py, 'o-', linewidth=linewidth, markersize=linewidth * 2, alpha=0.75,
```

[12]:
```
(<Figure size 1440x846.228 with 1 Axes>, <AxesSubplot:xlabel='X', ylabel='Y'>)
```



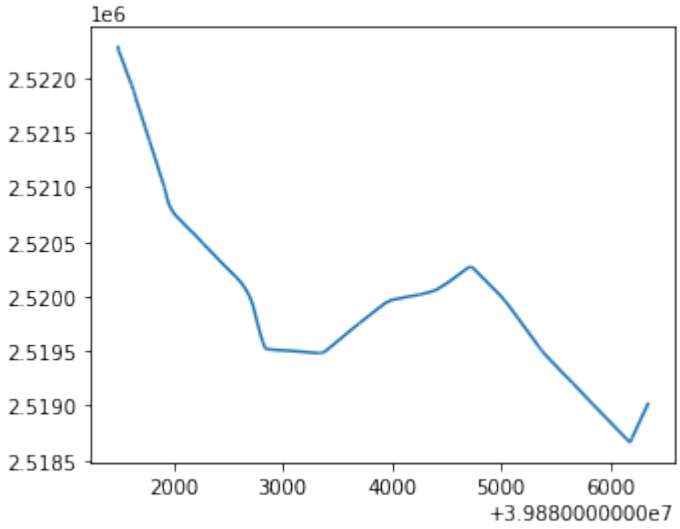
[14]:
```
#Obtain the path GeoDataFrame
pathdf = pd.DataFrame(matcher.path_pred_onlynodes,columns = ['u'])
pathdf['v'] = pathdf['u'].shift(-1)
pathdf = pathdf[-pathdf['v'].isnull()]
pathgdf = pd.merge(pathdf,edges_p.reset_index())
pathgdf = gpd.GeoDataFrame(pathgdf)
pathgdf.plot()
pathgdf.crs = {'init':'epsg:2416'}
```

(continues on next page)

(continued from previous page)

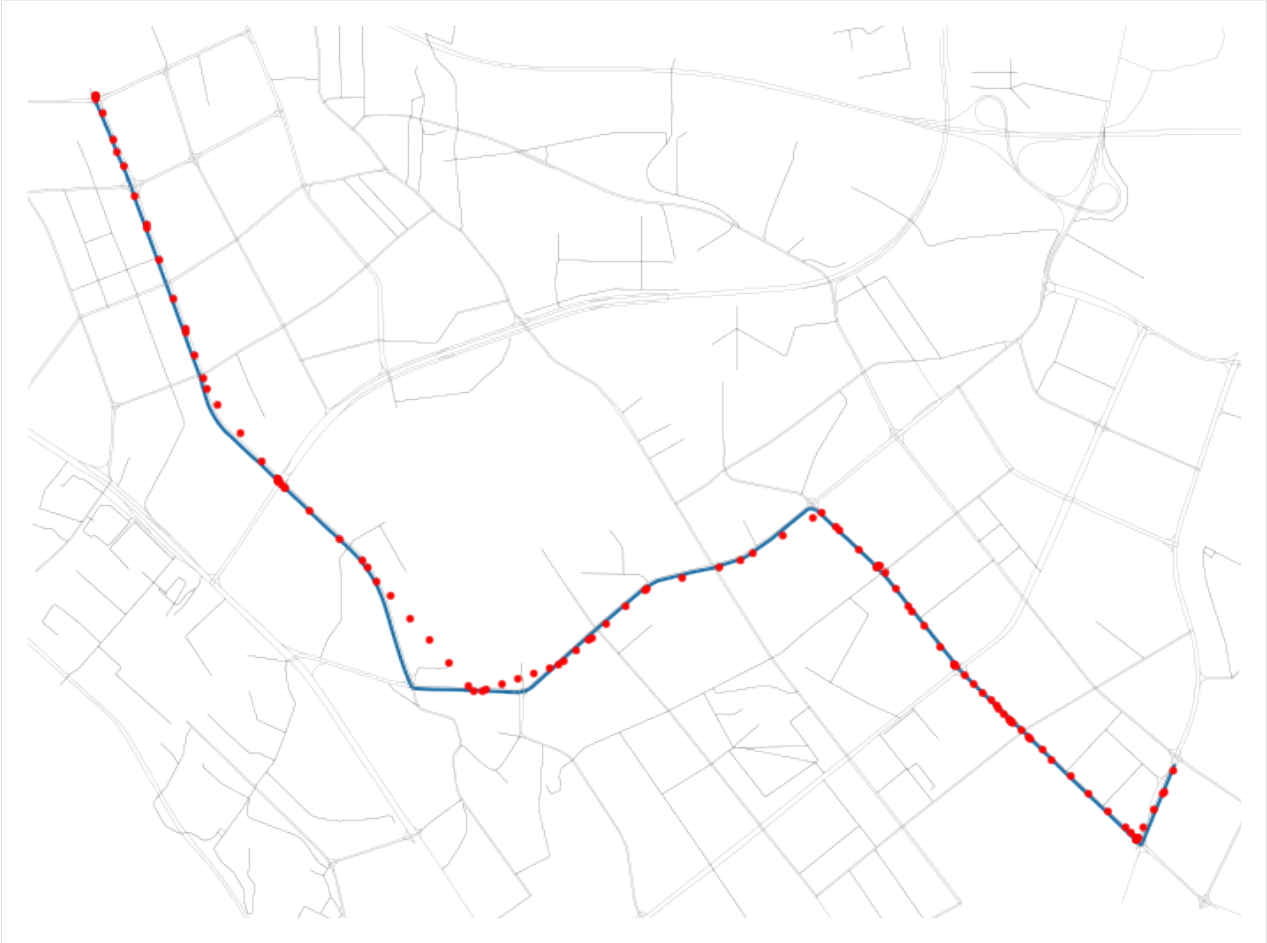
```
pathgdf_4326 = pathgdf.to_crs(4326)
```

```
/opt/anaconda3/lib/python3.8/site-packages/pyproj/crs/crs.py:131: FutureWarning: '+init=
↪<authority>:<code>' syntax is deprecated. '<authority>:<code>' is the preferred␣
↪initialization method. When making the change, be mindful of axis order changes: https:
↪//pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  in_crs_string = _prepare_from_proj_string(in_crs_string)
```



Visualization

[15]:
```
#Visualize with road network
import matplotlib.pyplot as plt

fig     = plt.figure(1,(8,8),dpi = 100)
ax      = plt.subplot(111)
plt.sca(ax)
fig.tight_layout(rect = (0.05,0.1,1,0.9))
#visualization bounds
bounds = pathgdf_4326.unary_union.bounds
gap = 0.003
bounds = [bounds[0]-gap,bounds[1]-gap,bounds[2]+gap,bounds[3]+gap]
#plot the matched path
pathgdf_4326.plot(ax = ax,zorder = 1)
#plot the road network geometry
tbd.clean_outofbounds(edges,bounds,col = ['lon','lat']).plot(ax = ax,color = '#333',lw =␣
↪0.1)
#plot the trajectory points
tmp_gdf.to_crs(4326).plot(ax = ax,color = 'r',markersize = 5,zorder = 2)

plt.axis('off')
plt.xlim(bounds[0],bounds[2])
plt.ylim(bounds[1],bounds[3])
plt.show()
```



CHAPTER

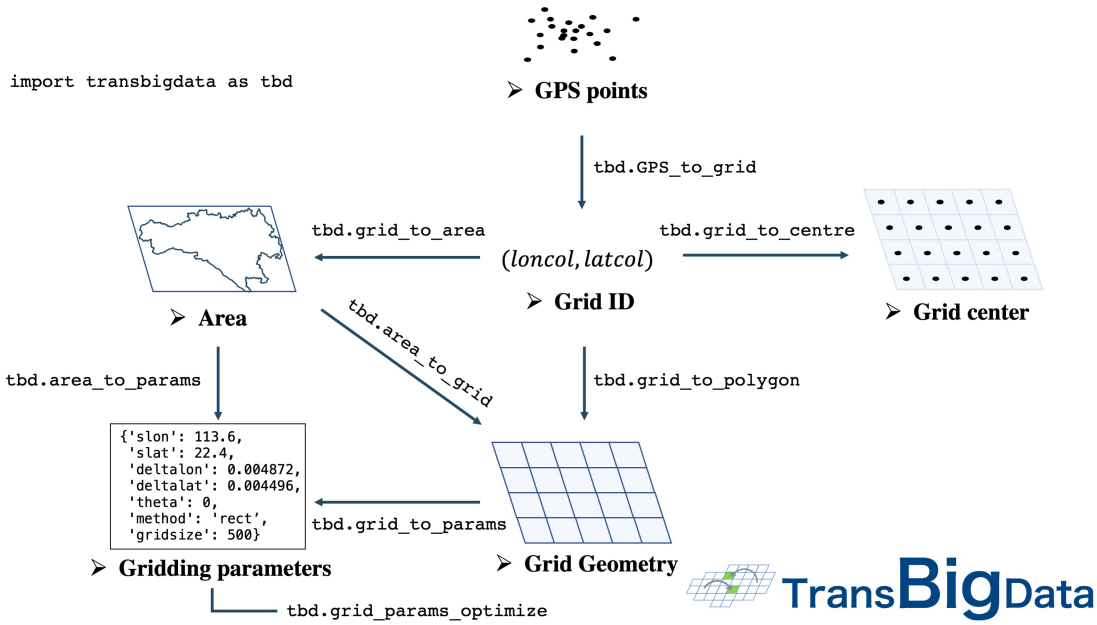
FIVE

CORE METHODS

5.1 Data Gridding

area_to_grid(location[, accuracy, method, ...])	Generate the rectangular grids in the bounds or shape
area_to_params(location[, accuracy, method])	Generate gridding params
GPS_to_grid(lon, lat, params)	Match the GPS data to the grids.
grid_to_centre(gridid, params)	The center location of the grid.
grid_to_polygon(gridid, params)	Generate the geometry column based on the grid ID.
grid_to_area(data, shape, params[, col])	Input the two columns of grid ID, the geographic polygon and gridding paramters.
grid_to_params(grid)	Regenerate gridding params from grid.
grid_params_optimize(data, initialparams[, ...])	Optimize the grid params
geohash_encode(lon, lat[, precision])	Input latitude and longitude and precision, and encode geohash code
geohash_decode(geohash)	Decode geohash code
geohash_togrid(geohash)	Input geohash code to generate geohash grid cell

5.1.1 Gridding Framework



transbigdata.**area_to_grid**(*location*, *accuracy=500*, *method='rect'*, *params='auto'*)

Generate the rectangular grids in the bounds or shape

Parameters

- **location** (*bounds(List) or shape(GeoDataFrame)*) – Where to generate grids. If bounds, [lon1, lat1, lon2, lat2](WGS84), where lon1 , lat1 are the lower-left coordinates, lon2 , lat2 are the upper-right coordinates If shape, it should be GeoDataFrame
- **accuracy** (*number*) – Grid size (meter)
- **method** (*str*) – rect, tri or hexa
- **params** (*list or dict*) – Gridding parameters. See https://transbigdata.readthedocs.io/en/latest/grids.html for detail information about gridding parameters. When Gridding parameters is given, accuracy will not be used.

Returns

- **grid** (*GeoDataFrame*) – Grid GeoDataFrame, LONCOL and LATCOL are the index of grids, HBLON and HBLAT are the center of the grids
- **params** (*list or dict*) – Gridding parameters. See https://transbigdata.readthedocs.io/en/latest/grids.html for detail information about gridding parameters.

transbigdata.**area_to_params**(*location*, *accuracy=500*, *method='rect'*)

Generate gridding params

Parameters

- **location** (*bounds(List) or shape(GeoDataFrame)*) – Where to generate grids. If bounds, [lon1, lat1, lon2, lat2](WGS84), where lon1 , lat1 are the lower-left coordinates, lon2 , lat2 are the upper-right coordinates If shape, it should be GeoDataFrame

- **accuracy** (*number*) – Grid size (meter)
- **method** (*str*) – rect, tri or hexa

Returns

params – Gridding parameters. See https://transbigdata.readthedocs.io/en/latest/grids.html for detail information about gridding parameters.

Return type

list or dict

transbigdata.**GPS_to_grid**(*lon*, *lat*, *params*)

Match the GPS data to the grids. The input is the columns of longitude, latitude, and the grids parameter. The output is the grid ID.

Parameters

- **lon** (*Series*) – The column of longitude
- **lat** (*Series*) – The column of latitude
- **params** (*list or dict*) – Gridding parameters. See https://transbigdata.readthedocs.io/en/latest/grids.html for detail information about gridding parameters.

Returns

- *Rectangle grids*
- **[LONCOL,LATCOL]** (*list*) – The two columns LONCOL and LATCOL together can specify a grid.
- *Triangle and Hexagon grids*
- **[loncol_1,loncol_2,loncol_3]** (*list*) – The index of the grid latitude. The two columns LONCOL and LATCOL together can specify a grid.

transbigdata.**grid_to_centre**(*gridid*, *params*)

The center location of the grid. The input is the grid ID and parameters, the output is the grid center location.

Parameters

- **gridid** (*list*) – if *Rectangle grids* [LONCOL,LATCOL] : Series

  The two columns LONCOL and LATCOL together can specify a grid.

  if *Triangle and Hexagon grids* [loncol_1,loncol_2,loncol_3] : Series

  The index of the grid latitude. The two columns LONCOL and LATCOL together can specify a grid.
- **params** (*list or dict*) – Gridding parameters. See https://transbigdata.readthedocs.io/en/latest/grids.html for detail information about gridding parameters.

Returns

- **HBLON** (*Series*) – The longitude of the grid center
- **HBLAT** (*Series*) – The latitude of the grid center

transbigdata.**grid_to_polygon**(*gridid*, *params*)

Generate the geometry column based on the grid ID. The input is the grid ID, the output is the geometry. Support rectangle, triangle and hexagon grids

Parameters

- **gridid** (*list*) – if *Rectangle grids* [LONCOL,LATCOL] : Series

The two columns LONCOL and LATCOL together can specify a grid.

if *Triangle and Hexagon grids* [loncol_1,loncol_2,loncol_3] : Series

The index of the grid latitude. The two columns LONCOL and LATCOL together can specify a grid.

- **params** (*list or dict*) – Gridding parameters. See https://transbigdata.readthedocs.io/en/latest/grids.html for detail information about gridding parameters.

Returns
geometry – The column of grid geographic polygon

Return type
Series

transbigdata.**grid_to_area**(*data*, *shape*, *params*, *col=['LONCOL', 'LATCOL']*)

Input the two columns of grid ID, the geographic polygon and gridding paramters. The output is the grid.

Parameters

- **data** (*DataFrame*) – Data, with two columns of grid ID
- **shape** (*GeoDataFrame*) – Geographic polygon
- **params** (*list or dict*) – Gridding parameters. See https://transbigdata.readthedocs.io/en/latest/grids.html for detail information about gridding parameters.
- **col** (*List*) – Column names [LONCOL,LATCOL] for rect grids or [loncol_1,loncol_2,loncol_3] for tri and hexa grids

Returns
data1 – Data gridding and mapping to the corresponding geographic polygon

Return type
DataFrame

transbigdata.**grid_to_params**(*grid*)

Regenerate gridding params from grid. Only support rect grids now.

Parameters
grid (*GeoDataFrame*) – grids generated by transbigdata

Returns
params – Gridding parameters. See https://transbigdata.readthedocs.io/en/latest/grids.html for detail information about gridding parameters.

Return type
list or dict

transbigdata.**grid_params_optimize**(*data*, *initialparams*, *col=['uid', 'lon', 'lat']*, *optmethod='centerdist'*, *printlog=False*, *sample=0*, *pop=15*, *max_iter=50*, *w=0.1*, *c1=0.5*, *c2=0.5*)

Optimize the grid params

Parameters

- **data** (*DataFrame*) – Trajectory data
- **initialparams** (*List*) – Initial griding params
- **col** (*List*) – Column names [uid,lon,lat]
- **optmethod** (*str*) – The method to optimize: centerdist, gini, gridscount

- **printlog** (*bool*) – Whether to print detail result
- **sample** (*int*) – Sample the data as input, if 0 it will not perform sampling
- **pop** – Params in PSO from scikit-opt
- **max_iter** – Params in PSO from scikit-opt
- **w** – Params in PSO from scikit-opt
- **c1** – Params in PSO from scikit-opt
- **c2** – Params in PSO from scikit-opt

Returns
params_optimized – Optimized params

Return type
List

5.1.2 geohash encoding

Geohash is a public geocoding system that encodes latitude and longitude geographic locations into strings of letters and numbers, which can also be decoded back to latitude and longitude. Each string represents a grid number, and the longer the length of the string, the higher the precision. According to wiki <https://en.wikipedia.org/wiki/Geohash>, the table of Geohash string lengths corresponding to precision is as follows.

geohash length(precision)	lat bits	lng bits	lat error	lng error	km error
1	2	3	±23	±23	±2500
2	5	5	±2.8	±5.6	±630
3	7	8	±0.70	±0.70	±78
4	10	10	±0.087	±0.18	±20
5	12	13	±0.022	±0.022	±2.4
6	15	15	±0.0027	±0.0055	±0.61
7	17	18	±0.00068	±0.00068	±0.076
8	20	20	±0.000085	±0.00017	±0.019

TransBigData also provides the function based on Geohash, the three functions are as follows:

`transbigdata.`**geohash_encode**(*lon*, *lat*, *precision=12*)

Input latitude and longitude and precision, and encode geohash code

Parameters

- **lon** (*Series*) – longitude Series
- **lat** (*Series*) – latitude Series
- **precision** (*number*) – geohash precision

Returns
geohash – encoded geohash Series

Return type
Series

`transbigdata.`**geohash_decode**(*geohash*)

Decode geohash code

Parameters
geohash (*Series*) – encoded geohash Series

Returns
- **lon** (*Series*) – decoded longitude Series
- **lat** (*Series*) – decoded latitude Series

transbigdata.**geohash_togrid**(*geohash*)

Input geohash code to generate geohash grid cell

Parameters
geohash (*Series*) – encoded geohash Series

Returns
poly – grid cell polygon for geohash

Return type
Series

Compared to the rectangular grid processing method provided in the TransBigData package, geohash is slower and does not provide a freely defined grid size. The following example shows how to use these three functions to utilize the geohash encoding, decoding, and the visualization

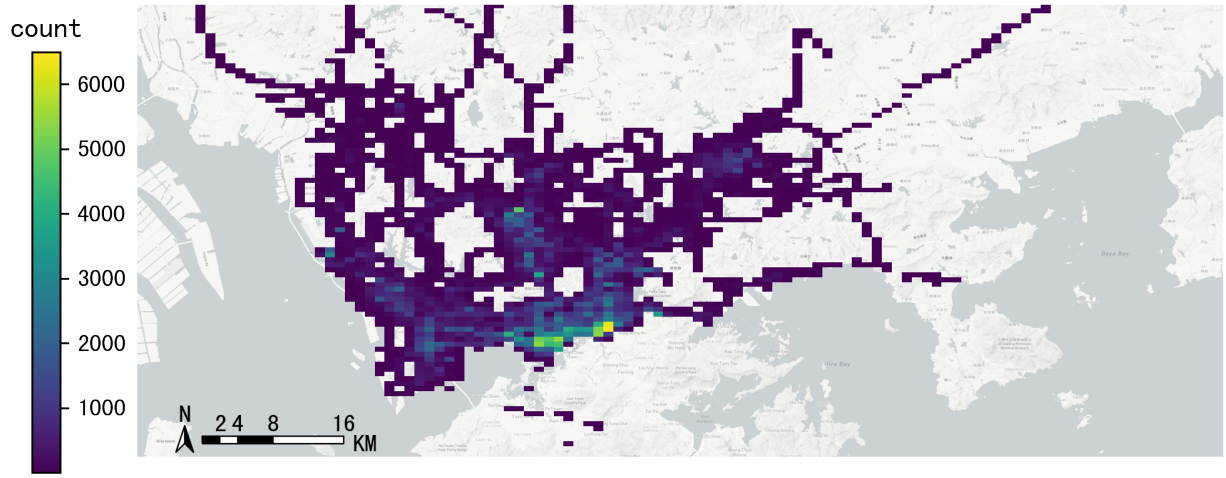
```
import transbigdata as tbd
import pandas as pd
import geopandas as gpd
#read data
data = pd.read_csv('TaxiData-Sample.csv',header = None)
data.columns = ['VehicleNum','time','slon','slat','OpenStatus','Speed']
```

```
#encode geohash
data['geohash'] = tbd.geohash_encode(data['slon'],data['slat'],precision=6)
data['geohash']
```

```
0         ws0btw
1         ws0btz
2         ws0btz
3         ws0btz
4         ws0by4
           ...
544994    ws131q
544995    ws1313
544996    ws131f
544997    ws1361
544998    ws10tq
Name: geohash, Length: 544999, dtype: object
```

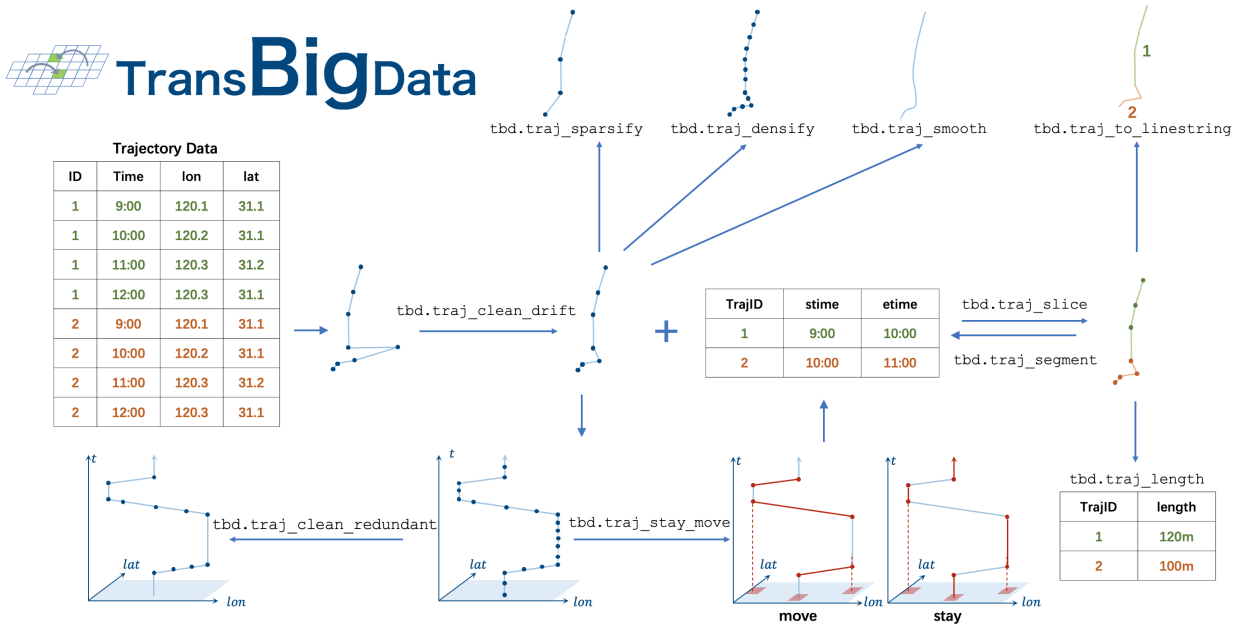
```
#Aggregate
dataagg = data.groupby(['geohash'])['VehicleNum'].count().reset_index()
dataagg['lon_geohash'],dataagg['lat_geohash'] = tbd.geohash_decode(dataagg['geohash'])
dataagg['geometry'] = tbd.geohash_togrid(dataagg['geohash'])
dataagg = gpd.GeoDataFrame(dataagg)
dataagg
```

```
bounds = [113.6,22.4,114.8,22.9]
import matplotlib.pyplot as plt
import plot_map
fig =plt.figure(1,(8,8),dpi=280)
ax =plt.subplot(111)
plt.sca(ax)
tbd.plot_map(plt,bounds,zoom = 12,style = 4)
cax = plt.axes([0.05, 0.33, 0.02, 0.3])
plt.title('count')
plt.sca(ax)
dataagg.plot(ax = ax,column = 'VehicleNum',cax = cax,legend = True)
tbd.plotscale(ax,bounds = bounds,textsize = 10,compasssize = 1,accuracy = 2000,rect = [0.
↪06,0.03],zorder = 10)
plt.axis('off')
plt.xlim(bounds[0],bounds[2])
plt.ylim(bounds[1],bounds[3])
plt.show()
```



5.2 Trajectory Processing

traj_clean_drift(data[, col, method, ...])	Delete the drift in the trajectory data.
traj_clean_redundant(data[, col])	Delete the data with the same information as the data before and after to reduce the amount of data.
traj_slice(traj_data, slice_data[, ...])	Slice the trajectory data according to the slice data.
traj_smooth(data[, col, proj, ...])	Smooth Trajectory Using Kalman Filter.
traj_segment(data[, groupby_col, retain_col])	Segment the trajectory in order and return the starting and ending information of each segment.
traj_densify(data[, col, timegap])	Trajectory densification, ensure that there is a trajectory point each timegap seconds
traj_sparsify(data[, col, timegap, method])	Trajectory sparsify.
traj_stay_move(data, params[, col, activitytime])	Input trajectory data and gridding parameters, identify stay and move
traj_to_linestring(traj_points[, col, timecol])	Input trajectory, generate GeoDataFrame
traj_mapmatch(traj, G[, col])	Nearest map matching: Find the nearest point on the road network for each trajectory point.
traj_length(move_points[, col, method])	Calculate Trajectory Length.



transbigdata.**traj_clean_drift**(*data*, *col=['VehicleNum', 'Time', 'Lng', 'Lat']*, *method='twoside'*, *speedlimit=80*, *dislimit=1000*, *anglelimit=30*)

Delete the drift in the trajectory data. The drift is defined as the data with a speed greater than the speed limit or the distance between the current point and the next point is greater than the distance limit or the angle between the current point, the previous point, and the next point is smaller than the angle limit. The speed limit is 80km/h by default, and the distance limit is 1000m by default. The method of cleaning drift data is divided into two methods: 'oneside' and 'twoside'. The 'oneside' method is to consider the speed of the current point and the next point, and the 'twoside' method is to consider the speed of the current point, the previous point, and the next point.

Parameters

- **data** (*DataFrame*) – Data
- **col** (*List*) – Column names, in the order of ['VehicleNum', 'Time', 'Lng', 'Lat']
- **method** (*string*) – Method of cleaning drift data, including 'oneside' and 'twoside'
- **speedlimit** (*number*) – Speed limitation
- **dislimit** (*number*) – Distance limit
- **anglelimit** (*number*) – Angle limit

Returns
data1 – Cleaned data

Return type
DataFrame

transbigdata.**traj_clean_redundant**(*data*, *col=['VehicleNum', 'Time', 'Lng', 'Lat']*)

Delete the data with the same information as the data before and after to reduce the amount of data. For example, if several consecutive data of an individual have the same information except for the time, only the first and last two data can be kept

Parameters

- **data** (*DataFrame*) – Data
- **col** (*List*) – The column name, in the order of ['Vehicleid, Time']. It will sort by time, and then determine the information of other columns besides the time

Returns
data1 – Cleaned data

Return type
DataFrame

transbigdata.**traj_slice**(*traj_data*, *slice_data*, *traj_col=['vid', 'time']*, *slice_col=['vid', 'stime', 'etime', 'tripid']*)

Slice the trajectory data according to the slice data. This method extracts data from a given set of trajectory data(traj_data) based on a specified time period(slice_data).

Parameters

- **traj_data** (*DataFrame*) – Trajectory data, containing the trajectory of each vehicle
- **slice_data** (*DataFrame*) – Slice data, containing the start time, end time and vehicleid of each slice
- **traj_col** (*List*) – The column name of trajectory data, in the sequence of [VehicleNum, Time]
- **slice_col** (*List*) – The column name of slice data, in the sequence of [VehicleNum_slice, Stime, Etime, SliceId]

Returns
data_sliced – The sliced trajectory data

Return type
DataFrame

Example

```
>>> tbd.traj_slice(GPSData, move, traj_col=['vid', 'time'], slice_col = ['vid',
↪'stime','etime','tripid'])
```

transbigdata.**traj_smooth**(*data*, *col=['id', 'time', 'lon', 'lat']*, *proj=False*, *process_noise_std=0.5*, *measurement_noise_std=1*)

Smooth Trajectory Using Kalman Filter.

Parameters

- **data** (*DataFrame*) – Trajectory data
- **col** (*list*) – Column names of the trajectory data
- **proj** (*bool*) – Whether to perform equidistant projection
- **process_noise_std** (*float*) – Standard deviation of the process noise
- **measurement_noise_std** (*float*) – Standard deviation of the measurement noise

Returns

data – Smoothed trajectory data

Return type

DataFrame

transbigdata.**traj_segment**(*data*, *groupby_col=['id', 'moveid']*, *retain_col=['time', 'lon', 'lat']*)

Segment the trajectory in order and return the starting and ending information of each segment. This function can segment GPS trajectory data, calculate the start and end information of each segment, and store the results in a DataFrame object. The input of this function includes a pandas DataFrame object containing GPS trajectory data, field names for grouping, and field names to be retained. The output is a pandas DataFrame object containing the starting and ending information of each segment, where each row represents a trajectory segment.

Parameters

- **data** (*DataFrame*) – The trajectory data needs to be sorted beforehand.
- **groupby_col** (*List*) – A list of strings specifying the groupby fields to be used for segmentation.
- **retain_col** (*List*) – A list of strings specifying the fields to be retained.

Returns

data – Containing the starting and ending information of each segment, where each row represents a trajectory segment.

Return type

DataFrame

transbigdata.**traj_densify**(*data*, *col=['Vehicleid', 'Time', 'Lng', 'Lat']*, *timegap=15*)

Trajectory densification, ensure that there is a trajectory point each timegap seconds

Parameters

- **data** (*DataFrame*) – Data
- **col** (*List*) – The column name, in the sequence of [Vehicleid, Time, lng, lat]
- **timegap** (*number*) – The sampling interval (second)

Returns

data1 – The processed data

Return type
DataFrame

transbigdata.**traj_sparsify**(*data*, *col=['Vehicleid', 'Time', 'Lng', 'Lat']*, *timegap=15*, *method='subsample'*)

Trajectory sparsify. When the sampling frequency of trajectory data is too high, the amount of data is too large, which is not convenient for the analysis of some studies that require less data frequency. This function can expand the sampling interval and reduce the amount of data.

Parameters

- **data** (*DataFrame*) – Data
- **col** (*List*) – The column name, in the sequence of [Vehicleid, Time, lng, lat]
- **timegap** (*number*) – Time gap between trajectory point
- **method** (*str*) – 'interpolate' or 'subsample'

Returns
data1 – Sparsified trajectory data

Return type
DataFrame

transbigdata.**traj_stay_move**(*data*, *params*, *col=['ID', 'dataTime', 'longitude', 'latitude']*, *activitytime=1800*)

Input trajectory data and gridding parameters, identify stay and move

Parameters

- **data** (*DataFrame*) – trajectory data
- **params** (*List*) – gridding parameters
- **col** (*List*) – The column name, in the order of ['ID','dataTime','longitude', 'latitude']
- **activitytime** (*Number*) – How much time to regard as activity

Returns

- **stay** (*DataFrame*) – stay information
- **move** (*DataFrame*) – move information

transbigdata.**traj_to_linestring**(*traj_points*, *col=['Lng', 'Lat', 'ID']*, *timecol=None*)

Input trajectory, generate GeoDataFrame

Parameters

- **traj_points** (*DataFrame*) – trajectory data
- **col** (*List*) – The column name, in the sequence of [lng, lat,trajectoryid]
- **timecol** (*str(Optional)*) – Optional, the column name of the time column. If given, the geojson with [longitude, latitude, altitude, time] in returns can be put into the Kepler to visualize the trajectory

Returns
traj – Generated trajectory

Return type
GeoDataFrame

transbigdata.**traj_mapmatch**(*traj*, *G*, *col=['lon', 'lat']*)

Nearest map matching: Find the nearest point on the road network for each trajectory point. When conducting nearest neighbor matching, we need to find the closest road segment on the road network for each trajectory point, and match the trajectory point to that segment. In practice, we can first extract the nodes of the road segments to form a set of points (i.e., extracting each coordinate point from each LineString in the geometry column), then calculate the nearest distance between the trajectory point and this set of points, and finally match the trajectory point to the road segment where the nearest distance's node is located. This process effectively transforms the problem of matching points to lines into a problem of matching points to points.

Parameters

- **traj** (*DataFrame*) – The trajectory point data set to be matched.
- **G** (*networkx multidigraph*) – The road network used for matching, created by osmnx.
- **col** (*list*) – The name of the longitude and latitude columns in the trajectory point data set.

Returns

traj_matched – The trajectory point data set after matching.

Return type

DataFrame

transbigdata.**traj_length**(*move_points*, *col=['lon', 'lat', 'moveid']*, *method='Haversine'*)

Calculate Trajectory Length. Input the trajectory point data and calculate the length of each trajectory in meters.

Parameters

- **move_points** (*DataFrame*) – Trajectory point data, which includes trajectory id, longitude, latitude, etc. Different trajectories are distinguished by trajectory id, and trajectory points are arranged in time order.
- **col** (*list*) – Column names of the trajectory point data, in the order of [longitude, latitude, trajectory id]
- **method** (*str*) – The method of calculating the trajectory length, optional 'Haversine' or 'Project', default is 'Haversine' using the haversine formula to calculate spherical distance, 'Project' transforms the data into a projected coordinate system to calculate plane distance.

Returns

move_trajs – Trajectory length data, including two columns of trajectory id and trajectory length, the unit is meters.

Return type

DataFrame

CHAPTER

SIX

GENERAL METHODS

6.1 Data Quality

data_summary(data[, col, ...])	Output the general information of the dataset.
sample_duration(data[, col])	Calculate the data sampling interval.

transbigdata.**data_summary**(*data*, *col=['Vehicleid', 'Time']*, *show_sample_duration=False*, *roundnum=4*)

Output the general information of the dataset.

Parameters

- **data** (*DataFrame*) – The trajectory points data
- **col** (*List*) – The column name, in the order of ['Vehicleid', 'Time']
- **show_sample_duration** (*bool*) – Whether to output individual sampling interval
- **roundnum** (*number*) – Number of decimal places

transbigdata.**sample_duration**(*data*, *col=['Vehicleid', 'Time']*)

Calculate the data sampling interval.

Parameters

- **data** (*DataFrame*) – Data
- **col** (*List*) – The column name, in the order of ['Vehicleid', 'Time']

Returns

sample_duration – A Series with the column name duration, the content is the sampling interval of the data, in seconds

Return type

DataFrame

6.2 Data Preprocess

clean_outofbounds(data, bounds[, col])	The input is the latitude and longitude coordinates of the lower left and upper right of the study area and exclude data that are outside the study area
clean_outofshape(data, shape[, col, accuracy])	Input the GeoDataFrame of the study area and exclude the data beyond the study area
id_reindex(data, col[, new, timegap, ...])	Renumber the ID columns of the data
id_reindex_disgap(data[, col, disgap, suffix])	Renumber the ID columns of the dataIf two adjacent records exceed the distance, the number is the new ID

transbigdata.**clean_outofbounds**(*data*, *bounds*, *col=['Lng', 'Lat']*)

The input is the latitude and longitude coordinates of the lower left and upper right of the study area and exclude data that are outside the study area

Parameters

- **data** (*DataFrame*) – Data
- **bounds** (*List*) – Latitude and longitude of the lower left and upper right of the study area, in the order of [lon1, lat1, lon2, lat2]
- **col** (*List*) – Column name of longitude and latitude

Returns
data1 – Data within the scope of the study

Return type
DataFrame

transbigdata.**clean_outofshape**(*data*, *shape*, *col=['Lng', 'Lat']*, *accuracy=500*)

Input the GeoDataFrame of the study area and exclude the data beyond the study area

Parameters

- **data** (*DataFrame*) – Data
- **shape** (*GeoDataFrame*) – The GeoDataFrame of the study area
- **col** (*List*) – Column name of longitude and latitude
- **accuracy** (*number*) – The size of grid. The principle is to do the data gridding first and then do the data cleaning. The smaller the size is, the higher accuracy it has

Returns
data1 – Data within the scope of the study

Return type
DataFrame

transbigdata.**id_reindex**(*data*, *col*, *new=False*, *timegap=None*, *timecol=None*, *suffix='_new'*, *sample=None*)

Renumber the ID columns of the data

Parameters

- **data** (*DataFrame*) – Data
- **col** (*str*) – Name of the ID column to be re-indexed
- **new** (*bool*) – False: the new number of the same ID will be the same index; True: according to the order of the table, the origin ID appears again with different index

- **timegap** (*number*) – If an individual does not appear for a period of time (timegap is the time threshold), it is numbered as a new individual. This parameter should be set with timecol to take effect.
- **timecol** (*str*) – The column name of time, it should be set with timegap to take effect
- **suffix** (*str*) – The suffix of the new column. When set to False, the former column will be replaced
- **sample** (*int (optional)*) – To desampling the data

Returns
data1 – Renumbered data

Return type
DataFrame

transbigdata.**id_reindex_disgap**(*data*, *col=['uid', 'lon', 'lat']*, *disgap=1000*, *suffix='_new'*)

Renumber the ID columns of the dataIf two adjacent records exceed the distance, the number is the new ID

Parameters

- **data** (*DataFrame*) – Data
- **col** (*str*) – Name of the ID column to be re-indexed
- **disgap** (*number*) – If two adjacent records exceed this distance, the number is the new ID
- **suffix** (*str*) – The suffix of the new column. When set to False, the former column will be replaced

Returns
data1 – Renumbered data

Return type
DataFrame

6.3 Data Acquisition

getbusdata(city, keywords[, accurate, timeout])	Obtain the geographic information of the bus station and bus line from the map service (Only in China)
getadmin(keyword, ak[, jscode, ...])	Input the keyword and the Amap ak.
get_isochrone_amap(lon, lat, reachtime, ak)	Obtain the isochrone from Amap reachcricle
get_isochrone_mapbox(lon, lat, reachtime[, ...])	Obtain the isochrone from mapbox isochrone

transbigdata.**getbusdata**(*city*, *keywords*, *accurate=True*, *timeout=20*)

Obtain the geographic information of the bus station and bus line from the map service (Only in China)

Parameters

- **city** (*str*) – city name
- **keywords** (*list*) – Keyword, the line name
- **accurate** (*bool*) – Accurate matching
- **timeout** (*number*) – Timeout of data fetching

Returns

- **data** (*GeoDataFrame*) – The generated bus line(WGS84)

- **stop** (*GeoDataFrame*) – The generated bus station(WGS84)

transbigdata.**getadmin**(*keyword*, *ak*, *jscode=''*, *subdistricts=False*, *timeout=20*)

Input the keyword and the Amap ak. The output is the GIS file of the administrative boundary (Only in China)

Parameters

- **keywords** (*str*) – The keyword. It might be the city name such as Shengzheng, or the administrative code such as 440500
- **ak** (*str*) – Amap accesstoken
- **jscode** (*jscode*) – Amap safty code
- **subdistricts** (*bool*) – Whether to output the information of the administrative district boundary
- **timeout** (*number*) – Timeout of data fetching

Returns

- **admin** (*GeoDataFrame*) – Administrative district(WGS84)
- **districts** (*DataFrame*) – The information of subdistricts. This can be used to further get the boundary of lower level districts

transbigdata.**get_isochrone_amap**(*lon*, *lat*, *reachtime*, *ak*, *jscode=''*, *mode=2*, *timeout=20*)

Obtain the isochrone from Amap reachcricle

Parameters

- **lon** (*float*) – Longitude of the start point(WGS84)
- **lat** (*float*) – Latitude of the start point(WGS84)
- **reachtime** (*number*) – Reachtime of the isochrone
- **ak** (*str*) – Amap access token
- **jscode** (*jscode*) – Amap safty code
- **mode** (*int or str*) – Travel mode, should be 0(bus), 1(subway), 2(bus+subway)
- **timeout** (*number*) – Timeout of data fetching

Returns

isochrone – The isochrone GeoDataFrame(WGS84)

Return type

GeoDataFrame

transbigdata.**get_isochrone_mapbox**(*lon*, *lat*, *reachtime*, *access_token='auto'*, *mode='driving'*, *timeout=20*)

Obtain the isochrone from mapbox isochrone

Parameters

- **lon** (*float*) – Longitude of the start point(WGS84)
- **lat** (*float*) – Latitude of the start point(WGS84)
- **reachtime** (*number*) – Reachtime of the isochrone
- **access_token** (*str*) – Mapbox access token, if *auto* it will use the preset access token
- **mode** (*bool*) – Travel mode, should be *driving*, *walking* or *cycling*
- **timeout** (*number*) – Timeout of data fetching

Returns
isochrone – The isochrone GeoDataFrame(WGS84)

Return type
GeoDataFrame

6.4 GIS Processing

ckdnearest(dfA_origin, dfB_origin[, Aname, ...])	Search the nearest points in dfB_origin for dfA_origin, and calculate the distance
ckdnearest_point(gdA, gdB)	This method will match the nearest points in gdfB to gdfA, and add a new column called dist
ckdnearest_line(gdfA, gdfB)	This method will seach from gdfB to find the nearest line to the point in gdfA.
splitline_with_length(Centerline[, maxlength])	The intput is the linestring GeoDataFrame.
merge_polygon(data, col)	The input is the GeoDataFrame of polygon geometry, and the col name.
polyon_exterior(data[, minarea])	The input is the GeoDataFrame of the polygon geometry.

6.4.1 Nearest neighbor searches

transbigdata.**ckdnearest**(*dfA_origin*, *dfB_origin*, *Aname=['lon', 'lat']*, *Bname=['lon', 'lat']*)

Search the nearest points in dfB_origin for dfA_origin, and calculate the distance

Parameters

- **dfA_origin** (*DataFrame*) – DataFrame A
- **dfB_origin** (*DataFrame*) – DataFrame B
- **Aname** (*List*) – The column of lng and lat in DataFrame A
- **Bname** (*List*) – The column of lng and lat in DataFrame A

Returns
gdf – The output DataFrame

Return type
DataFrame

transbigdata.**ckdnearest_point**(*gdA*, *gdB*)

This method will match the nearest points in gdfB to gdfA, and add a new column called dist

Parameters

- **gdA** (*GeoDataFrame*) – GeoDataFrame A, point geometry
- **gdB** (*GeoDataFrame*) – GeoDataFrame B, point geometry

Returns
gdf – The output DataFrame

Return type
DataFrame

transbigdata.**ckdnearest_line**(*gdfA*, *gdfB*)

This method will seach from gdfB to find the nearest line to the point in gdfA.

Parameters

- **gdA** (*GeoDataFrame*) – GeoDataFrame A, point geometry
- **gdB** (*GeoDataFrame*) – GeoDataFrame B, linestring geometry

Returns

gdf – Searching the nearset linestring in gdfB for the point in gdfA

Return type

DataFrame

The following example will show how to search the nearest point-point, nearest point-edge throuh TransBigData. This method is based on KDTree algorithm. The computation complexity is o(log(n)). For more details, refer to wikihttps://en.wikipedia.org/wiki/K-d_tree

Point to point matching (DataFrame and DataFrame)

```
In [1]: import transbigdata as tbd

In [2]: import pandas as pd

In [3]: import geopandas as gpd

In [4]: from shapely.geometry import LineString

In [5]: dfA = gpd.GeoDataFrame([[1,2],[2,4],[2,6],
   ...:                         [2,10],[24,6],[21,6],
   ...:                         [22,6]],columns = ['lon1','lat1'])
   ...:

In [6]: dfA
Out[6]:
   lon1  lat1
0     1     2
1     2     4
2     2     6
3     2    10
4    24     6
5    21     6
6    22     6

In [7]: dfB = gpd.GeoDataFrame([[1,3],[2,5],[2,2]],columns = ['lon','lat'])

In [8]: dfB
Out[8]:
   lon  lat
```

(continues on next page)

(continued from previous page)

```
0    1    3
1    2    5
2    2    2
```

Use *transbigdata.ckdnearest()* to match points to points, if the inputs are two DataFrame without geometry columns, you should specify the *lon* and *lat* columns.

```
In [9]: tbd.ckdnearest(dfA,dfB,Aname=['lon1','lat1'],Bname=['lon','lat'])
Out[9]:
   lon1  lat1  index  lon  lat          dist
0     1     2      0    1    3  1.111949e+05
1     2     4      1    2    5  1.111949e+05
2     2     6      1    2    5  1.111949e+05
3     2    10      1    2    5  5.559746e+05
4    24     6      1    2    5  2.437393e+06
5    21     6      1    2    5  2.105798e+06
6    22     6      1    2    5  2.216318e+06
```

Point to point searching

Transform DataFrame to GeoDataFrame

```
In [10]: dfA['geometry'] = gpd.points_from_xy(dfA['lon1'],dfA['lat1'])

In [11]: dfA
Out[11]:
   lon1  lat1                  geometry
0     1     2   POINT (1.00000 2.00000)
1     2     4   POINT (2.00000 4.00000)
2     2     6   POINT (2.00000 6.00000)
3     2    10  POINT (2.00000 10.00000)
4    24     6  POINT (24.00000 6.00000)
5    21     6  POINT (21.00000 6.00000)
6    22     6  POINT (22.00000 6.00000)

In [12]: dfB['geometry'] = gpd.points_from_xy(dfB['lon'],dfB['lat'])

In [13]: dfB
Out[13]:
   lon  lat                 geometry
0    1    3  POINT (1.00000 3.00000)
1    2    5  POINT (2.00000 5.00000)
2    2    2  POINT (2.00000 2.00000)
```

transbigdata.ckdnearest_point()

```
In [14]: tbd.ckdnearest_point(dfA,dfB)
Out[14]:
   lon1  lat1                geometry_x  ...  lon  lat                geometry_y
0     1     2   POINT (1.00000 2.00000)  ...    1    3  POINT (1.00000 3.00000)
1     2     4   POINT (2.00000 4.00000)  ...    2    5  POINT (2.00000 5.00000)
2     2     6   POINT (2.00000 6.00000)  ...    2    5  POINT (2.00000 5.00000)
3     2    10  POINT (2.00000 10.00000)  ...    2    5  POINT (2.00000 5.00000)
4    24     6  POINT (24.00000 6.00000)  ...    2    5  POINT (2.00000 5.00000)
5    21     6  POINT (21.00000 6.00000)  ...    2    5  POINT (2.00000 5.00000)
6    22     6  POINT (22.00000 6.00000)  ...    2    5  POINT (2.00000 5.00000)

[7 rows x 8 columns]
```

Point to Line searching (GeoDataFrame and GeoDataFrame)

In this case, Table A is still a node file, Table B is a linestring file

```
In [15]: dfA['geometry'] = gpd.points_from_xy(dfA['lon1'],dfA['lat1'])

In [16]: dfB['geometry'] = [LineString([[1,1],[1.5,2.5],[3.2,4]]),
   ....:                    LineString([[1,0],[1.5,0],[4,0]]),
   ....:                    LineString([[1,-1],[1.5,-2],[4,-4]])]
   ....:

In [17]: dfB
Out[17]:
   lon  lat                                           geometry  index
0    1    3  LINESTRING (1.00000 1.00000, 1.50000 2.50000, ...      0
1    2    5  LINESTRING (1.00000 0.00000, 1.50000 0.00000, ...      1
2    2    2  LINESTRING (1.00000 -1.00000, 1.50000 -2.00000...      2

In [18]: tbd.ckdnearest_line(dfA,dfB)
Out[18]:
   lon1  lat1  ... lat                                           geometry_y
0     1     2  ...   3  LINESTRING (1.00000 1.00000, 1.50000 2.50000, ...
1     2     4  ...   3  LINESTRING (1.00000 1.00000, 1.50000 2.50000, ...
2     2     6  ...   3  LINESTRING (1.00000 1.00000, 1.50000 2.50000, ...
3     2    10  ...   3  LINESTRING (1.00000 1.00000, 1.50000 2.50000, ...
4    21     6  ...   3  LINESTRING (1.00000 1.00000, 1.50000 2.50000, ...
5    22     6  ...   3  LINESTRING (1.00000 1.00000, 1.50000 2.50000, ...
6    24     6  ...   5  LINESTRING (1.00000 0.00000, 1.50000 0.00000, ...

[7 rows x 8 columns]
```

6.4.2 Split the line

splitline_with_length can be used to split a line into several sub-line with a maximum length threshold

`transbigdata.`**`splitline_with_length`**(*Centerline*, *maxlength=100*)

The intput is the linestring GeoDataFrame. The splited line's length will be no longer than maxlength

Parameters

- **`Centerline`** (*GeoDataFrame*) – Linestring geometry
- **`maxlength`** (*number*) – The maximum length of the splited line

Returns

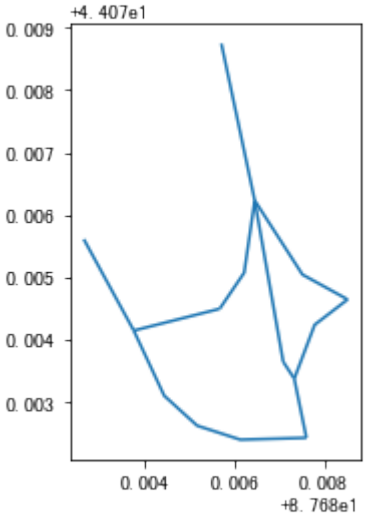
splitedline – Splited line

Return type

GeoDataFrame

The following case will show how to split a line itno 100 subline

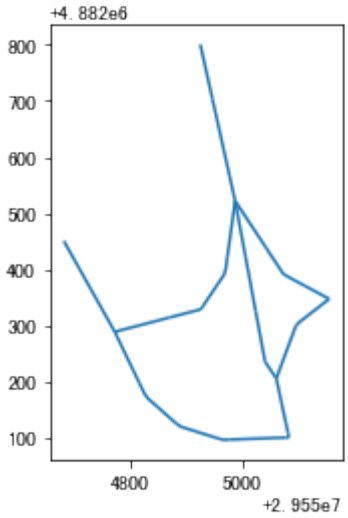
```
#
import geopandas as gpd
Centerline = gpd.read_file(r'test_lines.json')
Centerline.plot()
```




```
#
Centerline.crs = {'init':'epsg:4326'}
Centerline = Centerline.to_crs(epsg = '4517')
#
Centerline['length'] = Centerline.length
Centerline
```

```
#100
import transbigdata as tbd
splitedline = tbd.splitline_with_length(Centerline,maxlength = 100)
```

```
#
splitedline.plot()
```




```
#
splitedline
```

6.4.3 Polygon processing

transbigdata.**merge_polygon**(*data*, *col*)

The input is the GeoDataFrame of polygon geometry, and the col name. This function will merge the polygon based on the category in the mentioned column

Parameters

- **data** (*GeoDataFrame*) – The polygon geometry
- **col** (*str*) – The column name for indicating category

Returns

data1 – The merged polygon

Return type

GeoDataFrame

transbigdata.**polyon_exterior**(*data*, *minarea=0*)

The input is the GeoDataFrame of the polygon geometry. The method will construct new polygon by extending the outer boundary of the ploygon

Parameters

- **data** (*GeoDataFrame*) – The polygon geometry
- **minarea** (*number*) – The minimum area. Polygon of less area will be removed

Returns

data1 – The processed polygon

Return type

GeoDataFrame

6.5 Load the basemap

plot_map(plt, bounds[, zoom, style, printlog])	Plot the basemap
plotscale(ax, bounds[, textcolor, textsize, ...])	Add compass and scale for a map
set_mapboxtoken(mapboxtoken)	

set_imgsavepath(imgsavepath)	Set savepath for maps
read_imgsavepath()	Read map savepath
read_mapboxtoken()	Read mapboxtoken

6.5.1 Settings before start

The TransBigData package provides the function of drawing map basemap on matplotlib. The basemap is provided by mapbox and the coordinate system is WGS84. If you want to use this function, you first need to click This link to register for a mapbox account. Register as a developer on the mapbox, and obtain a mapbox token. This linkintroduces the function of mapbox token.

If you have obtained the mapbox token, you can use the following code to set the mapbox token for TransBigData (you only need to set it once, and you don't need to reset it when you reopen python later)

```
import transbigdata as tbd
#Set your mapboxtoken with the following code
tbd.set_mapboxtoken('pk.eyxxxxxxxxxx.xxxxxxxxx')
# The token you applied for must be set in it.
# Copying this line of code directly is invalid
```

In addition, you need to set the storage location of a map basemap. When the same location is displayed next time, the map will be read and loaded locally

```
# Set your map basemap storage path
# On linux or mac, the path is written like this.
# Note that there is a backslash at the end
tbd.set_imgsavepath(r'/Users/xxxx/xxxx/')

# On windows, the path is written like this.
# Finally, pay attention to two slashes to prevent escape
tbd.set_imgsavepath(r'E:\pythonscript\xxx\\')
```

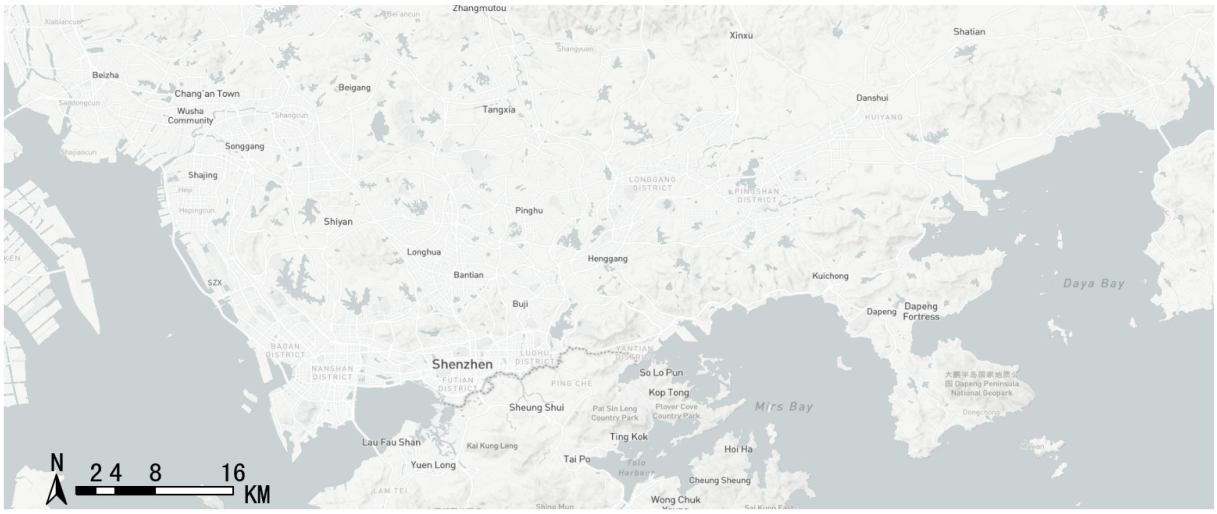
After setting, the next time you draw the base map, you will create a tileimg folder under the path you set, and put all the base maps in it. Try the following code to see if you can draw the base map successfully

```
# Define display range
bounds = [113.6,22.4,114.8,22.9]
# Plot Frame
import matplotlib.pyplot as plt
fig =plt.figure(1,(8,8),dpi=250)
ax =plt.subplot(111)
plt.sca(ax)
# Add map basemap
tbd.plot_map(plt,bounds,zoom = 11,style = 4)
```

(continues on next page)

(continued from previous page)

```
# Add scale bar and north arrow
tbd.plotscale(ax,bounds = bounds,textsize = 10,compasssize = 1,accuracy = 2000,rect = [0.
↪06,0.03],zorder = 10)
plt.axis('off')
plt.xlim(bounds[0],bounds[2])
plt.ylim(bounds[1],bounds[3])
plt.show()
```



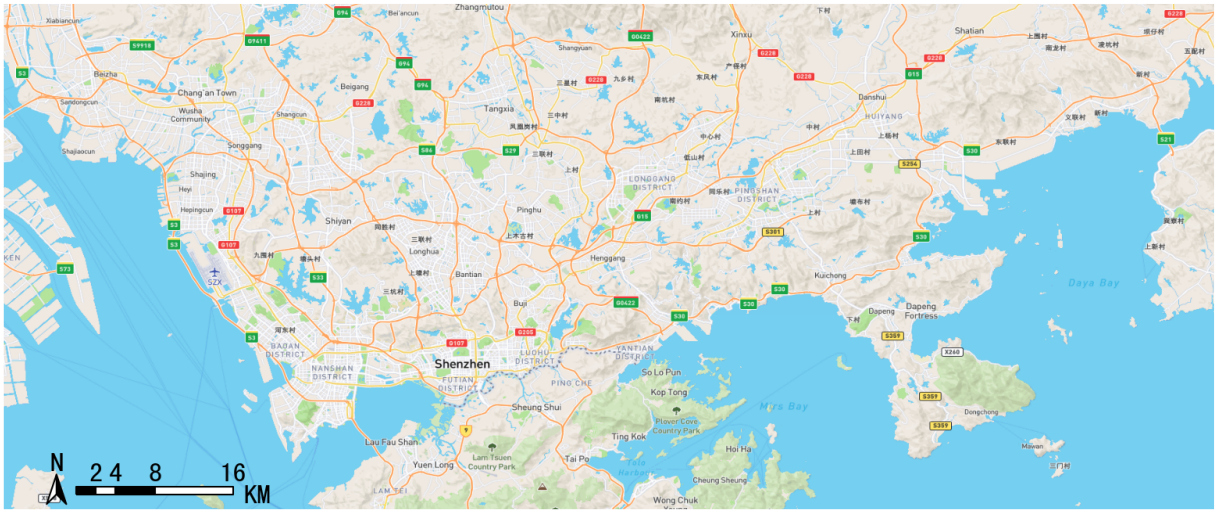
transbigdata.**plot_map**(*plt*, *bounds*, *zoom='auto'*, *style=0*, *printlog=False*)

Plot the basemap

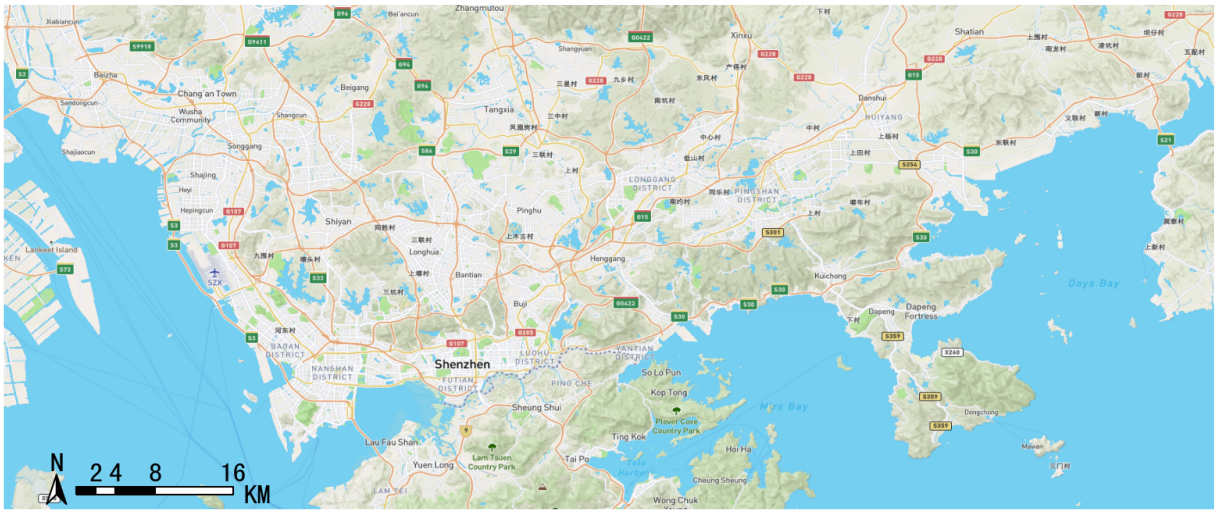
Parameters

- **plt** (*matplotlib.pyplot*) – Where to plot
- **bounds** (*List*) – The drawing boundary of the base map, [lon1,lat1,lon2,lat2] (WGS84 coordinate system), where lon1 and lat1 are the coordinates of the lower left corner and lon2 and lat2 are the coordinates of the upper right corner
- **zoom** (*number*) – The larger the magnification level of the base map, the longer the loading time. Generally, the range for a single city is between 12 and 16
- **printlog** (*bool*) – Show log
- **style** (*number*) – The style of map basemap can be 1-10, as follows

Basemap style 1streets

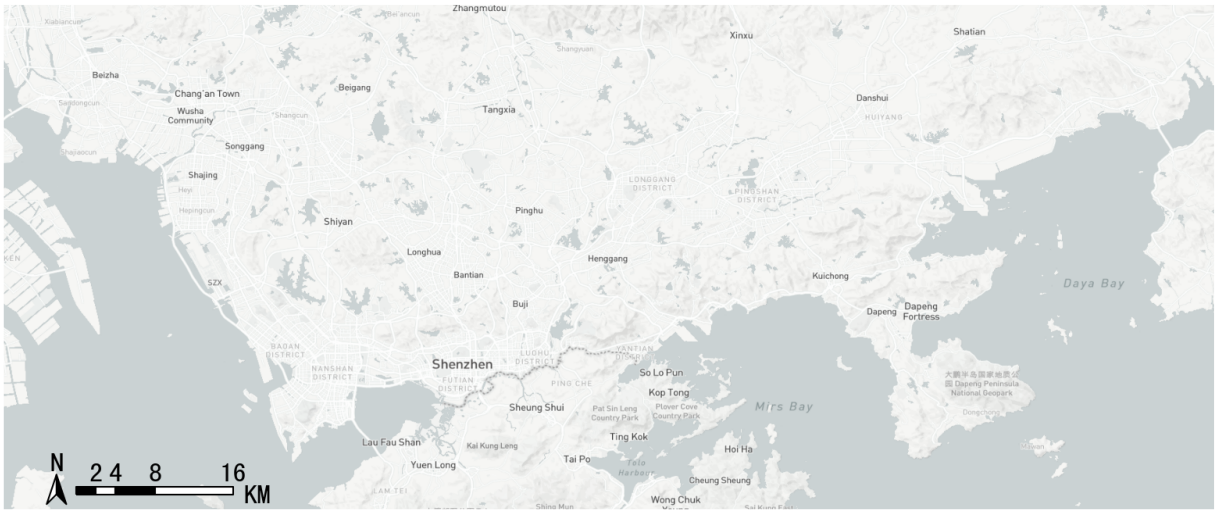


Basemap style 2outdoors



Basemap style 3satellite

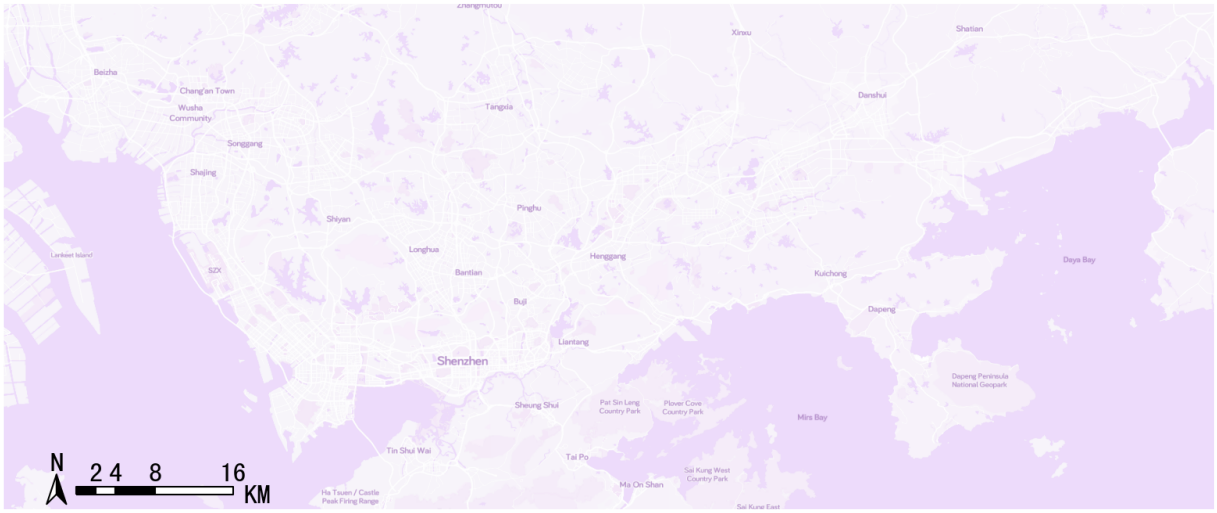
Basemap style 4light



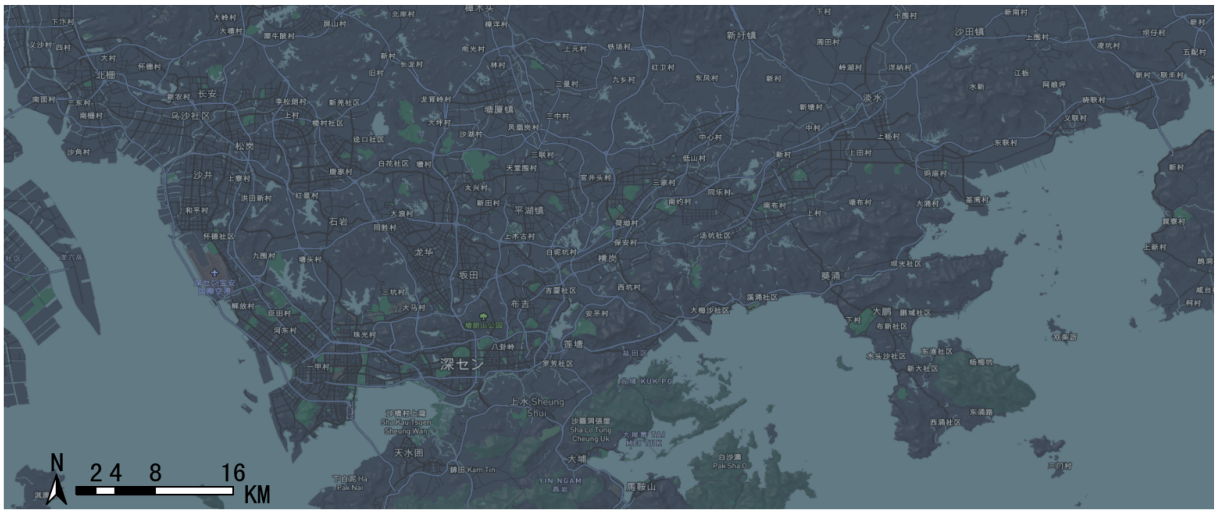
Basemap style 5dark

Basemap style 6light-ch

Basemap style 7ice creem

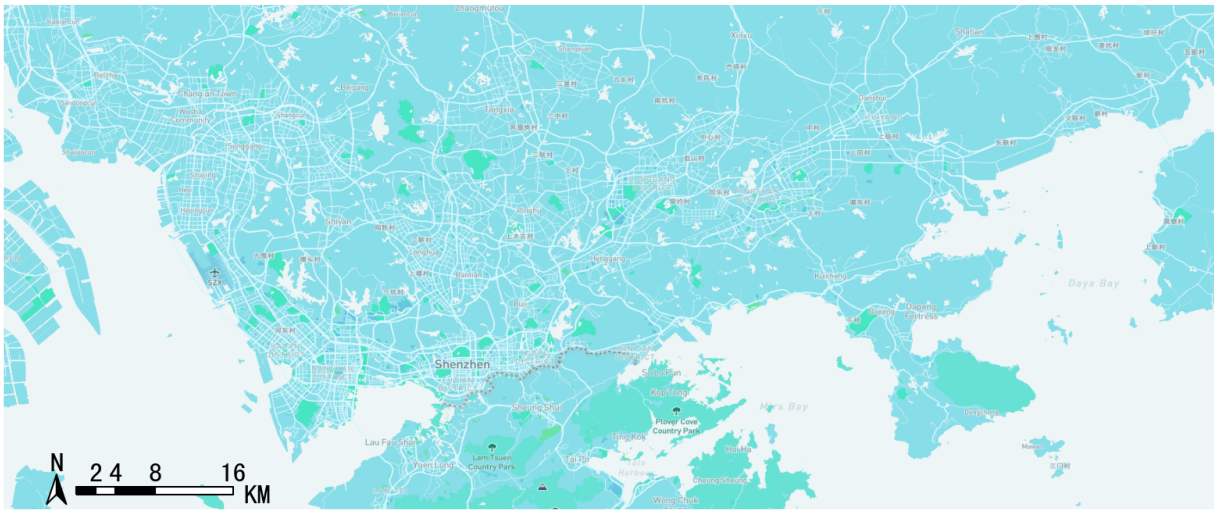


Basemap style 8night



Basemap style 9terrain

Basemap style 10basic blue



Basemap style 11light()

Basemap style 12dark()

Self-defined style

support selfdefined mapbox style

```
tbd.plot_map(plt,bounds,zoom = 11,style = 'mapbox://styles/ni1o1/
↪cl38pljx0006r14qp7ioy7gcc')
```

6.5.2 Compass and scale

transbigdata.**plotscale**(*ax*, *bounds*, *textcolor='k'*, *textsize=8*, *compasssize=1*, *accuracy='auto'*, *rect=[0.1, 0.1]*, *unit='KM'*, *style=1*, ***kwargs*)

Add compass and scale for a map

Parameters

- **bounds** (*List*) – The drawing boundary of the base map, [lon1,lat1,lon2,lat2] (WGS84 coordinate system), where lon1 and lat1 are the coordinates of the lower left corner and lon2 and lat2 are the coordinates of the upper right corner
- **textsize** (*number*) – size of the text
- **compasssize** (*number*) – size of the compass
- **accuracy** (*number*) – Length of scale bar (m)
- **unit** (*str*) – 'KM','km','M','m', the scale units
- **style** (*number*) – 1 or 2, the style of the scale
- **rect** (*List*) – The approximate position of the scale bar in the figure, such as [0.9,0.9], is in the upper right corner

```
tbd.plotscale(ax,bounds = bounds,textsize = 10,compasssize = 1,accuracy = 2000,rect = [0.
↪06,0.03])
```

6.6 Coordinates and Distances

gcj02tobd09(lng, lat)	Convert coordinates from GCJ02 to BD09
gcj02towgs84(lng, lat)	Convert coordinates from GCJ02 to WGS84
wgs84togcj02(lng, lat)	Convert coordinates from WGS84 to GCJ02
wgs84tobd09(lon, lat)	Convert coordinates from WGS84 to BD09
bd09togcj02(bd_lon, bd_lat)	Convert coordinates from BD09 to GCJ02
bd09towgs84(lon, lat)	Convert coordinates from BD09 to WGS84
bd09mctobd09(x, y)	Convert coordinates from BD09MC to BD09
transform_shape(gdf, method)	Convert coordinates of all data.
getdistance(lon1, lat1, lon2, lat2)	Input the origin/destination location in the sequence of [lon1, lat1, lon2, lat2] (in decimal) from DataFrame.

6.6.1 transbigdata.gcj02tobd09

transbigdata.**gcj02tobd09**(*lng*, *lat*)

Convert coordinates from GCJ02 to BD09

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)

- **lat** (*Series or number*) – Latitude (Converted)

6.6.2 transbigdata.gcj02towgs84

transbigdata.**gcj02towgs84**(*lng*, *lat*)

Convert coordinates from GCJ02 to WGS84

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

6.6.3 transbigdata.wgs84togcj02

transbigdata.**wgs84togcj02**(*lng*, *lat*)

Convert coordinates from WGS84 to GCJ02

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

6.6.4 transbigdata.wgs84tobd09

transbigdata.**wgs84tobd09**(*lon*, *lat*)

Convert coordinates from WGS84 to BD09

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

6.6.5 transbigdata.bd09togcj02

transbigdata.**bd09togcj02**(*bd_lon*, *bd_lat*)

Convert coordinates from BD09 to GCJ02

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

6.6.6 transbigdata.bd09towgs84

transbigdata.**bd09towgs84**(*lon*, *lat*)

Convert coordinates from BD09 to WGS84

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

6.6.7 transbigdata.bd09mctobd09

transbigdata.**bd09mctobd09**(*x*, *y*)

Convert coordinates from BD09MC to BD09

Parameters

- **x** (*Series or number*) – x coordinates
- **y** (*Series or number*) – y coordinates

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

6.6.8 transbigdata.transform_shape

transbigdata.**transform_shape**(*gdf*, *method*)

Convert coordinates of all data. The input is the geographic elements' DataFrame.

Parameters

- **gdf** (*GeoDataFrame*) – Geographic elements
- **method** (*function*) – The coordinate converting function

Returns
gdf – The result of converting

Return type
GeoDataFrame

6.6.9 transbigdata.getdistance

transbigdata.**getdistance**(*lon1*, *lat1*, *lon2*, *lat2*)

Input the origin/destination location in the sequence of [lon1, lat1, lon2, lat2] (in decimal) from DataFrame. The output is the distance (m).

Parameters

- **lon1** (*Series or number*) – Start longitude
- **lat1** (*Series or number*) – Start latitude
- **lon2** (*Series or number*) – End longitude
- **lat2** (*Series or number*) – End latitude

Returns
distance – The distance

Return type
Series or number

6.6.10 Coordinate convertering method

TransBigData package provides quick converting of coordinates such as GCJ02, BD09, BD09mc, WGS94

transbigdata.**gcj02tobd09**(*lng*, *lat*)

Convert coordinates from GCJ02 to BD09

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

transbigdata.**bd09togcj02**(*bd_lon*, *bd_lat*)

Convert coordinates from BD09 to GCJ02

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

transbigdata.**wgs84togcj02**(*lng*, *lat*)

Convert coordinates from WGS84 to GCJ02

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

transbigdata.**gcj02towgs84**(*lng*, *lat*)

Convert coordinates from GCJ02 to WGS84

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

transbigdata.**wgs84tobd09**(*lon*, *lat*)

Convert coordinates from WGS84 to BD09

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

transbigdata.**bd09towgs84**(*lon*, *lat*)

Convert coordinates from BD09 to WGS84

Parameters

- **lng** (*Series or number*) – Longitude
- **lat** (*Series or number*) – Latitude

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

transbigdata.**bd09mctobd09**(*x*, *y*)

Convert coordinates from BD09MC to BD09

Parameters

- **x** (*Series or number*) – x coordinates
- **y** (*Series or number*) – y coordinates

Returns

- **lng** (*Series or number*) – Longitude (Converted)
- **lat** (*Series or number*) – Latitude (Converted)

Coordinates reciprocal converting, based on numpy column computation:

```
>>> data['Lng'],data['Lat'] = tbd.wgs84tobd09(data['Lng'],data['Lat'])
>>> data['Lng'],data['Lat'] = tbd.wgs84togcj02(data['Lng'],data['Lat'])
>>> data['Lng'],data['Lat'] = tbd.gcj02tobd09(data['Lng'],data['Lat'])
>>> data['Lng'],data['Lat'] = tbd.gcj02towgs84(data['Lng'],data['Lat'])
>>> data['Lng'],data['Lat'] = tbd.bd09togcj02(data['Lng'],data['Lat'])
>>> data['Lng'],data['Lat'] = tbd.bd09towgs84(data['Lng'],data['Lat'])
>>> data['Lng'],data['Lat'] = tbd.bd09mctobd09(data['Lng'],data['Lat'])
```

Convert coordinates of the geographic elements

transbigdata.**transform_shape**(*gdf*, *method*)

Convert coordinates of all data. The input is the geographic elements' DataFrame.

Parameters

- **gdf** (*GeoDataFrame*) – Geographic elements
- **method** (*function*) – The coordinate converting function

Returns
gdf – The result of converting

Return type
GeoDataFrame

6.6.11 Distance measurment

transbigdata.**getdistance**(*lon1*, *lat1*, *lon2*, *lat2*)

Input the origin/destination location in the sequence of [lon1, lat1, lon2, lat2] (in decimal) from DataFrame. The output is the distance (m).

Parameters

- **lon1** (*Series or number*) – Start longitude
- **lat1** (*Series or number*) – Start latitude
- **lon2** (*Series or number*) – End longitude

- **lat2** (*Series or number*) – End latitude

Returns
distance – The distance

Return type
Series or number

6.7 Data Visualization

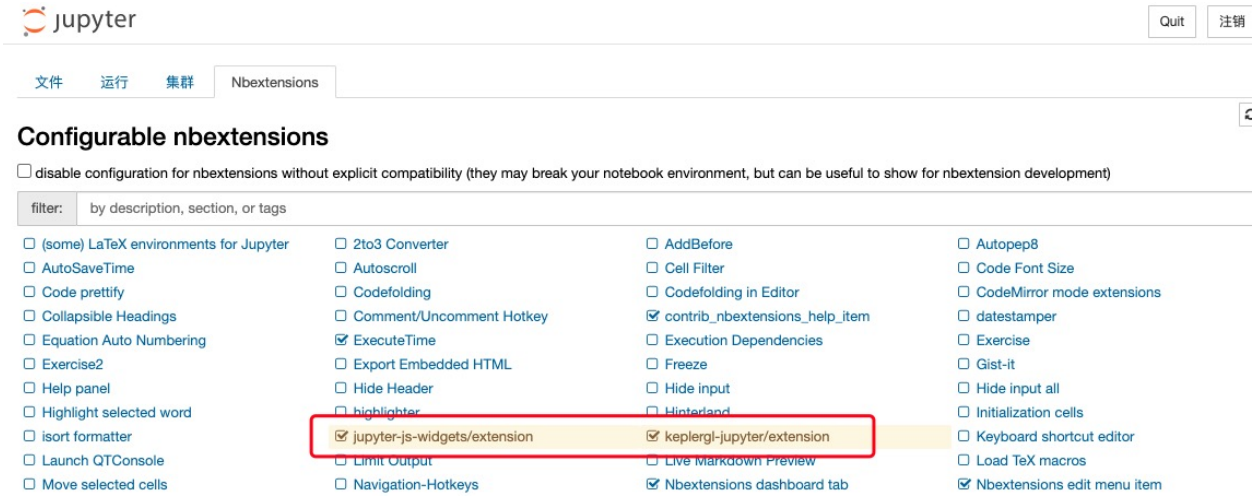
visualization_data(data[, col, accuracy, ...])	The input is the data points, this function will aggregate and then visualize it
visualization_trip(trajdata[, col, zoom, height])	The input is the trajectory data and the column name.
visualization_od(oddata[, col, zoom, ...])	The input is the OD data and the column.

6.7.1 Visualization Settings in Jupyter

The TransBigData package also provide one-click data organization and visualization methods based on the visualization plugin provided by kepler.gl to .
To use this feature, please install the keplergl package for python first.

```
pip install keplergl
```

If you want to display the visualization results in jupyter notebook, you need to check the jupyter-js-widgets (which may need to be installed separately) and keplergl-jupyter plugins



6.7.2 Visualization of data point distribution

transbigdata.**visualization_data**(*data*, *col=['lon', 'lat']*, *accuracy=500*, *height=500*, *maptype='point'*, *zoom='auto'*)

The input is the data points, this function will aggregate and then visualize it

Parameters

- **data** (*DataFrame*) – The data point
- **col** (*List*) – The column name. The user can choose a non-weight Origin-Destination (OD) data, in the sequence of [longitude, latitude]. For this, The aggregation is automatic. Or, the user can also input a weighted OD data, in the sequence of [longitude, latitude, count]
- **zoom** (*number*) – Map zoom level (Optional). Default value: auto
- **height** (*number*) – The height of the map frame
- **accuracy** (*number*) – Grid size
- **maptype** (*str*) – Map type, 'point' or 'heatmap'

Returns
vmap – Visualizations provided by keplergl

Return type
keplergl.keplergl.KeplerGl

6.7.3 Trajectory Visualization

transbigdata.**visualization_trip**(*trajdata*, *col=['Lng', 'Lat', 'ID', 'Time']*, *zoom='auto'*, *height=500*)

The input is the trajectory data and the column name. The output is the visualization result based on kepler

Parameters

- **trajdata** (*DataFrame*) – Trajectory points data
- **col** (*List*) – The column name, in the sequence of [longitude, latitude, vehicle id, time]
- **zoom** (*number*) – Map zoom level
- **height** (*number*) – The height of the map frame

Returns
vmap – Visualizations provided by keplergl

Return type
keplergl.keplergl.KeplerGl

6.7.4 OD Visualization

transbigdata.**visualization_od**(*oddata*, *col=['slon', 'slat', 'elon', 'elat']*, *zoom='auto'*, *height=500*, *accuracy=500*, *mincount=0*)

The input is the OD data and the column. The output is the visualization result based on kepler

Parameters

- **oddata** (*DataFrame*) – OD data

- **col** (*List*) – The column name. The user can choose a non-weight Origin-Destination (OD) data, in the sequence of [origin longitude, origin latitude, destination longitude, destination latitude]. For this, The aggregation is automatic. Or, the user can also input a weighted OD data, in the sequence of [origin longitude, origin latitude, destination longitude, destination latitude, count]
- **zoom** (*number*) – Map zoom level (Optional). Default value: auto
- **height** (*number*) – The height of the map frame
- **accuracy** (*number*) – Grid size
- **mincount** (*number*) – The minimum OD counts, OD with less counts will not be displayed

Returns
vmap – Visualizations provided by keplergl

Return type
keplergl.keplergl.KeplerGl

6.8 Activity

plot_activity(data[, col, figsize, dpi, ...])	Plot the activity plot of individual
entropy(sequence)	Calculate entropy.
entropy_rate(sequence)	Calculate entropy rate.
ellipse_params(data[, col, confidence, epsg])	confidence ellipse parameter estimation for point data
ellipse_plot(ellip_params, ax, **kwargs)	Enter the parameters of the confidence ellipse and plot the confidence ellipse

6.8.1 Activity plot

transbigdata.**plot_activity**(*data*, *col=['stime', 'etime', 'group']*, *figsize=(10, 5)*, *dpi=250*, *shuffle=True*, *xticks_rotation=0*, *xticks_gap=1*, *yticks_gap=1*, *fontsize=12*)

Plot the activity plot of individual

Parameters

- **data** (*DataFrame*) – activity information of one person
- **col** (*List*) – The column name. [starttime,endtime,group] of activities, *group* control the color
- **figsize** (*List*) – The figure size
- **dpi** (*Number*) – The dpi of the figure
- **shuffle** (*bool*) – Whether to shuffle the activity
- **xticks_rotation** (*Number*) – rotation angle of xticks
- **xticks_gap** (*Number*) – gap of xticks
- **yticks_gap** (*Number*) – gap of yticks
- **fontsize** (*Number*) – font size of xticks and yticks

6.8.2 Entropy

transbigdata.**entropy**(*sequence*)

Calculate entropy.

Parameters
sequence (*List,DataFrame,Series*) – sequence data

Returns
entropy

Return type
Number

transbigdata.**entropy_rate**(*sequence*)

Calculate entropy rate. Reference: Goulet-Langlois, G., Koutsopoulos, H. N., Zhao, Z., & Zhao, J. (2017). Measuring regularity of individual travel patterns. IEEE Transactions on Intelligent Transportation Systems, 19(5), 1583-1592.

Parameters
sequence (*List,DataFrame,Series*) – sequence data

Returns
entropy_rate

Return type
Number

6.8.3 Confidence ellipse

transbigdata.**ellipse_params**(*data*, *col=['lon', 'lat']*, *confidence=95*, *epsg=None*)

confidence ellipse parameter estimation for point data

Parameters

- **data** (*DataFrame*) – point data
- **confidence** (*number*) – confidence level: 9995 or 90
- **epsg** (*number*) – If given, the original coordinates are transformed from WGS84 to the given EPSG coordinate system for confidence ellipse parameter estimation
- **col** (*List*) – Column names, [lonlat]

Returns
params – Centroid ellipse parameters[pos,width,height,theta,area,oblateness] Respectively [Center point coordinates, minor axis, major axis, angle, area, oblateness]

Return type
List

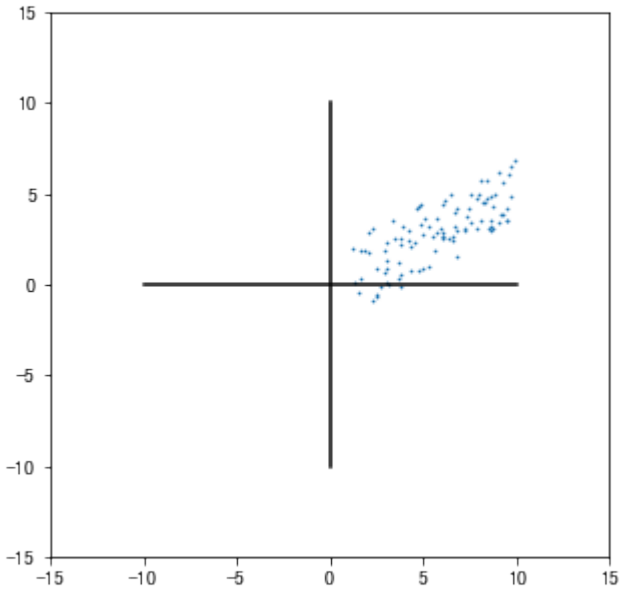
transbigdata.**ellipse_plot**(*ellip_params*, *ax*, ***kwargs*)

Enter the parameters of the confidence ellipse and plot the confidence ellipse

Parameters

- **ellip_params** (*List*) – Centroid ellipse parameters[pos,width,height,theta,area,oblateness] Respectively[Center point coordinates, minor axis, major axis, angle , area, oblateness]
- **ax** (*matplotlib.axes._subplots.AxesSubplot*) – Where to plot

```
import pandas as pd
import transbigdata as tbd
import numpy as np
#
data = np.random.uniform(1,10,(100,2))
data[:,1:] = 0.5*data[:,0:1]+np.random.uniform(-2,2,(100,1))
data = pd.DataFrame(data,columns = ['x','y'])

#
import matplotlib.pyplot as plt
plt.figure(1,(5,5))
#
plt.scatter(data['x'],data['y'],s = 0.5)
#
plt.plot([-10,10],[0,0],c = 'k')
plt.plot([0,0],[-10,10],c = 'k')
plt.xlim(-15,15)
plt.ylim(-15,15)
plt.show()
```



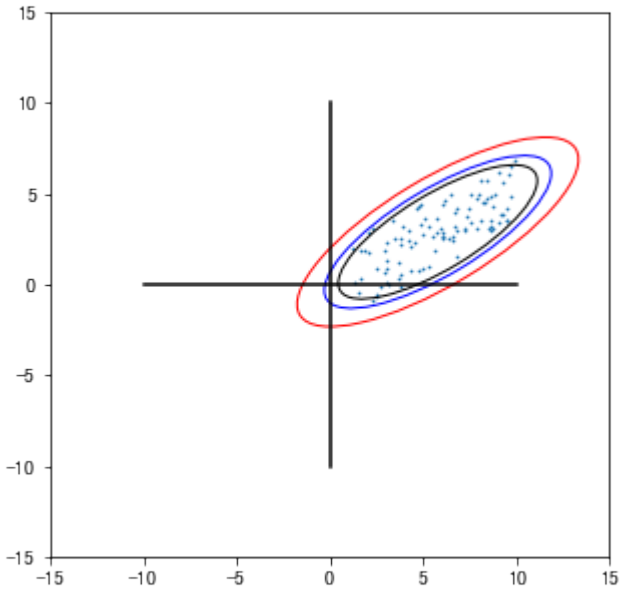
xy [

```
ellip_params = tbd.ellipse_params(data,confidence=95,col = ['x','y'])
ellip_params
```

```
[array([5.78928146, 2.88466235]),
 4.6981983145616875,
 14.04315715927693,
 -58.15524535916836,
 51.8186366184246,
 0.6654457212665993]
```

tbd.ellipse_plot

```
#
import matplotlib.pyplot as plt
plt.figure(1,(5,5))
ax = plt.subplot(111)
#
plt.scatter(data['x'],data['y'],s = 0.5)
#
#99%
ellip_params = tbd.ellipse_params(data,confidence=99,col = ['x','y'])
tbd.ellipse_plot(ellip_params,ax,fill = False,edgecolor = 'r',linewidth = 1)
#95%
ellip_params = tbd.ellipse_params(data,confidence=95,col = ['x','y'])
tbd.ellipse_plot(ellip_params,ax,fill = False,edgecolor = 'b',linewidth = 1)
#90%
ellip_params = tbd.ellipse_params(data,confidence=90,col = ['x','y'])
tbd.ellipse_plot(ellip_params,ax,fill = False,edgecolor = 'k',linewidth = 1)
#
plt.plot([-10,10],[0,0],c = 'k')
plt.plot([0,0],[-10,10],c = 'k')
plt.xlim(-15,15)
plt.ylim(-15,15)
plt.show()
```



6.9 Data Aggregating

dataagg(data, shape[, col, accuracy])	Aggregate data to traffic zone
odagg_grid(oddata, params[, col, arrow])	Aggregate the OD matrix and generate the grid geometry.
odagg_shape(oddata, shape[, col, params, ...])	Generate the OD aggregation results and the corresponding geometry.

transbigdata.**dataagg**(*data*, *shape*, *col=['Lng', 'Lat', 'count']*, *accuracy=500*)

Aggregate data to traffic zone

Parameters

- **data** (*DataFrame*) – The origin DataFrame
- **shape** (*GeoDataFrame*) – The shape of the traffic zone
- **col** (*List*) – You can either choose to input two columns, i.e., ['Lng','Lat'], or to input three columns, i.e., ['Lng','Lat','count']", where count means the points count
- **accuracy** (*number*) – The idea is to first implement data gridding and then the aggregation. Here, the grid size will be determined. The less the size is, the higher the accuracy will have.

Returns

- **aggresult** (*GeoDataFrame*) – Traffic zone. The count column is the output result
- **data1** (*DataFrame*) – The zone-matched data

transbigdata.**odagg_grid**(*oddata*, *params*, *col=['slon', 'slat', 'elon', 'elat']*, *arrow=False*, ***kwargs*)

Aggregate the OD matrix and generate the grid geometry. The input is the OD matrix (each row represents a trip). The OD will assigned to grids and then aggregated in the form of GeoDataFrame.

Parameters

- **oddata** (*DataFrame*) – OD data
- **col** (*List*) – The column of the origin/destination location,['slon','slat', 'elon','elat']. The default weight is 1 for each column. You can also add the weight parameter, for example, ['slon', 'slat','elon','elat','count'].
- **params** (*List*) – Gridding parameters (lonStart,latStart,deltaLon,deltaLat), lonStart and latStart are the lower-left coordinates, deltaLon, deltaLat are the length and width of a single grid
- **arrow** (*bool*) – Whether the generated OD geographic line contains arrows

Returns

oddata1 – GeoDataFrame of OD after aggregation

Return type

GeoDataFrame

transbigdata.**odagg_shape**(*oddata*, *shape*, *col=['slon', 'slat', 'elon', 'elat']*, *params=None*, *round_accuracy=6*, *arrow=False*, ***kwargs*)

Generate the OD aggregation results and the corresponding geometry. The input is the OD data (each row represents a trip). The OD will assigned to grids and then aggregated in the form of GeoDataFrame.

Parameters

- **oddata** (*DataFrame*) – OD data

- **shape** (*GeoDataFrame*) – GeoDataFrame of the target traffic zone
- **col** (*List*) – The column of the origin/destination location,['slon','slat','elon' ,'elat']. The default weight is 1 for each column. You can also add the weight parameter, for example, ['slon','slat','elon','elat', 'count'].
- **params** (*List (optional)*) – Gridding parameters (lonStart,latStart,deltaLon,deltaLat), lonStart and latStart are the lower-left coordinates, deltaLon, deltaLat are the length and width of a single grid If availabel, After the data gridding, the traffic zone will be matched based on the grid center. If not available, then the matching will be processed based on longitude and latitude. When the number of data items is large, the matching efficiency will be improved greatly thanks to data gridding.
- **round_accuracy** (*number*) – The number of decimal for latitude and longitude when implementing aggregation
- **arrow** (*bool*) – Whether the generated OD geographic line contains arrows

Returns
oddata1 – GeoDataFrame of OD after aggregation

Return type
GeoDataFrame

6.10 Others

dumpjson(data, path)	Input the json data and save it as a file.

transbigdata.**dumpjson**(*data*, *path*)

Input the json data and save it as a file. This method is suitable for sovling the problem that numpy cannot be compatiable with json package.

Parameters

- **data** (*json*) – The json data to be saved
- **path** (*str*) – The storage path

CHAPTER

SEVEN

METHODS FOR SPECIFIC DATA

7.1 Mobilephone data processing

mobile_stay_duration(staydata[, col, ...])	Input the stay point data to identify the duration during night and day time.
mobile_identify_home(staydata[, col, ...])	Identify home location from mobile phone stay data.
mobile_identify_work(staydata[, col, ...])	Identify work location from mobile phone stay data.

transbigdata.**mobile_stay_duration**(*staydata*, *col=['stime', 'etime']*, *start_hour=8*, *end_hour=20*)

Input the stay point data to identify the duration during night and day time.

Parameters

- **staydata** (*DataFrame*) – Stay data
- **col** (*List*) – The column name, in the order of [‘starttime’,’endtime’]
- **start_hour** (*Number*) – Start hour of day time
- **end_hour** (*Number*) – End hour of day time

Returns

- **duration_night** (*Series*) – Duration at night time
- **duration_day** (*Series*) – Duration at day time

transbigdata.**mobile_identify_home**(*staydata*, *col=['uid', 'stime', 'etime', 'LONCOL', 'LATCOL']*, *start_hour=8*, *end_hour=20*)

Identify home location from mobile phone stay data. The rule is to identify the locations with longest duration in night time.

Parameters

- **staydata** (*DataFrame*) – Stay data
- **col** (*List*) – The column name, in the order of [‘uid’,’stime’, ‘etime’, ‘locationtag1’, ‘locationtag2’, . . .]. There can be multiple ‘locationtag’ columns to specify the location.
- **start_hour** (*Number*) – Start hour and end hour of day time
- **end_hour** (*Number*) – Start hour and end hour of day time

Returns

home – Home location of mobile phone users

Return type

DataFrame

transbigdata.**mobile_identify_work**(*staydata*, *col=['uid', 'stime', 'etime', 'LONCOL', 'LATCOL']*, *minhour=3*, *start_hour=8*, *end_hour=20*, *workdaystart=0*, *workdayend=4*)

Identify work location from mobile phone stay data. The rule is to identify the locations with longest duration in day time on weekdays(Average duration should over *minhour*).

Parameters

- **staydata** (*DataFrame*) – Stay data
- **col** (*List*) – The column name, in the order of ['uid','stime','etime', 'locationtag1', 'locationtag2', ...]. There can be multiple 'locationtag' columns to specify the location.
- **minhour** (*Number*) – Minimun duration in work days (hours).
- **workdaystart** (*Number*) – Start and end of work days in the week. 0 - Monday, 4 - Friday
- **workdayend** (*Number*) – Start and end of work days in the week. 0 - Monday, 4 - Friday
- **start_hour** (*Number*) – Start hour and end hour of day time
- **end_hour** (*Number*) – Start hour and end hour of day time

Returns
work – work location of mobile phone users

Return type
DataFrame

7.2 Taxi GPS data processing

clean_taxi_status(data[, col, timelimit])	Deletes records of instantaneous changes in passenger carrying status from taxi data.
taxigps_to_od(data[, col])	Input Taxi GPS data, extract OD information
taxigps_traj_point(data, oddata[, col])	Input Taxi data and OD data to extract the trajectory points for delivery and idle trips

transbigdata.**clean_taxi_status**(*data*, *col=['VehicleNum', 'Time', 'OpenStatus']*, *timelimit=None*)

Deletes records of instantaneous changes in passenger carrying status from taxi data. These abnormal records can affect travel order judgments. Judgement method: If the passenger status of the previous record and the next record are different from this record for the same vehicle, then this record should be deleted.

Parameters

- **data** (*DataFrame*) – Data
- **col** (*List*) – Column names, in the order of ['VehicleNum', 'Time', 'OpenStatus']
- **timelimit** (*number*) – Optional, in seconds. If the time between the previous record and the next record is less than the time threshold, then it will be deleted

Returns
data1 – Cleaned data

Return type
DataFrame

transbigdata.**taxigps_to_od**(*data*, *col=['VehicleNum', 'Stime', 'Lng', 'Lat', 'OpenStatus']*)

Input Taxi GPS data, extract OD information

Parameters

- **data** (*DataFrame*) – Taxi GPS data
- **col** (*List*) – Column names in the data, need to be in order [vehicle id, time, longitude, latitude, passenger status]

Returns
oddata – OD information

Return type
DataFrame

transbigdata.**taxigps_traj_point**(*data*, *oddata*, *col=['Vehicleid', 'Time', 'Lng', 'Lat', 'OpenStatus']*)

Input Taxi data and OD data to extract the trajectory points for delivery and idle trips

Parameters

- **data** (*DataFrame*) – Taxi GPS data, field name specified by col variable
- **oddata** (*DataFrame*) – Taxi OD data
- **col** (*List*) – Column names, need to be in order [vehicle id, time, longitude, latitude, passenger status]

Returns

- **data_deliver** (*DataFrame*) – Trajectory points for delivery trips
- **data_idle** (*DataFrame*) – Trajectory points for idle trips

7.3 Bike-sharing data processing

bikedata_to_od(data[, col, startend])	Input bike-sharing order data (with data only generated when the lock is switched on and off), specify the column name, and extract the ride and parking information from it

7.3.1 transbigdata.bikedata_to_od

transbigdata.**bikedata_to_od**(*data*, *col=['BIKE_ID', 'DATA_TIME', 'LONGITUDE', 'LATITUDE', 'LOCK_STATUS']*, *startend=None*)

Input bike-sharing order data (with data only generated when the lock is switched on and off), specify the column name, and extract the ride and parking information from it

Parameters

- **data** (*DataFrame*) – Bike-sharing order data
- **col** (*List*) – Column names, the order cannot be changed.['BIKE_ID', 'DATA_TIME','LONGITUDE','LATITUDE','LOCK_STATUS']
- **startend** (*List*) – The start time and end time of the observation period, for example ['2018-08-27 00:00:00','2018-08-28 00:00:00']. If it is passed in, the riding and parking situations (from the beginning of the observation period to the first appearance of the bicycle) and (from the last appearance of the bicycle to the end of the observation period) are considered.

Returns

- **move_data** (*DataFrame*) – Riding data
- **stop_data** (*DataFrame*) – Parking data

transbigdata.**bikedata_to_od**(*data*, *col=['BIKE_ID', 'DATA_TIME', 'LONGITUDE', 'LATITUDE', 'LOCK_STATUS']*, *startend=None*)

Input bike-sharing order data (with data only generated when the lock is switched on and off), specify the column name, and extract the ride and parking information from it

Parameters

- **data** (*DataFrame*) – Bike-sharing order data
- **col** (*List*) – Column names, the order cannot be changed.['BIKE_ID', 'DATA_TIME','LONGITUDE','LATITUDE','LOCK_STATUS']
- **startend** (*List*) – The start time and end time of the observation period, for example ['2018-08-27 00:00:00','2018-08-28 00:00:00']. If it is passed in, the riding and parking situations (from the beginning of the observation period to the first appearance of the bicycle) and (from the last appearance of the bicycle to the end of the observation period) are considered.

Returns

- **move_data** (*DataFrame*) – Riding data
- **stop_data** (*DataFrame*) – Parking data

7.4 Bus GPS data processing

busgps_arriveinfo(data, line, stop[, col, ...])	Input bus GPS data, bus route and station GeoDataFrame, this method can identify the bus arrival and departure information
busgps_onewaytime(arrive_info, start, end[, col])	Input the departure information table drive_info and the station information table stop to calculate the one-way travel time

7.4.1 transbigdata.busgps_arriveinfo

transbigdata.**busgps_arriveinfo**(*data*, *line*, *stop*, *col=['VehicleId', 'GPSDateTime', 'lon', 'lat', 'stopname']*, *stopbuffer=200*, *mintime=300*, *disgap=200*, *project_epsg='auto'*, *timegap=1800*, *projectoutput=False*)

Input bus GPS data, bus route and station GeoDataFrame, this method can identify the bus arrival and departure information

Parameters

- **data** (*DataFrame*) – Bus GPS data. It should be the data from one bus route, and need to contain vehicle ID, GPS time, latitude and longitude (wgs84)
- **line** (*GeoDataFrame*) – GeoDataFrame for the bus line
- **stop** (*GeoDataFrame*) – GeoDataFrame for bus stops
- **col** (*List*) – Column names, in the order of [vehicle ID, time, longitude, latitude, station name]

- **stopbuffer** (*number*) – Meter. When the vehicle approaches the station within this certain distance, it is considered to be arrive at the station.
- **mintime** (*number*) – Seconds. Within a short period of time that the bus arrive at bus station again, it will not be consider as another arrival
- **disgap** (*number*) – Meter. The distance between the front point and the back point of the vehicle, which is used to determine whether the vehicle is moving or not
- **project_epsg** (*number*) – The matching algorithm will convert the data into a projection coordinate system to calculate the distance, here the epsg code of the projection coordinate system is given
- **timegap** (*number*) – Seconds. For how long the vehicle does not appear, it will be considered as a new vehicle
- **projectoutput** (*bool*) – Whether to output the projected data

Returns
arrive_info – Bus arrival and departure information

Return type
DataFrame

7.4.2 transbigdata.busgps_onewaytime

transbigdata.**busgps_onewaytime**(*arrive_info*, *start*, *end*, *col=['VehicleId', 'stopname', 'arrivetime', 'leavetime']*)

Input the departure information table drive_info and the station information table stop to calculate the one-way travel time

Parameters

- **arrive_info** (*DataFrame*) – The departure information table drive_info
- **start** (*Str*) – Starting station name
- **end** (*Str*) – Ending station name
- **col** (*List*) – Column name [vehicle ID, station name,arrivetime,leavetime]

Returns
onewaytime – One-way travel time of the bus

Return type
DataFrame

transbigdata.**busgps_arriveinfo**(*data*, *line*, *stop*, *col=['VehicleId', 'GPSDateTime', 'lon', 'lat', 'stopname']*, *stopbuffer=200*, *mintime=300*, *disgap=200*, *project_epsg='auto'*, *timegap=1800*, *projectoutput=False*)

Input bus GPS data, bus route and station GeoDataFrame, this method can identify the bus arrival and departure information

Parameters

- **data** (*DataFrame*) – Bus GPS data. It should be the data from one bus route, and need to contain vehicle ID, GPS time, latitude and longitude (wgs84)
- **line** (*GeoDataFrame*) – GeoDataFrame for the bus line
- **stop** (*GeoDataFrame*) – GeoDataFrame for bus stops

- **col** (*List*) – Column names, in the order of [vehicle ID, time, longitude, latitude, station name]
- **stopbuffer** (*number*) – Meter. When the vehicle approaches the station within this certain distance, it is considered to be arrive at the station.
- **mintime** (*number*) – Seconds. Within a short period of time that the bus arrive at bus station again, it will not be consider as another arrival
- **disgap** (*number*) – Meter. The distance between the front point and the back point of the vehicle, which is used to determine whether the vehicle is moving or not
- **project_epsg** (*number*) – The matching algorithm will convert the data into a projection coordinate system to calculate the distance, here the epsg code of the projection coordinate system is given
- **timegap** (*number*) – Seconds. For how long the vehicle does not appear, it will be considered as a new vehicle
- **projectoutput** (*bool*) – Whether to output the projected data

Returns
arrive_info – Bus arrival and departure information

Return type
DataFrame

transbigdata.**busgps_onewaytime**(*arrive_info*, *start*, *end*, *col=['VehicleId', 'stopname', 'arrivetime', 'leavetime']*)

Input the departure information table drive_info and the station information table stop to calculate the one-way travel time

Parameters

- **arrive_info** (*DataFrame*) – The departure information table drive_info
- **start** (*Str*) – Starting station name
- **end** (*Str*) – Ending station name
- **col** (*List*) – Column name [vehicle ID, station name,arrivetime,leavetime]

Returns
onewaytime – One-way travel time of the bus

Return type
DataFrame

7.5 Bus and subway network topology modeling

7.5.1

metro_network(line, stop[, transfertime, ...])	Inputting the metro station data and outputting the network topology model.
get_shortest_path(G, stop, ostation, dstation)	Obtain the travel path of shortest path from the metro nextwork
get_k_shortest_paths(G, stop, ostation, ...)	Obtain the k th shortest paths from the metro nextwork
get_path_traveltime(G, path)	Obtain the travel time of the path
split_subwayline(line, stop)	To slice the metro line with metro stations to obtain metro section information (This step is useful in subway passenger flow visualization)

transbigdata.**metro_network**(*line*, *stop*, *transfertime=5*, *nxgraph=True*)

Inputting the metro station data and outputting the network topology model. The graph generated relies on NetworkX. The travel time is consist of: operation time between stations + stop time at each station + transfer time

Parameters

- **line** (*GeoDataFrame*) – Lines. Should have *line* column to store line name *speed* column to store metro speed and *stoptime* column to store stop time at each station.
- **stop** (*GeoDataFrame*) – Bus/metro stations
- **transfertime** (*number*) – Travel time per transfer
- **nxgraph** (*bool*) – Default True, if True then output the network G constructed by NetworkX, if False then output the edges1(line section), edge2(station transfer), and the node of the network

Returns

- *When the nxgraph parameter is True*
- ==========
- **G** (*networkx.classes.graph.Graph*) – Network G built by networkx.
- *When the nxgraph parameter is False (This is for detail design of the network)*
- ==========
- **edge1** (*DataFrame*) – Network edge for line section.
- **edge2** (*DataFrame*) – Network edge for transfering.
- **node** (*List*) – Network nodes.

transbigdata.**get_shortest_path**(*G*, *stop*, *ostation*, *dstation*)

Obtain the travel path of shortest path from the metro nextwork

Parameters

- **G** (*networkx.classes.graph.Graph*) – metro network
- **stop** (*DataFrame*) – metro stop dataframe
- **ostation** (*str*) – O station name

- **dstation** (*str*) – D station name

Returns
path – travel path: list of station names

Return type
list

transbigdata.**get_k_shortest_paths**(*G*, *stop*, *ostation*, *dstation*, *k*)

Obtain the k th shortest paths from the metro nextwork

Parameters

- **G** (*networkx.classes.graph.Graph*) – metro network
- **stop** (*DataFrame*) – metro stop dataframe
- **ostation** (*str*) – O station name
- **dstation** (*str*) – D station name
- **k** (*int*) – the k th shortest paths

Returns
paths – travel path: list of travel paths

Return type
list

transbigdata.**get_path_traveltime**(*G*, *path*)

Obtain the travel time of the path

Parameters

- **G** (*networkx.classes.graph.Graph*) – metro network
- **path** (*list*) – list of stationnames

Returns
traveltime – travel time of the path

Return type
float

transbigdata.**split_subwayline**(*line*, *stop*)

To slice the metro line with metro stations to obtain metro section information (This step is useful in subway passenger flow visualization)

Parameters

- **line** (*GeoDataFrame*) – Bus/metro lines
- **stop** (*GeoDataFrame*) – Bus/metro stations

Returns
metro_line_splited – Generated section line shape

Return type
GeoDataFrame

INDEX

T

V

W