mobilkit

Enrico Ubaldi enrico.ubaldi@mindearth.org - MindEarth

Jan 20, 2024

CONTENTS:

mobilkit is a library for analyzing human mobility data in Python, leveraging on the *Dask* framework for faster, parallel computation.

The library is in continuos development and currently allows to:

- Load and filter raw mobility data covering large spatial and temporal extensions.
- Compute the user statistics (number of active days, number of positions recorded) and filter them accordingly to the analysis.
- extract home and work locations of users based on a given tessellation.
- compute the land use of a given urban region
- characterize the displacement of people under different grouping (distance from an epicenter, socio-economic index, etc.) after a major event

CHAPTER

ONE

DOCUMENTATION

Besides this documentation, many example notebooks can be found in the original repo under the docs/examples folder. Detailed notebooks with all the functionalities shown are found in examples/.

CHAPTER

TWO

COLLABORATE WITH US

mobilkit is an active project and any contribution is welcome.

You are encouraged to report any issue or problem encountered while using the software or to seek for support.

If you would like to contribute or add functionalities to *mobilkit*, feel free to fork the project, open an issue and contact us.

CHAPTER

THREE

INSTALLATION

Note: *mobilkit* will install a complete installation of Dask, so consider installing it in a virtualenv to connect to an existing dask cluster.

Note: You can try *mobilkit* without installing it on Binder, just click below

1. Create an environment *mobilkit*

```
python3 -m venv mobilkit
```

2. Activate

```
source mobilkit/bin/activate
```

3. Update pip to latest version in the environment

```
pip install --upgrade pip
```

4. Install mobilkit

```
pip install mobilkit
```

5. OPTIONAL to use *mobilkit* on the jupyter notebook
   - Activate the virutalenv:

```
source mobilkit/bin/activate
```

   - Install jupyter notebook:

```
pip install jupyter
```

   - Run jupyter notebook

```
jupyter notebook
```

   - (Optional) install the kernel with a specific name

```
ipython kernel install --user --name=mobilkit_env
```

If you already have scikit-mobility installed, skip the environment creation and run these commands from the *skmob* anaconda environment.

mobilkit by default will only install core packages needed to run the main functions. There are three optional packages of dipendencies (the *mobilkit[complete]* installs everything):

- *[viz]* will install *contextily*, needed to visualize map backgrounds in certain viz functions;
- *[doc]* will install all the needed packages to build the docs;
- *[skmob]* will install *scikit-mobility* as well.

CHAPTER

FOUR

TEST THE INSTALLATION

```
> source activate mobilkit
(mobilkit) > python
>>> import mobilkit
```

CHAPTER

FIVE

CITING

If you use *mobilkit* please cite us:

Note: Enrico Ubaldi, Takahiro Yabe, Nicholas K. W. Jones, Maham Faisal Khan, Satish V. Ukkusuri, Riccardo Di Clemente and Emanuele Strano Mobilkit: A Python Toolkit for Urban Resilience and Disaster Risk Management Analytics using High Frequency Human Mobility Data, 2021, KDD 2021 Humanitarian Mapping Workshop, https://arxiv.org/abs/2107.14297

Bibtex:

@misc{ubaldi2021mobilkit,
title={Mobilkit: A Python Toolkit for Urban Resilience and Disaster Risk Management Analytics using High Frequency Human Mobility Data}, author={Enrico Ubaldi and Takahiro Yabe and Nicholas K. W. Jones and Maham Faisal Khan and Satish V. Ukkusuri and Riccardo Di Clemente and Emanuele Strano}, year={2021}, eprint={2107.14297}, primaryClass={cs.CY}, archivePrefix={arXiv}}

CHAPTER

SIX

CREDITS AND CONTACTS

This code has been developed by Mindearth, the Global Facility for Disaster Reduction and Recovery (GFDRR) and Purdue University.

Funding was provided by the Spanish Fund for Latin America and the Caribbean (SFLAC) under the Disruptive Technologies for Development (DT4D) program.

The findings, interpretations, and conclusions expressed in this repository and in the example notebooks are entirely those of the authors. They do not necessarily represent the views of the International Bank for Reconstruction and Development/World Bank and its affiliated organizations, or those of the Executive Directors of the World Bank or the governments they represent.

6.1 Quickstart

This is a notebook explaining the usage of the main features of `mobilkit`.

The features covered here are:

- how to create a synthetic dataset from the Microsoft's GeoLife dataset.
- how to create a tessellation shapefile in case you only have a collection of centroid;
- load data from a `pandas` dataframe;
- tessellate the pings (assign them to a given location);
- compute the land use of an urban area;
- compute the resident population for each area and compare it with census figures;
- compute user activity statistics and filter users accordingly;
- compute the displacement figures in a given area.

To allow the publication of the data we used an open dataset such as the GeoLife one. We augment the number of users observed by using some functions present in the `mobilkit.loader` module.

Depending on the case we map each user/day or each user/week to a synthetic user performing the same events as in the original dataset at the same original time, just traslating everything in the synthetic day or week..

To continue you have to download the GeoLife dataset and uncompress it in the `data` directory.

[1]:
```
%config Completer.use_jedi = False
%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
import pytz
from datetime import datetime
import geopandas as gpd

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

import mobilkit
from dask.distributed import Client
from shapely.geometry import Polygon, Point
from dask import dataframe as dd

import warnings
warnings.filterwarnings('ignore')

import mobilkit
```

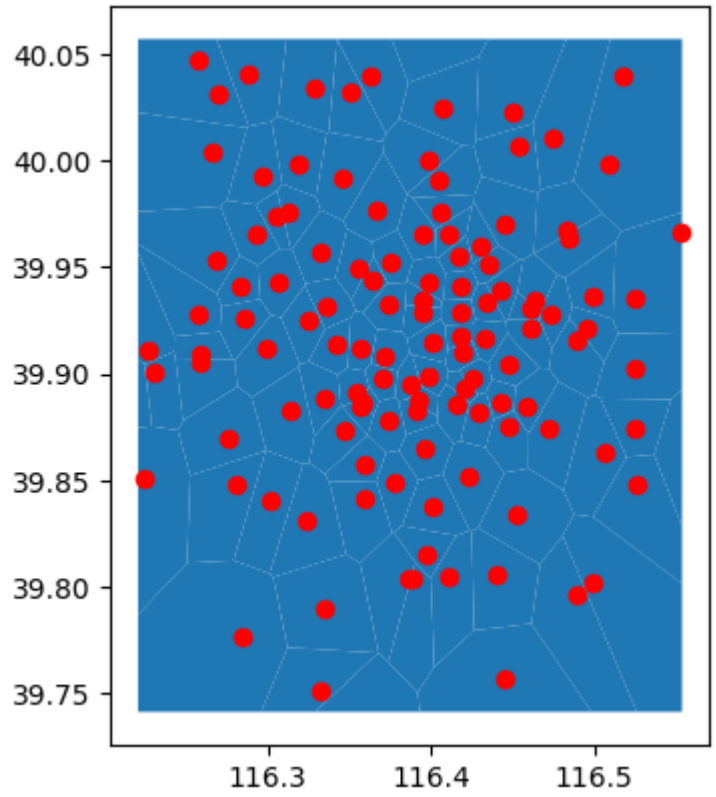
6.1.1 Create tessellation from points

I use the file with centroids found here, select some points in Beijing and create a Voronoi tessellation of it.

[2]:
```
# Choose the spatial extent of your analysis and where to save the Vortonoi tessellation
box = (116.20, 39.74, 116.56, 40.06)
voronoi_file = "../../data/Beijing/voronoi_points_beijing.shp"

df_china = gpd.read_file("../../data/Beijing/DT19new/PopCensus2010_township.shp")
if not os.path.exists(voronoi_file):
    poly_box = mobilkit.spatial.box2poly(box)
    df_china = df_china[df_china.within(poly_box)]
    poly_gdf = gpd.GeoDataFrame(["Region"], geometry=[poly_box], crs=df_china.crs)
    layer = mobilkit.spatial.makeVoronoi(df_china)
    layer.to_file(voronoi_file)
else:
    layer = gpd.read_file(voronoi_file)
    box = layer.unary_union.bounds
    poly_box = mobilkit.spatial.box2poly(box)
    df_china = df_china[df_china.within(poly_box)]
```

[3]:
```
ax = layer.plot()
df_china.plot(color="r", ax=ax)
```

[3]:
```
<Axes: >
```



6.1.2 Load/reload the geolife trajectories data

You should manually download the data from here and put them into to `data/` folder and unzip them there.

```
cd ../data/
wget https://download.microsoft.com/download/F/4/8/F4894AA5-FDBC-481E-9285-D5F8C4C4F039/
↪Geolife%20Trajectories%201.3.zip
unzip Geolife%20Trajectories%201.3.zip
```

[4]:
```
# !wget -P ../../data/ https://download.microsoft.com/download/F/4/8/F4894AA5-FDBC-481E-
↪9285-D5F8C4C4F039/Geolife%20Trajectories%201.3.zip
# !unzip -d ../../data/ ../../data/Geolife\ Trajectories\ 1.3.zip
```

[5]:
```
geolifePath = "../../data/Geolife Trajectories 1.3"
pkl_trajectories = "../../data/Geolife Trajectories 1.3/processed_traj.pkl"
if not os.path.exists(pkl_trajectories):
    df_geolife = mobilkit.loader.loadGeolifeData(geolifePath)
    df_geolife.to_pickle(pkl_trajectories)
else:
    df_geolife = pd.read_pickle(pkl_trajectories)

df_geolife.head()
```

[5]:

	UTC	acc	datetime	lat	lng	uid
0	1224730384	1	2008-10-23 10:53:04+08:00	39.984702	116.318417	0

(continues on next page)

(continued from previous page)

```
1  1224730390    1 2008-10-23 10:53:10+08:00  39.984683  116.318450    0
2  1224730395    1 2008-10-23 10:53:15+08:00  39.984686  116.318417    0
3  1224730400    1 2008-10-23 10:53:20+08:00  39.984688  116.318385    0
4  1224730405    1 2008-10-23 10:53:25+08:00  39.984655  116.318263    0
```

6.1.3 Create synthetic days/week from the data

We perform two expansion of the data: - a weekly one, where each (user,week) couple is treated as a separate user and all the events are moved to a synthetic week keeping the original weekday, hour, minute and second of the recorded point; - a daily one, where each (user,day) couple is treated as a separate user and all the events are moved to a synthetic day keeping the original hour, minute and second of the recorded point;

[6]:
```
# One day with all the data projected to a single day
selected_day = datetime(2020, 6, 1, tzinfo=pytz.timezone("UTC"))
df_users_day = mobilkit.loader.syntheticGeoLifeDay(df_geolife, selected_day=selected_day)
df_users_day["uid"].nunique()
```

[6]:
```
11152
```

[7]:
```
# One week with all the data projected to a single week
selected_week = datetime(2020, 6, 4, tzinfo=pytz.timezone("UTC"))
df_users_week = mobilkit.loader.syntheticGeoLifeWeek(df_geolife, selected_week=selected_
↪week)
df_users_week["uid"].nunique()
```

```
Anticipated the date to Monday:  2020-06-01 00:00:00+00:00
```

[7]:
```
2524
```

We now have two dataframes containing our data.

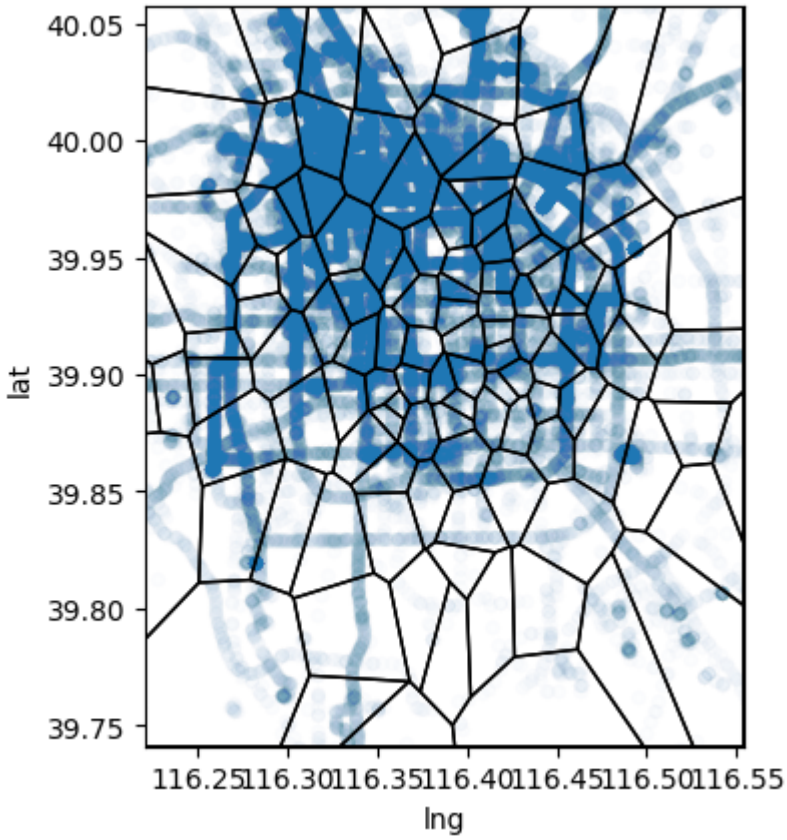
Each row is an event containing the spatial and temporal information.

[8]:
```
df_users_week.head(4)
```

[8]:
```
          UTC  acc                  datetime        lat         lng  uid
0  1591267984    1 2020-06-04 10:53:04+00:00  39.984702  116.318417    0
1  1591267990    1 2020-06-04 10:53:10+00:00  39.984683  116.318450    0
2  1591267995    1 2020-06-04 10:53:15+00:00  39.984686  116.318417    0
3  1591268000    1 2020-06-04 10:53:20+00:00  39.984688  116.318385    0
```

[9]:
```
# Get a small fraction of the pings and see how they are distributed over the␣
↪tessellation
ax = df_users_day.sample(frac=.01).plot("lng","lat",kind="scatter", alpha=.01)
ax = layer.plot(color="none", edgecolor="k", ax=ax)

box = layer.unary_union.bounds
plt.xlim(box[0], box[2])
plt.ylim(box[1], box[3])
```

[9]:
```
(39.74096801500008, 40.05733690900007)
```



6.1.4 Create dask client

Launch worker and scheduler if working on localhost with:

```
dask-worker 127.0.0.1:8786 --nworkers -1 & dask-scheduler
```

If you get an error with `Popen` in `dask-worker`, add the option `--preload-nanny multiprocessing.popen_spawn_posix` to the first command.

[10]:
```
client = Client(address="127.0.0.1:8786")
client
```

[10]:
```
<Client: 'tcp://192.168.1.20:8786' processes=48 threads=48, memory=188.55 GiB>
```

6.1.5 Load events in dask

Here we use the `dask` API, see the `loader` module on how to load pings from raw files.

[11]:
```
dd_users_raw = dd.from_pandas(df_users_day, npartitions=20)
dd_week_raw = dd.from_pandas(df_users_week, npartitions=20)
```

6.1.6 Compute user stats

We first focus on the weekly dataframe because it has more than one day (but less users).

We show here how to compute the basic user stats.

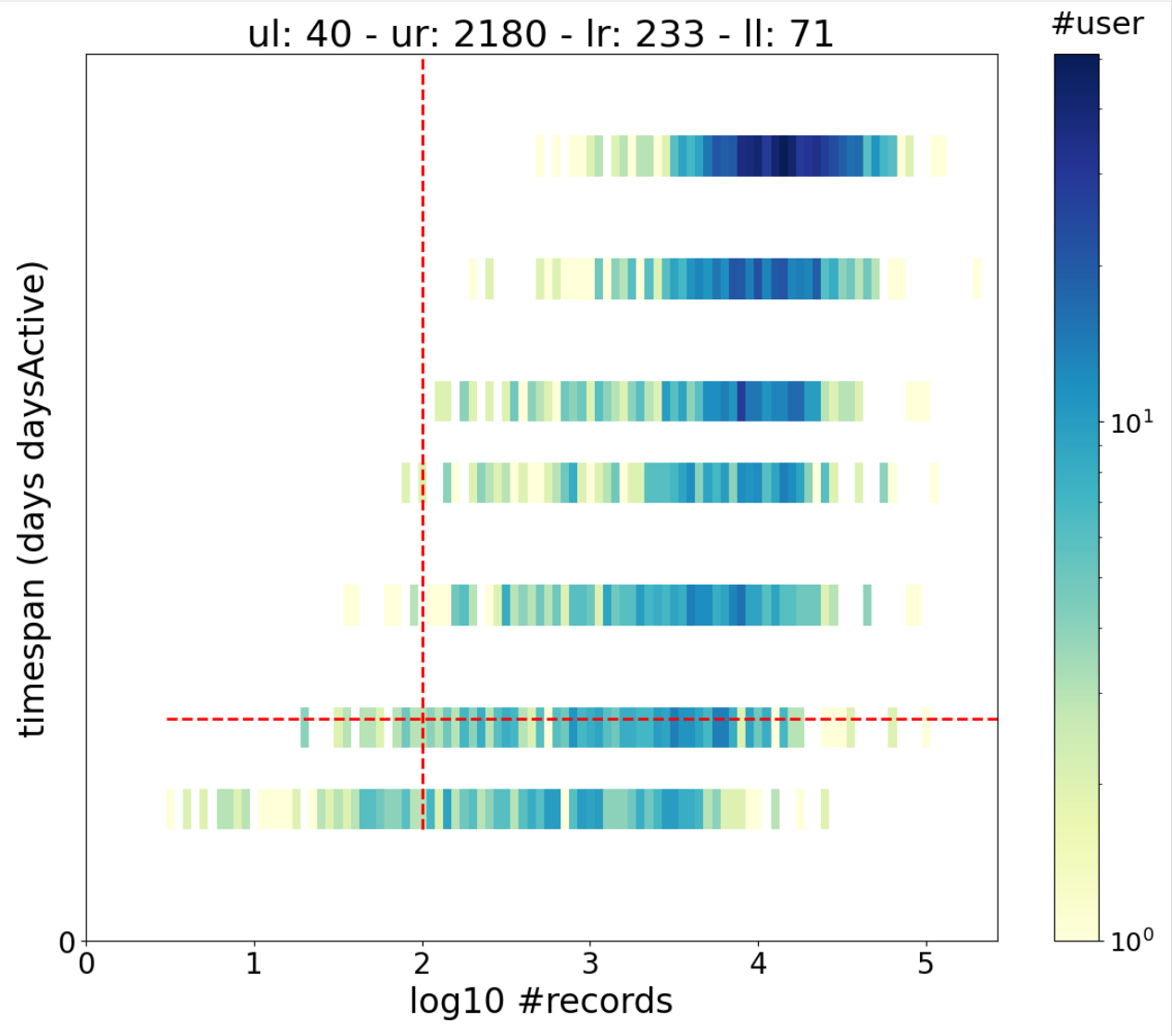
You can see the `mobilkit.stats.userStats` documentation for details but we get back some basic stats for each user.

[12]:
```
# Use the .compute() to make it a pandas df
users_stats_df = mobilkit.stats.userStats(dd_week_raw).compute()
users_stats_df.head(4)
```

[12]:
```
   uid                  min_day                  max_day  pings  daysActive  \
0   82 2020-06-01 00:00:00+00:00 2020-06-07 00:00:00+00:00  14027           7
1  102 2020-06-01 00:00:00+00:00 2020-06-07 00:00:00+00:00  16598           7
2  111 2020-06-01 00:00:00+00:00 2020-06-07 00:00:00+00:00  11562           7
3  141 2020-06-03 00:00:00+00:00 2020-06-06 00:00:00+00:00   3909           3

   daysSpanned                                pingsPerDay          avg
0            6     [779, 1814, 790, 1848, 852, 2096, 5848]  2003.857143
1            6   [641, 2493, 3853, 1181, 2695, 2251, 3484]  2371.142857
2            6  [1165, 1124, 1038, 1001, 2203, 1453, 3578]  1651.714286
3            3                           [218, 932, 2759]  1303.000000
```

[13]:
```
# We can plot the distribution of the number of pings and active days to see how many␣
↪users we have in each quadrant
mobilkit.stats.plotUsersHist(users_stats_df, min_pings=100, min_days=2)
```

[13]:
```
<Axes: title={'center': 'ul: 40 - ur: 2180 - lr: 233 - ll: 71'}, xlabel='log10 #records',
↪ ylabel='timespan (days daysActive)'>
```



6.1.7 Filter users based on stats

We want to keep only users with at least 2 active days and 100 pings.

[14]:
```
# We either provide users filtering by hand
valid_users = list(users_stats_df.query("daysActive >= 2 & pings > 100")["uid"])
df_filtered = mobilkit.stats.filterUsersFromSet(dd_week_raw, users_set=valid_users)
```

[15]:
```
# Or we could have had the stats and filter computed at once
df_filtered, users_stats_df, valid_users = mobilkit.stats.filterUsers(dd_week_raw,␣
↪minPings=100, minDaysActive=2)
```

6.1.8 Assign pings to an area

We first assign each ping to a given area passing the name of the shapefile to use.

With `filterAreas=True` we are discarding all the events that fall outside of our ROI.

[16]:
```
dd_week_with_zones, tessellation_gdf = mobilkit.spatial.tessellate(df_filtered,
                                                                   tesselation_
↪shp=voronoi_file,
                                                                   filterAreas=True)
```

6.1.9 Compute home and work areas

Once we have each ping assigned to a given area, we can sort out the home and work areas of each user by looking where he spends most of the time during day and night-time.

We first add the `isHome` and `isWork` columns, then we pass this df to the home location function to see where an agent lives.

[17]:
```
# Add the home/work columns, all the events within the given hours will be considered␣
↪home/work
dd_week_hw = mobilkit.stats.userHomeWork(dd_week_with_zones,
                                         homeHours=(20, 8),
                                         workHours=(9,18))
```

[18]:
```
# Compute the locations and pass them to pandas:
# - the tile_IDs of the areas of home and work;
# - the lat and lon of the home and work locations;
df_hw_locs = mobilkit.stats.userHomeWorkLocation(dd_week_hw)
df_hw_locs_pd = df_hw_locs.compute()
```

The synthetic case

We now merge our population estimate with the one given in the original shapefile in the `POP` column.

Results are not beautiful but remember that: - we are working on a small dataset (~200 original users) expanded to simulate many users in an arbitrary way; - the spatial tessellation may be different from the original one as we reconstructed it with a Voronoi tessellation;

While we focus on the Beijing synthetic case here, in the next section we will show the estimations for the Mexico usecase, where results are found to be in very good agreement with the empirical case.

[19]:
```
df_hw_locs_pd.head(4)
```

[19]:
```
     tot_pings  home_tile_ID   lat_home    lng_home  home_pings  work_tile_ID  \
uid
82      4297.0         106.0  40.004416  116.322909      1218.0         106.0
102    16531.0         106.0  39.996643  116.325115      2471.0         106.0
111    11562.0         106.0  40.000315  116.322960      2240.0         106.0
141     2200.0         102.0  39.981531  116.338790        78.0          89.0

      lat_work    lng_work  work_pings
uid
```

(continues on next page)

(continued from previous page)

```
82   39.997723  116.323012      1040.0
102  40.000683  116.323300      1966.0
111  40.001965  116.322623      3066.0
141  39.967705  116.339616       522.0
```

[20]:
```
population_per_area = df_hw_locs_pd.reset_index().groupby("home_tile_ID").agg({
                                                "uid": "nunique",
                                                "home_pings": "sum"}).reset_index()

population_per_area = population_per_area.rename(columns={
                                                "home_tile_ID": "tile_ID",
                                                "uid": "POP_DATA",
                                                "home_pings": "pings"})
population_per_area.head(2)
```

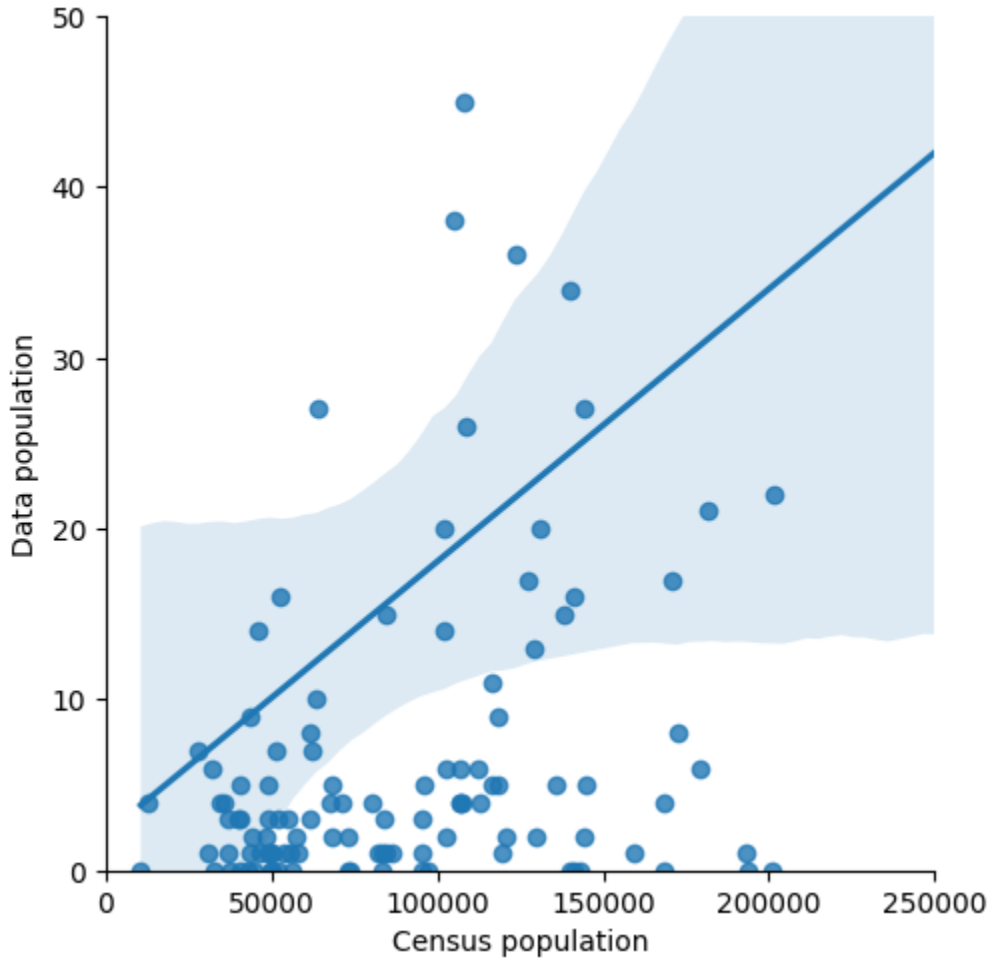
[20]:
```
   tile_ID  POP_DATA   pings
0      0.0         6  1498.0
1      2.0         1     0.0
```

[21]:
```
# Merge with gdf
gdf_areas = pd.merge(tessellation_gdf, population_per_area, on="tile_ID", how="left")
gdf_areas["POP_DATA"] = gdf_areas["POP_DATA"].fillna(0)
gdf_areas.head(4)
```

[21]:
```
   TID     POP      M      F   AGE0   AGE15  AGE65     Address  \
0  257  102402  51818  50584  11322   83441   7639
1  259  168444  94211  74233  17898  144834   5712
2  262   49612  27649  21963   4472   42847   2293
3   92   48076  24392  23684   4975   39191   3910

                                          geometry  tile_ID  POP_DATA  \
0  POLYGON ((116.29603 39.74097, 116.31217 39.771...        0       6.0
1  POLYGON ((116.22050 39.74097, 116.22050 39.786...        1       0.0
2  POLYGON ((116.39314 39.74097, 116.39185 39.750...        2       1.0
3  POLYGON ((116.39185 39.75000, 116.37391 39.764...        3       2.0

    pings
0  1498.0
1     NaN
2     0.0
3     0.0
```

[22]:
```
sns.lmplot(x="POP", y="POP_DATA", data=gdf_areas)
# plt.loglog()
plt.xlim(0, 250000)
plt.ylim(0, 50)

plt.xlabel("Census population")
plt.ylabel("Data population")
```

[22]:
```
Text(24.625000000000007, 0.5, 'Data population')
```



The Mexican case

We load the results of the population analysis in Mexico for the Puebla earthquake and see the agreement between census and mobility estimation at different aggregation levels, from the smallest (AGEB, street blocks) to the largest (Municipios, city level).

Note that these data are not included in the repository to preserve users' privacy.

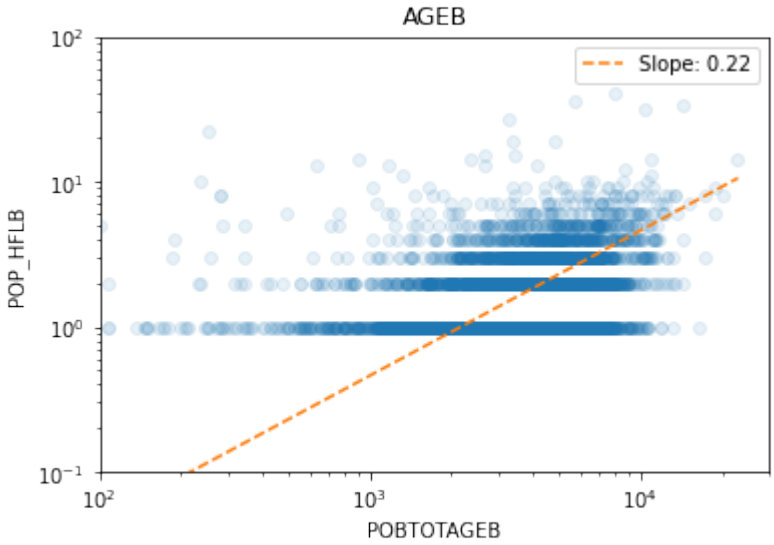
This section is inserted as an example to show the capabilities of the mobility data to measure the population spatial density.

[24]:
```
# Table-preview not shown so as not to disclose original dataset statistics
population_mexico_df = pd.read_csv("../../data/population_estimate_mexico.csv")
population_mexico_df.head(2)
```

[25]:
```
# The smallest aggregations (street block level)
mobilkit.viz.plot_pop(population_mexico_df, "AGEB")
plt.xlim(1e2, 3e4)
plt.ylim(1e-1, 1e2)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.075
Model:                            OLS   Adj. R-squared:                  0.074
Method:                 Least Squares   F-statistic:                     352.2
Date:                Wed, 02 Jun 2021   Prob (F-statistic):           1.25e-75
Time:                        08:16:16   Log-Likelihood:                -208.40
No. Observations:                4369   AIC:                             420.8
Df Residuals:                    4367   BIC:                             433.6
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.5443      0.042    -13.058      0.000      -0.626      -0.463
x1             0.2211      0.012     18.766      0.000       0.198       0.244
==============================================================================
Omnibus:                      454.477   Durbin-Watson:                   1.779
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              610.535
Skew:                           0.857   Prob(JB):                    2.65e-133
Kurtosis:                       3.645   Cond. No.                         41.4
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

[25]:
```
(0.1, 100.0)
```



[26]:
```
# Aggregate to locality (districts) and municipio (city)
population_mexico_df["CVEGEO_LOC"] = population_mexico_df["CVEGEO"].apply(lambda s: s[:-
↪4])
population_mexico_df["CVEGEO_MUN"] = population_mexico_df["CVEGEO_LOC"].apply(lambda s:␣
↪s[:-4])
```

(continues on next page)

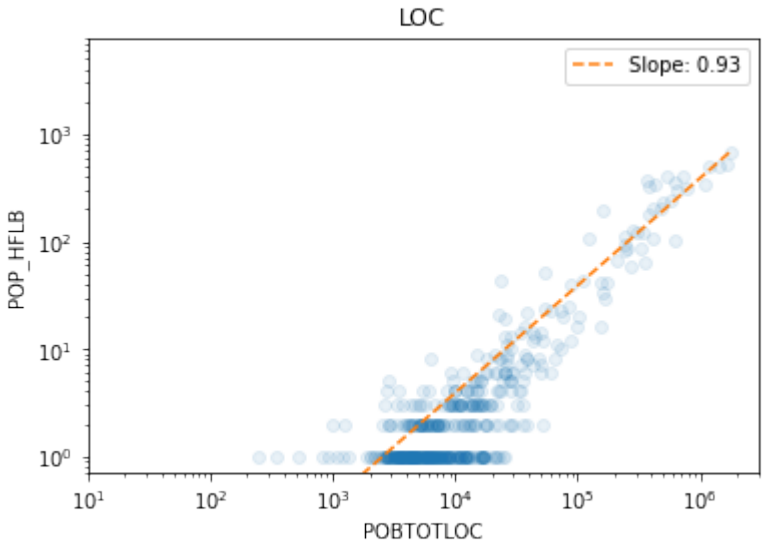
(continued from previous page)

```
urban_areas_loc_gdf = population_mexico_df.groupby("CVEGEO_LOC").agg({
        "POP_HFLB": "sum",
        "POBTOT": "sum",
        "CVEGEO_MUN": "first",
        "CVE_LOC": "first",
        "CVE_ENT": "first",
        "CVE_MUN": "first",
    }).reset_index()

urban_areas_mun_gdf = population_mexico_df.groupby("CVEGEO_MUN").agg({
        "POP_HFLB": "sum",
        "POBTOT": "sum",
        "CVE_ENT": "first",
        "CVE_MUN": "first",
    }).reset_index()
```

[27]:
```
# The locality aggregation level (district level)
mobilkit.viz.plot_pop(urban_areas_loc_gdf, "LOC")
plt.xlim(1e1,3e6)
plt.ylim(7e-1,8e3)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.766
Model:                            OLS   Adj. R-squared:                  0.766
Method:                 Least Squares   F-statistic:                     1535.
Date:                Wed, 02 Jun 2021   Prob (F-statistic):          7.01e-150
Time:                        08:16:24   Log-Likelihood:                -113.13
No. Observations:                 470   AIC:                             230.3
Df Residuals:                     468   BIC:                             238.6
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -3.3410      0.098    -34.198      0.000      -3.533      -3.149
x1             0.9302      0.024     39.185      0.000       0.884       0.977
==============================================================================
Omnibus:                        8.056   Durbin-Watson:                   1.578
Prob(Omnibus):                  0.018   Jarque-Bera (JB):                8.871
Skew:                           0.229   Prob(JB):                       0.0118
Kurtosis:                       3.492   Cond. No.                         29.9
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

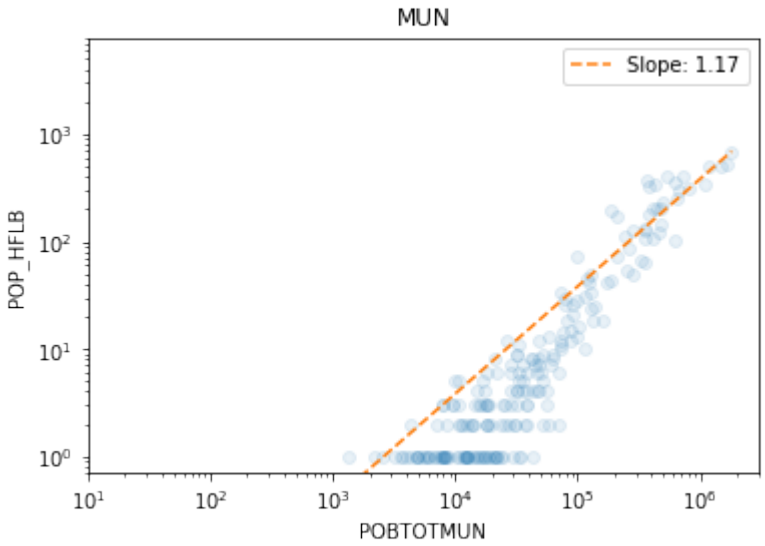
[27]:
```
(0.7, 8000.0)
```



[ ]:
```
# The municipality aggregation level
mobilkit.viz.plot_pop(urban_areas_mun_gdf, "MUN")
plt.xlim(1e1,3e6)
plt.ylim(7e-1,8e3)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.854
Model:                            OLS   Adj. R-squared:                  0.853
Method:                 Least Squares   F-statistic:                     1214.
Date:                Wed, 02 Jun 2021   Prob (F-statistic):           8.83e-89
Time:                        08:16:25   Log-Likelihood:                -52.923
No. Observations:                 210   AIC:                             109.8
Df Residuals:                     208   BIC:                             116.5
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -4.5717      0.156    -29.313      0.000      -4.879      -4.264
x1             1.1737      0.034     34.846      0.000       1.107       1.240
==============================================================================
Omnibus:                        1.063   Durbin-Watson:                   1.894
Prob(Omnibus):                  0.588   Jarque-Bera (JB):                0.891
Skew:                          -0.158   Prob(JB):                        0.640
Kurtosis:                       3.048   Cond. No.                         35.0
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

```
(0.7, 8000.0)
```



[ ]:

6.1.10 Displacement measures

Now we show how to measure the location of users in time.

We have to tell how many initial days to use to determine the original home location.

Then, we tell how many days to use for each window to determine the dynamical home location of each user (and how many pings we want at least for a night to be valid).

[28]:
```
# The parameters of the home location in window function
initial_days_home = 2
home_days_window = 2
start_date=None

# Compute running home location
running_home_df = mobilkit.temporal.homeLocationWindow(dd_week_hw,
                    initial_days_home=initial_days_home,
                    home_days_window=home_days_window,
                    start_date=None, stop_date=None)
```

```
Got the delta days distributed as: count    29331.000000
mean         3.639324
std          2.024058
min          0.000000
25%          2.000000
50%          4.000000
75%          5.000000
max          7.000000
Name: deltaDay, dtype: float64
Doing window 01 / 02
Doing window 02 / 02
```

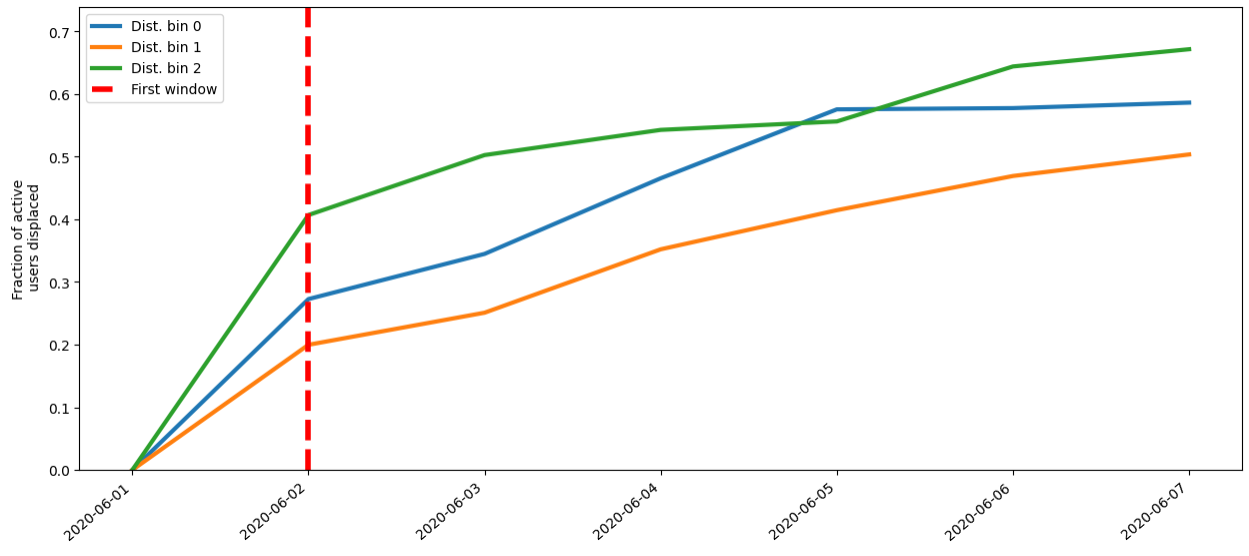
[29]:
```
# We now have for ech user and time window (with its initial date) the location
# where he supposedly spent the night and how many pings are recorded there
running_home_df.head(4)
```

[29]:
```
0  pings  tile_ID  timeSlice  uid               window_date
0      9      106          0    1 2020-06-01 00:00:00+00:00
1    273      106          0    2 2020-06-01 00:00:00+00:00
2    314      106          0    4 2020-06-01 00:00:00+00:00
3    212      106          0    7 2020-06-01 00:00:00+00:00
```

After we determined the residing area for each user/night we use `mobilkit.temporal.computeDisplacementFigures` to get four objects containing the results of the analysis.

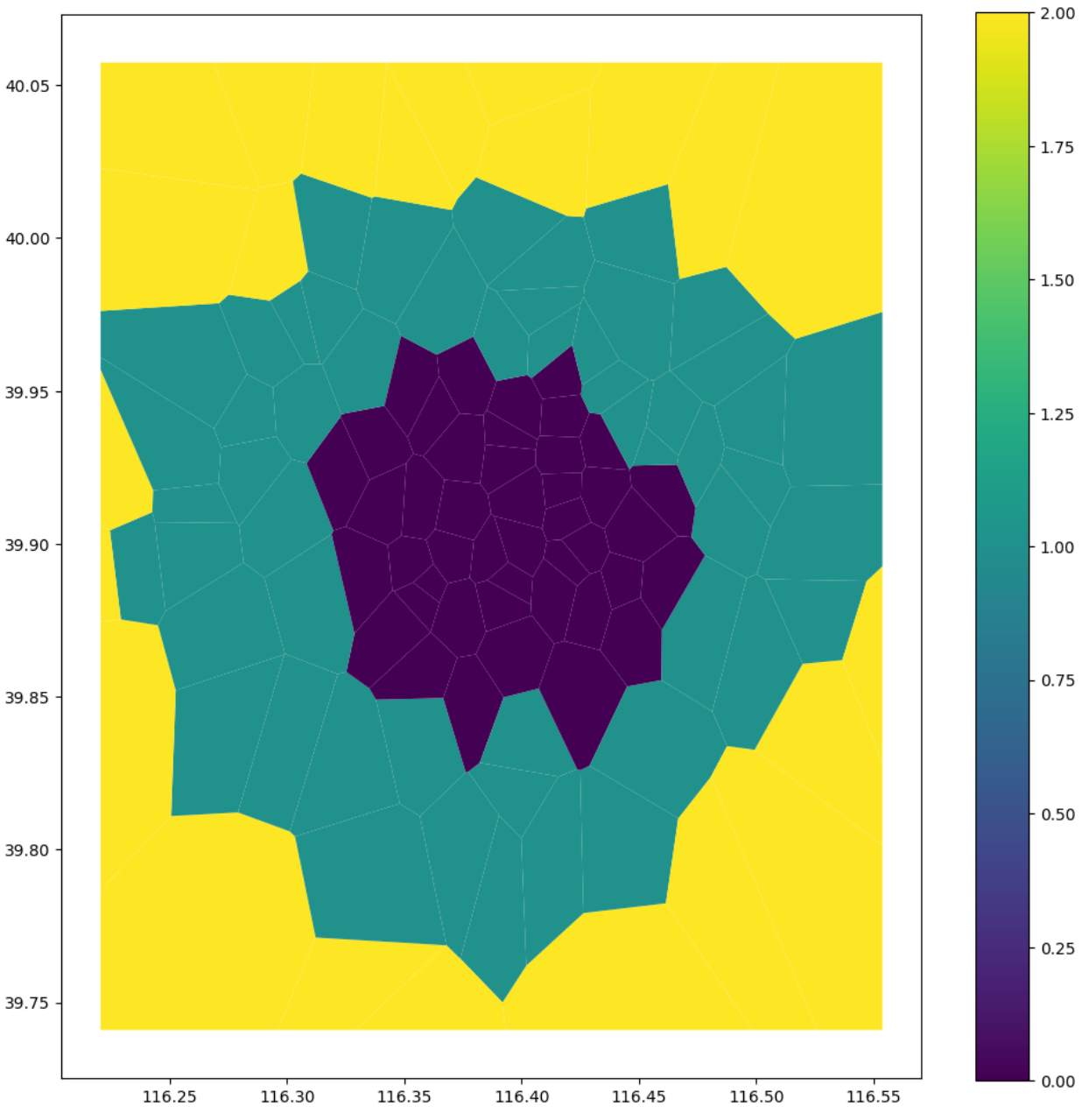
The ones we are interested in are: - `pivoted_df` telling for each night where a user slept; - `count_users_per_area` telling for each area how many users originally residing there were active and how many were displaced on that day;

[30]:
```
# Compute displacement figures

minimum_pings_per_night = 3

pivoted_df, original_home,\
    heaps, count_users_per_area = mobilkit.temporal.computeDisplacementFigures(
        running_home_df, minimum_pings_per_night=minimum_pings_per_night,
)

# The lat lon of the center
epicenter = [tessellation_gdf.unary_union.centroid.xy[1][0],
             tessellation_gdf.unary_union.centroid.xy[0][0]]

# Assess displacement based on distance from epicenter
fig, gdf_enriched = mobilkit.temporal.plotDisplacement(count_users_per_area, pivoted_df,
                                    tessellation_gdf,
                                    epicenter=epicenter,
                                    bins=3)
```



[31]:
```
# Viusualize the distance bins
fig, ax = plt.subplots(1,1,figsize=(12,12))
ax.set_aspect("equal")
gdf_enriched.plot("distance_bin", legend=True, ax=ax)
```

[31]:
```
<Axes: >
```



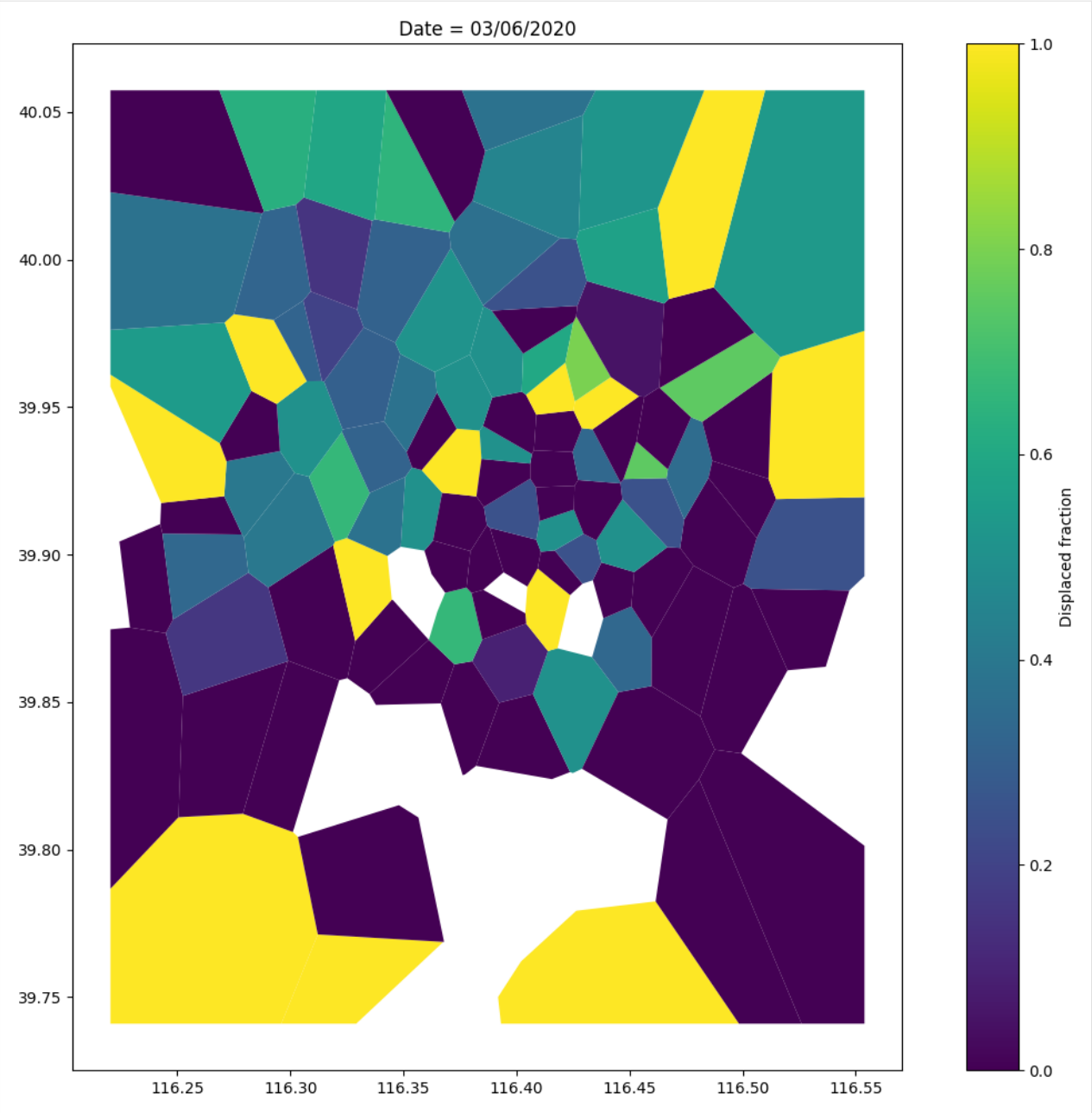
[32]:
```
# Visualize the displacement rate per area on a given date
dates_sorted = sorted(pivoted_df.columns)
selected_date_index = 2

gdf_enriched["displaced_at_date"] = gdf_enriched["tile_ID"].apply(lambda a:
```

(continues on next page)

(continued from previous page)

```
                                                    count_users_per_area[a]["displaced
↪"][selected_date_index]
                                                    / max(1, count_users_per_area[a][
↪"active"][selected_date_index])
                                                    if a in count_users_per_area else␣
↪None)

fig, ax = plt.subplots(1,1,figsize=(15,12))
ax.set_aspect("equal")
gdf_enriched.plot("displaced_at_date", legend=True, ax=ax, legend_kwds={'label':
↪"Displaced fraction"})
plt.title("Date = %s" % dates_sorted[selected_date_index].strftime("%d/%m/%Y"))
```

[32]:
```
Text(0.5, 1.0, 'Date = 03/06/2020')
```



6.1.11 Land use

For this particular analysis we use the daily data because they map to more users (more stats).

We start assigning every ping to a location and then we compute the activity profiles.

[33]:
```
dd_usr_with_zones, tessellation_gdf = mobilkit.spatial.tessellate(dd_users_raw,
                                                                  tesselation_
↪shp=voronoi_file,
                                                                  filterAreas=True)
dd_usr_with_zones.head()
```

[33]:
```
          UTC  acc                 datetime        lat         lng  uid  \
0  1591008784    1 2020-06-01 10:53:04+00:00  39.984702  116.318417    0
1  1591008790    1 2020-06-01 10:53:10+00:00  39.984683  116.318450    0
2  1591008795    1 2020-06-01 10:53:15+00:00  39.984686  116.318417    0
3  1591008800    1 2020-06-01 10:53:20+00:00  39.984688  116.318385    0
4  1591008805    1 2020-06-01 10:53:25+00:00  39.984655  116.318263    0

   tile_ID
0       99
1       99
2       99
3       99
4       99
```

[34]:
```
# This is the time period over which we want to compute the average activity of an area
# normalization="total" tells to the program to normalize the activity of each area␣
↪dividing it by
# the overall volume of pings or users found in all the ROI. See the docs for other␣
↪normalization
# strategies.
selected_profile_period = "day"
total_profiles_df = mobilkit.temporal.computeTemporalProfile(dd_usr_with_zones, timeBin=
↪"H",
                                                             byArea=True,
                                                             profile=selected_profile_
↪period,
                                                            normalization="total").
↪compute()
```

[35]:
```
# We compute the residual activity of users found in a given area
signal_column = "users"
results, mappings =  mobilkit.temporal.computeResiduals(total_profiles_df,
            signal_column=signal_column, profile=selected_profile_period)
```

Finally we try to cluster these profiles in some groups. We use hierarchical clustering of the residual activity profiles using the cosine metric.

This plot tells us the score of the partitioning, the higer the better.

Given that we do not have many data we select n=4 clusters even though we do not have a clear maximum.

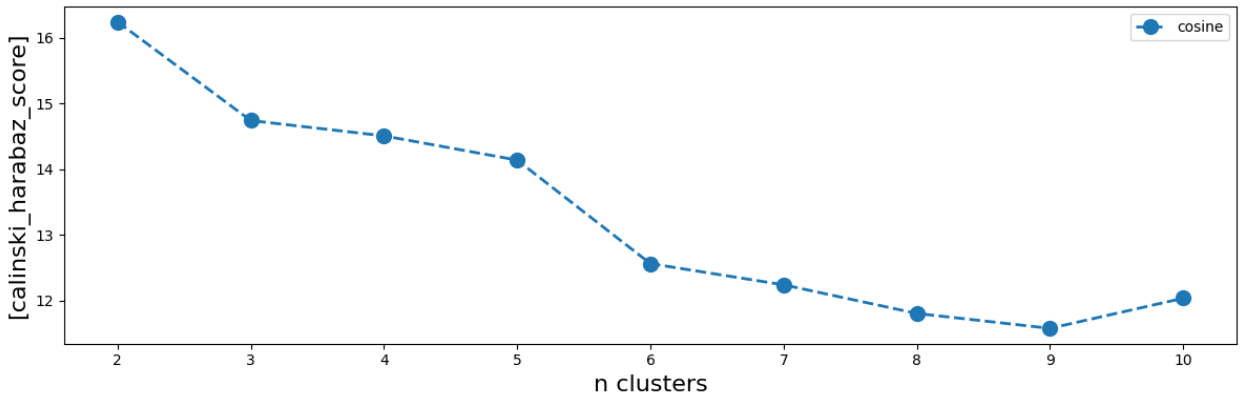
[36]:
```
signal_to_use = "residual"
metric = "cosine"  # The metric to be used in computing the distance matrix
results_clusters = mobilkit.tools.computeClusters(results,
                                                  signal_to_use,
                                                  metric=metric,
                                                  nClusters=range(2,11))

# Visualize score
ax_score = mobilkit.tools.checkScore(results_clusters)
```

```
Done n clusters = 02
Done n clusters = 03
Done n clusters = 04
```

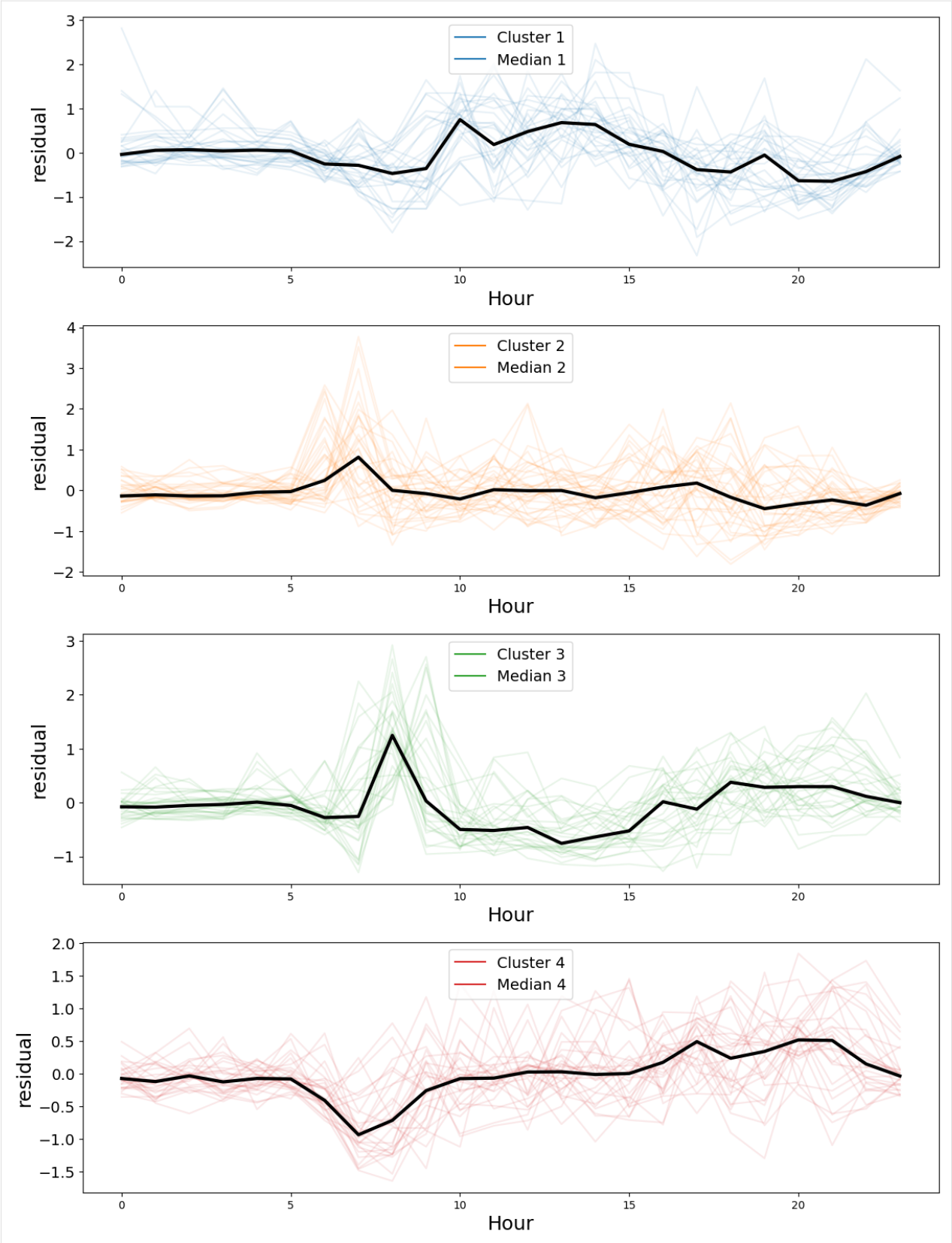
(continues on next page)

(continued from previous page)

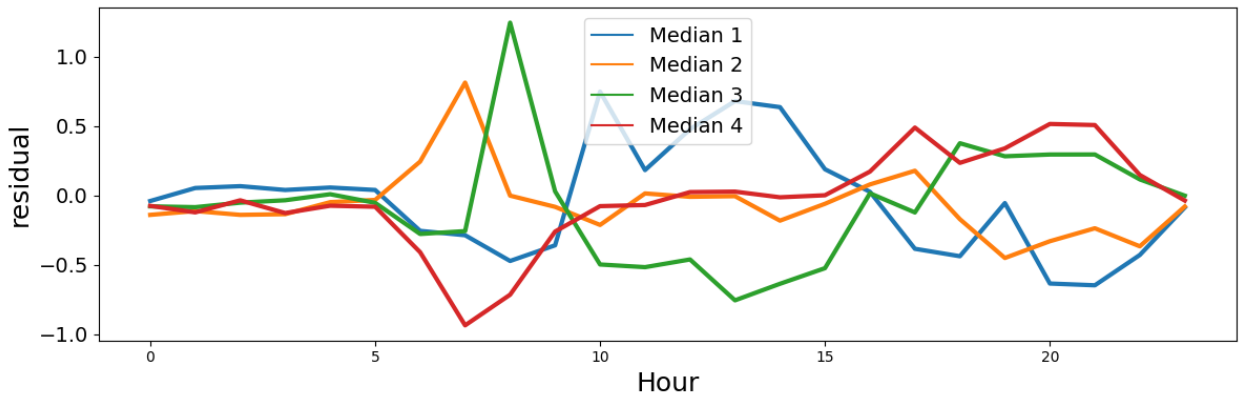
```
Done n clusters = 05
Done n clusters = 06
Done n clusters = 07
Done n clusters = 08
Done n clusters = 09
Done n clusters = 10
```



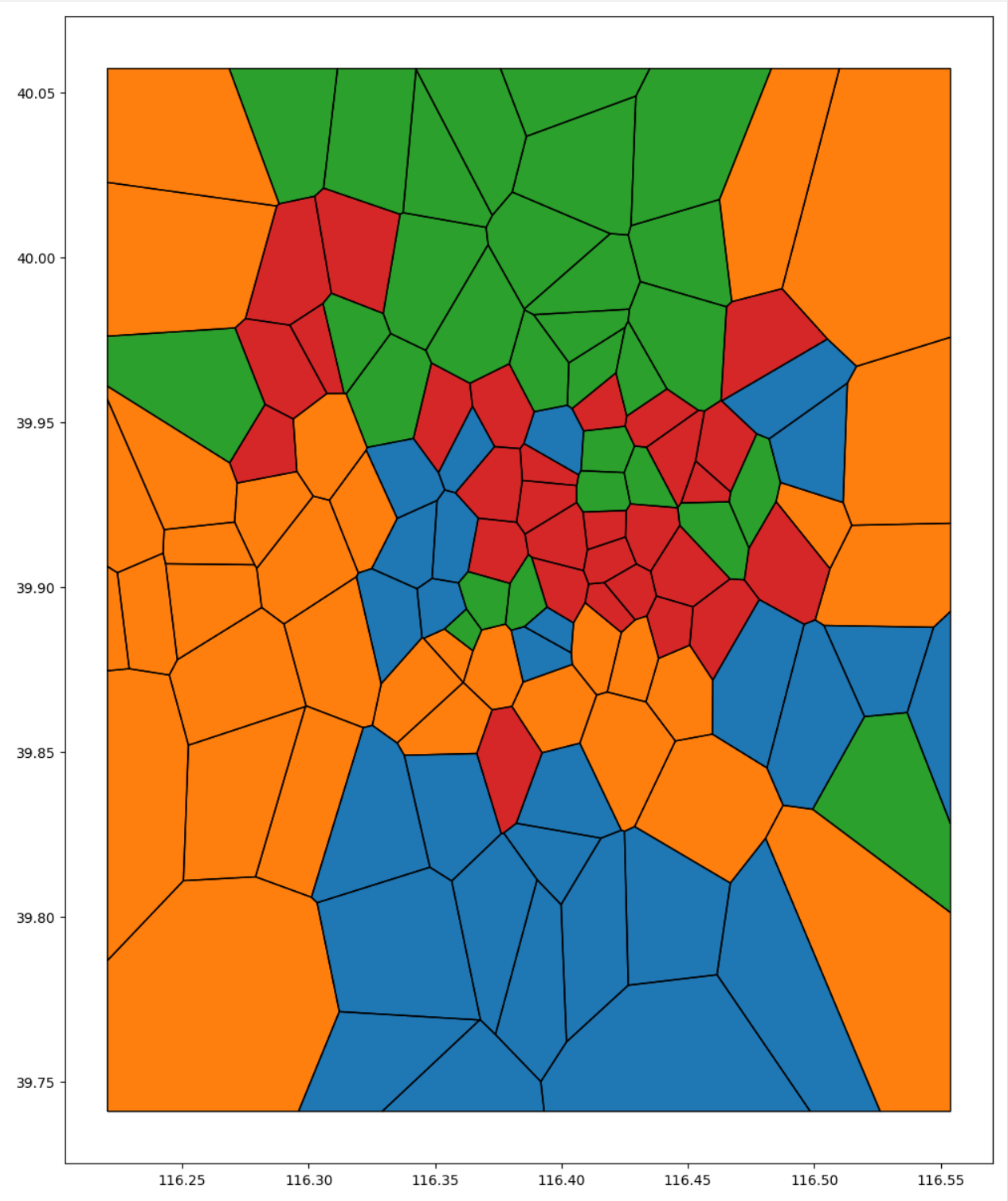
[37]:
```
# Plot clusters profiles and map: We select 4 clusters and plot their profiles and map.
nClusters = 4
ax = mobilkit.tools.visualizeClustersProfiles(results_clusters,
            nClusts=nClusters, showMean=False, showMedian=True, showCurves=True)
```


Cluster 1
Median 1
residual
Hour
Cluster 2
Median 2
residual
Hour
Cluster 3
Median 3
residual
Hour
Cluster 4
Median 4
residual
Hour

[38]:
```
# We compare all the medians activity profiles of the 4 clusters
ax = mobilkit.tools.visualizeClustersProfiles(results_clusters,
            nClusts=nClusters, showMean=False, showMedian=True, showCurves=False,␣
↪together=True)
```



[39]:
```
# We plot the map of the clusters, with the same colors as before.
gdf_update, ax_mappa = mobilkit.tools.plotClustersMap(tessellation_gdf, results_clusters,
                                                      mappings, nClusts=nClusters)
```


40.05
40.00
39.95
39.90
39.85
39.80
39.75
116.25
116.30
116.35
116.40
116.45
116.50
116.55

[ ]:

6.2 Mobility for resilience: population analysis

This notebook shows the preliminary steps done using `mobilkit` to load raw HFLB data, determine the population estimates of each area and prepare the data for displacement and POI visit rates.

We start loading raw HFLB data using the `mobilkit.loader` module.

[1]:
```
%matplotlib inline
%config Completer.use_jedi = False

import os
import sys
from copy import copy, deepcopy
from glob import glob
from collections import Counter
import pytz
from datetime import datetime

import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from mpl_toolkits.axes_grid1 import make_axes_locatable
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()

import numpy as np
import pandas as pd
import seaborn as sns
import geopandas as gpd
import contextily as ctx
import pyproj
from scipy import stats
from sklearn import cluster

import dask
from dask.distributed import Client
from dask import dataframe as dd

### import mobility libraries
import skmob
import mobilkit

sns.set_context("notebook", font_scale=1.5)
```

[10]:
```
dask.__version__ ### tested using Dask version 2020.12.0
```

[10]:
```
'2020.12.0'
```

6.2.1 Load data

Set up Dask

- Notes:
  - Use **Dask** library for high-speed computation on edge computer
    - https://dask.org/
    - accumulates tasks and runs actual computation when ".compute()" is given
  - If cluster computing is available, using PySpark is recommended
- Click the URL of the Dashboard below to monitor progress

[12]:
```
client = Client(address="127.0.0.1:8786") ### choose number of cores to use
client
```

[12]:
```
<Client: 'tcp://192.168.178.34:8786' processes=2 threads=2, memory=28.00 GB>
```

6.2.2 Load raw data using `mobilkit` interface

[16]:
```
datapath = "../../data/"
outpath  = "../../results/"

### define temporal cropping parameters (including these dates)
timezone = "America/Mexico_City"

startdate = "2017-09-04"
enddate   = "2017-10-08"

nightendtime = "09:00:00"
nightstarttime = "18:00:00"

# How to translate the original columns in the mobilkit's nomenclature
colnames = {"id": "uid",
            "gaid": "gaid",
            "hw": "hw",
            "lat": "lat",
            "lon": "lng",
            "accuracy": "acc",
            "unixtime": "UTC",
            "noise": "noise"
           }

# Wehere raw data are stored
filepath = "/data/DataWB/sample/*.part"
ddf = mobilkit.loader.load_raw_files(filepath,
                                     version="wb",
                                     sep=",",
                                     file_schema=colnames,
                                     start_date=startdate,
                                     stop_date=enddate,
```

(continues on next page)

(continued from previous page)

```
                                        timezone=timezone,
                                        header=True,
                                        minAcc=300.,
                                       )
```

Quickly compute min/max of space-time

Use the `mobilkit` and `skmob` column names notations.

[23]:
```
dmin, dmax, lonmin, lonmax, latmin, latmax = dask.compute(ddf.UTC.min(),
                                                          ddf.UTC.max(),
                                                          ddf.lng.min(), ddf.lng.max(),
                                                          ddf.lat.min(), ddf.lat.max()
                                                         )
```

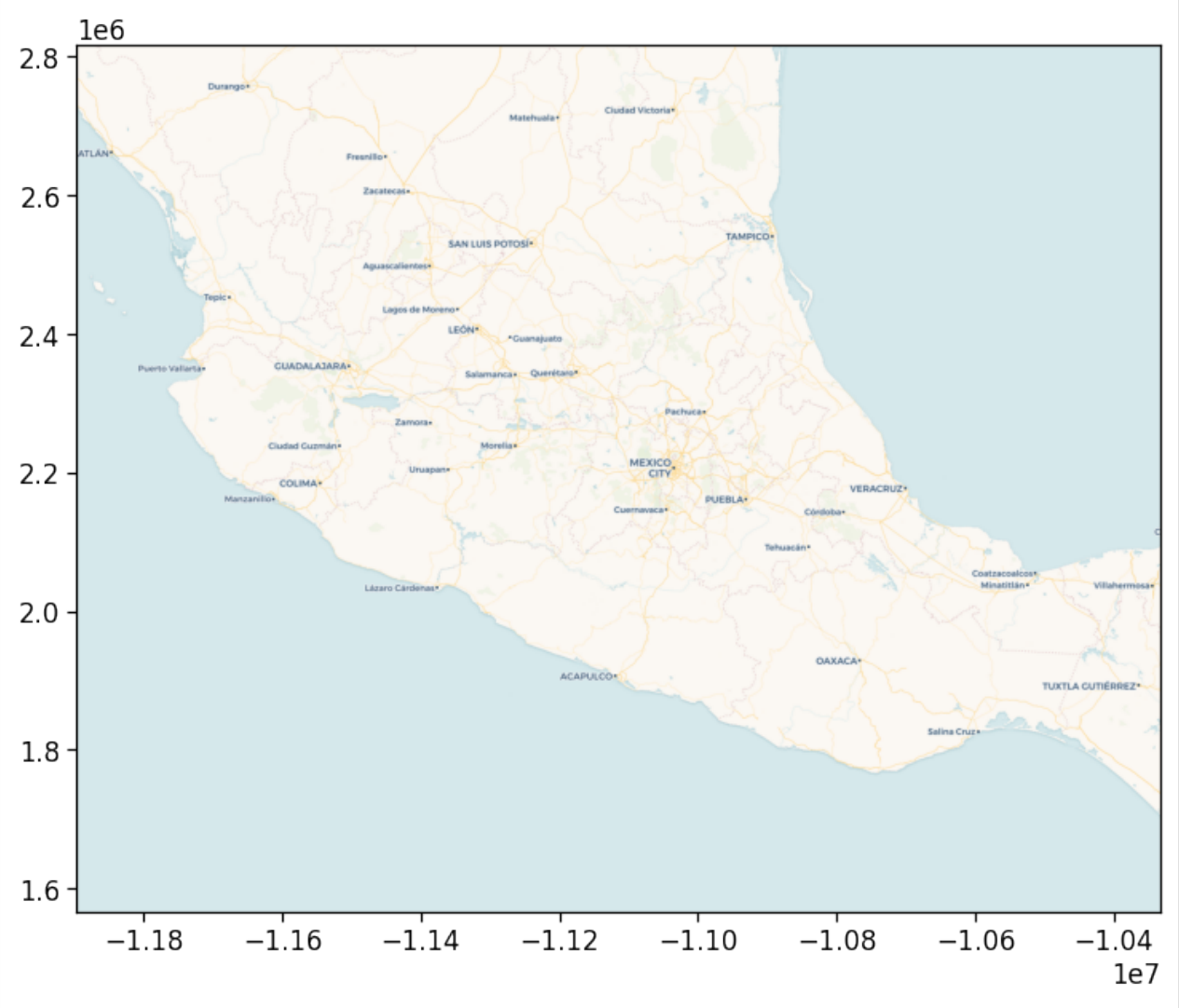
[54]:
```
print(mobilkit.loader.fromunix2fulldate(dmin),
      mobilkit.loader.fromunix2fulldate(dmax))
print(lonmin, lonmax, latmin, latmax)
```

```
2017-09-03 10:07:17 2017-10-09 07:13:57
-105 -95 15.5 22.55
```

[55]:
```
boundary = (lonmin, latmin, lonmax, latmax)
```

[55]:
```
(-105, 15.5, -95, 22.55)
```

[57]:
```
mobilkit.viz.visualize_boundarymap(boundary)
```



Sample of dataset (choose a very small fraction)

[28]:
```
%%time
ddf_sample = ddf.sample(frac=0.0001).compute()
```

```
CPU times: user 11.9 s, sys: 736 ms, total: 12.6 s
Wall time: 5min 11s
```

[29]:
```
len(ddf_sample)
```

[29]:
```
32750
```

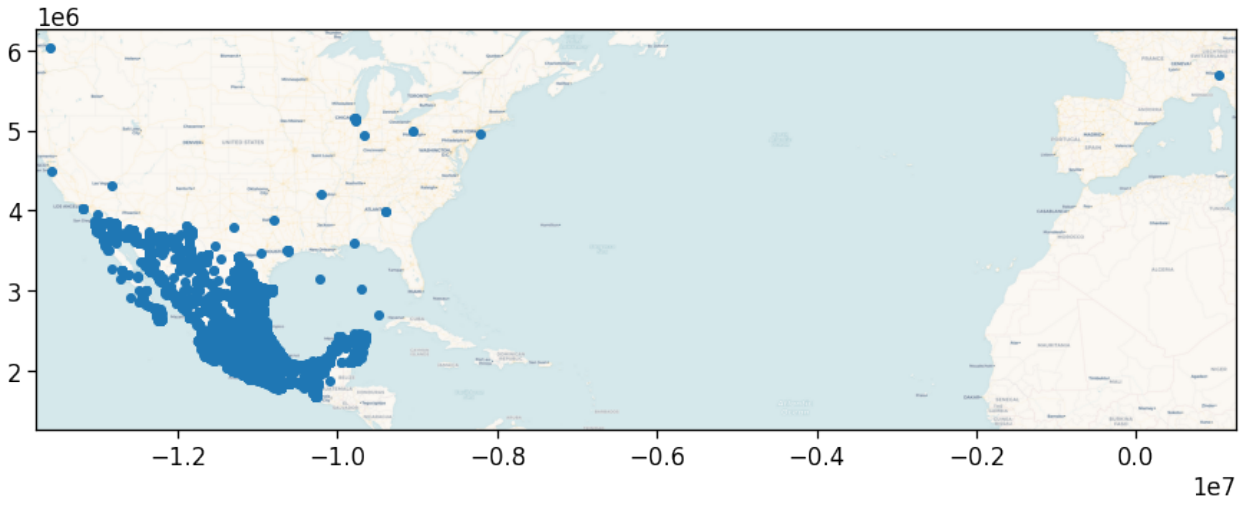
[30]:
```
mobilkit.viz.visualize_simpleplot(ddf_sample)
```

```
/home/ubi/Sandbox/mobilkit_dask/mobenv/lib/python3.9/site-packages/pyproj/crs/crs.py:53:␣
↪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-
```

(continues on next page)

(continued from previous page)

```
↪in-proj-6
  return _prepare_from_string(" ".join(pjargs))
```



6.2.3 Clean data

- Some ideas on data cleaning:
  - **geographical boundary**; analyze data only within a specific area
  - **temporal boundary**; analyze data only within a specific timeframe
  - **Users' data quality**; select users with more than X datapoints, etc.

Geographical boundary

[31]:
```
### define boundary box: (min long, min lat, max long, max lat)
# ==== Parameters === #
bbox = (-106.3, 15.5, -86.3, 29.1)
```

[32]:
```
# ddf_sc = ddf.map_partitions(data_preprocess.crop_spatial, bbox)
ddf_sc = ddf.map_partitions(mobilkit.loader.crop_spatial, bbox)
```

Temporal boundary

These computation gets done automatically when loading now.

We only have to filter night hours.

[34]:
```
nightendtime = "09:00:00"
nightstarttime = "18:00:00"

ddf_tc2 = ddf_sc.map_partitions(mobilkit.loader.crop_time,
                                nightendtime,
```

(continues on next page)

(continued from previous page)

```
                                    nightstarttime,
                                    timezone)
```

select users with sufficient data points

- **users_totalXpoints** : select users with more than X data points throughout entire period
- **users_Xdays** : select users with observations of more than X days
- **users_Xavgps** : select users with more than X observations per day
- **users_Xdays_Xavgps** : select users that satisfy both criteria

[38]:
```
# ==== Parameters === #
mindays = 3
avgpoints = 1

ddf = ddf.assign(uid=ddf["id"])
users_stats = mobilkit.stats.userStats(ddf).compute()
valid_users = set(users_stats[
                (users_stats["avg"] > avgpoints)
                & (users_stats["daysActive"] > mindays)
                ]["uid"].values)

ddf_clean = mobilkit.stats.filterUsersFromSet(ddf, valid_users)
# I do not have this col...
# ddf_clean_homework = ddf_clean[ddf_clean["hw"]=="HOMEWORK"]
# I keep only events during night
ddf_clean_homework = ddf_clean_homework[~ddf_clean_homework["datetime"].dt.hour.
↪between(8,19)]
```

6.2.4 Home location estimation

Estimation using Meanshift

- took ~ 2 hours 15 minutes for entire dataset (mindays=1, avgpoints=0.1)

We compute home location and we later split it into its latitude and longitude.

[42]:
```
id_home = ddf_clean_homework.groupby("uid").apply(mobilkit.spatial.meanshift)\
                                .compute()\
                                .reset_index()\
                                .rename(columns={0:"home"})
toc = datetime.now()
print("Number of IDs with estimated homes: ",len(id_home))
```

```
<ipython-input-42-2437fa6c87f0>:2: UserWarning: `meta` is not specified, inferred from␣
↪partial data. Please provide `meta` if the result is unexpected.
  Before: .apply(func)
  After:  .apply(func, meta={'x': 'f8', 'y': 'f8'}) for dataframe result
  or:     .apply(func, meta=('x', 'f8'))            for series result
  id_home = ddf_clean_homework.groupby("uid").apply(mobilkit.spatial.meanshift)\
```

(continues on next page)

(continued from previous page)

```
/home/ubi/Sandbox/mobilkit_dask/mobenv/lib/python3.9/site-packages/distributed/worker.py:
↪3445: UserWarning: Large object of size 19.15 MB detected in task graph:
  (['a81dbcb8f4c35834d6619a45d67f34d95911fab1318710d ... e5ce12ca182'],)
Consider scattering large objects ahead of time
with client.scatter to reduce scheduler burden and
keep data on workers

    future = client.submit(func, big_data)    # bad

    big_future = client.scatter(big_data)     # good
    future = client.submit(func, big_future)  # good
  warnings.warn(
```

```
Number of IDs with estimated homes:  279541
```

[44]:
```
### save to csv file
id_home.to_csv("../data/"+"id_home_"+str(mindays)+"_"+str(avgpoints).replace(".","")+".
↪csv")
```

[46]:
```
id_home["lon"] = id_home["home"].apply(lambda x : x[0])
id_home["lat"] = id_home["home"].apply(lambda x : x[1])
id_home = id_home.drop(columns=["home"])[["uid","lon","lat"]]
id_home.lon = id_home.lon.astype("float64")
id_home.lat = id_home.lat.astype("float64")
```

[48]:
```
# Create a geodataframe for spatial queries
idhome_gdf = gpd.GeoDataFrame(id_home, geometry=gpd.points_from_xy(id_home.lon, id_home.
↪lat))
```

6.2.5 Compute administrative region for each ID

manzana shape data (for only urban areas)

[49]:
```
### load shape data
areas = ["09_Manzanas_INV2016_shp","17_Manzanas_INV2016_shp",
         "21_Manzanas_INV2016_shp","29_Manzanas_INV2016_shp"]
manz_shp = gpd.GeoDataFrame()
for i,a in enumerate(areas):
    manz_f = "data/spatial/manzanas_shapefiles/"+a+"/"
    manz_shp1 = gpd.read_file(manz_f)
    manz_shp = manz_shp.append(manz_shp1, ignore_index=True)
    print("done",i)
```

```
done 0
done 1
done 2
done 3
```

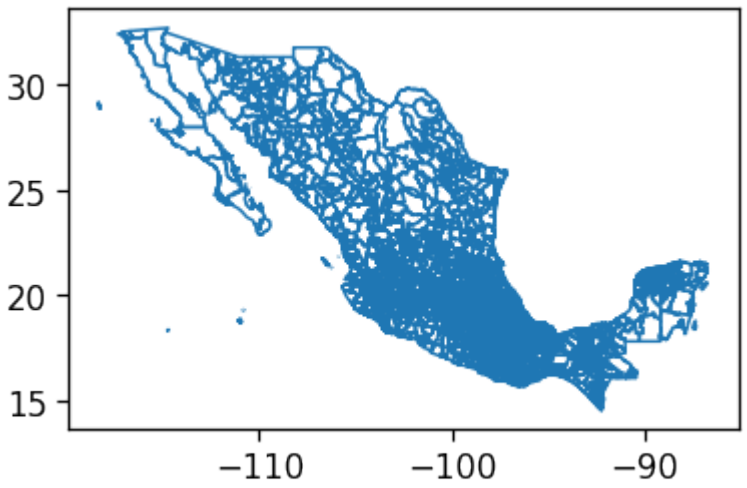
[50]:
```
manz_shp = manz_shp[["geometry","CVEGEO",'ENT','MUN','LOC','AGEB', 'MZA']]
manz_shp.head()
```

[50]:
```
                                              geometry            CVEGEO ENT  \
0  POLYGON ((-99.20644 19.51393, -99.20640 19.513...  0900200010010001  09
1  POLYGON ((-99.20594 19.51418, -99.20586 19.514...  0900200010010002  09
2  POLYGON ((-99.20526 19.51279, -99.20526 19.512...  0900200010010003  09
3  POLYGON ((-99.20563 19.51279, -99.20562 19.512...  0900200010010004  09
4  POLYGON ((-99.20655 19.51278, -99.20653 19.512...  0900200010010005  09

   MUN   LOC  AGEB  MZA
0  002  0001  0010  001
1  002  0001  0010  002
2  002  0001  0010  003
3  002  0001  0010  004
4  002  0001  0010  005
```

By Entidad or Municipio stratification

[51]:
```
adm2_f = datapath+"spatial/boundaries_shapefiles/mex_admbnda_adm2_govmex/"
adm2_shp = gpd.read_file(adm2_f)
adm2_shp.boundary.plot()
```

[51]:
```
<AxesSubplot:>
```



[52]:
```
adm2_shp = adm2_shp[["ADM2_PCODE","geometry"]]
adm2_shp["ent"] = adm2_shp["ADM2_PCODE"].apply(lambda x : x[2:4])
adm2_shp["entmun"] = adm2_shp["ADM2_PCODE"].apply(lambda x : x[2:])
```

[53]:
```
adm2_shp.head()
```

[53]:
```
  ADM2_PCODE                                           geometry ent entmun
0    MX01001  POLYGON ((-102.09775 22.02325, -102.09857 22.0...  01  01001
1    MX01002  POLYGON ((-101.99941 22.21951, -101.99940 22.2...  01  01002
2    MX01003  POLYGON ((-102.57625 21.96778, -102.57626 21.9...  01  01003
3    MX01004  POLYGON ((-102.25320 22.37449, -102.25239 22.3...  01  01004
4    MX01005  POLYGON ((-102.31034 22.03716, -102.30653 22.0...  01  01005
```

Spatial join with manzana data

- compute what geographical boundar each home location is in

[2]:
```
id_manz = gpd.sjoin(idhome_gdf, manz_shp, how="inner", op='within')
id_manz["loc_code"]  = id_manz["CVEGEO"].apply(lambda x : x[:9])
id_manz["ageb_code"] = id_manz["CVEGEO"].apply(lambda x : x[:13])
id_manz["mza_code"]  = id_manz["CVEGEO"].apply(lambda x : x[:16])
id_manz = id_manz.drop(columns=["LOC","AGEB","MZA"])
```

Spatial join with entidad/muncipio data

[3]:
```
id_entmun = gpd.sjoin(idhome_gdf, adm2_shp, how="inner", op='within')
```

6.2.6 Validation using census population data

Population data for all levels

[56]:
```
poppath = datapath+"sociodemographic/populationdata/"
df_pop = pd.DataFrame()
for es in ["09","17","21","29"]:
    pop = poppath+"resultados_ageb_urbana_"+es+"_cpv2010.csv"
    df_pop1 = pd.read_csv(pop)[["entidad","mun","loc","ageb","mza","pobtot"]]
    df_pop = df_pop.append(df_pop1, ignore_index=True)

df_pop["CVEGEO"] = df_pop.apply(lambda row: str(row["entidad"]).zfill(2)+
                                str(row["mun"]).zfill(3)+
                                str(row["loc"]).zfill(4)+
                                str(row["ageb"]).zfill(4)+
                                str(row["mza"]).zfill(3), axis=1)
```

[57]:
```
df_pop.head()
```

[57]:

	entidad	mun	loc	ageb	mza	pobtot	CVEGEO
0	9	0	0	0000	0	8851080	0900000000000000
1	9	2	0	0000	0	414711	0900200000000000
2	9	2	1	0000	0	414711	0900200010000000
3	9	2	1	0010	0	3424	0900200010010000
4	9	2	1	0010	1	202	0900200010010001

Entidad level

[59]:
```
ent_ids = id_entmun.groupby("ent").uid.count().reset_index()
ent_pop = df_pop[(df_pop["mun"]==0) & (df_pop["loc"]==0)
                 & (df_pop["ageb"]=="0000") & (df_pop["mza"]==0)][["entidad","pobtot"]]
ent_pop["ent"] = ent_pop["entidad"].apply(lambda x : str(x).zfill(2))
ent_ids_pop = ent_pop.merge(ent_ids, on="ent")
```

Municipio level

[60]:
```
mun_ids = id_entmun.groupby("entmun").uid.count().reset_index()
mun_pop = df_pop[(df_pop["mun"]!=0) & (df_pop["loc"]==0)
                 & (df_pop["ageb"]=="0000") & (df_pop["mza"]==0)][["CVEGEO","pobtot"]]
mun_pop["entmun"] = mun_pop["CVEGEO"].apply(lambda x : str(x)[:5])
mun_ids_pop = mun_pop.merge(mun_ids, on="entmun")
```

Localidades level

[61]:
```
loc_ids = id_manz.groupby("loc_code").uid.count().reset_index()
loc_pop = df_pop[(df_pop["mun"]!=0) & (df_pop["loc"]!=0)
                 & (df_pop["ageb"]=="0000") & (df_pop["mza"]==0)][["CVEGEO","pobtot"]]
loc_pop["loc_code"] = loc_pop["CVEGEO"].apply(lambda x : str(x)[:9])
loc_ids_pop = loc_pop.merge(loc_ids, on="loc_code")
```

AGEB level

[62]:
```
ageb_ids = id_manz.groupby("ageb_code").uid.count().reset_index()
ageb_pop = df_pop[(df_pop["mun"]!=0) & (df_pop["loc"]!=0)
                 & (df_pop["ageb"]!="0000") & (df_pop["mza"]==0)][["CVEGEO","pobtot"]]
ageb_pop["ageb_code"] = ageb_pop["CVEGEO"].apply(lambda x : str(x)[:13])
ageb_ids_pop = ageb_pop.merge(ageb_ids, on="ageb_code")
```

Manzana level

[63]:
```
mza_ids = id_manz.groupby("mza_code").uid.count().reset_index()
mza_pop = df_pop[(df_pop["mun"]!=0) & (df_pop["loc"]!=0)
                 & (df_pop["ageb"]!="0000") & (df_pop["mza"]!=0)][["CVEGEO","pobtot"]]
mza_pop["mza_code"] = mza_pop["CVEGEO"].apply(lambda x : str(x)[:17])
mza_ids_pop = mza_pop.merge(mza_ids, on="mza_code")
```

6.2.7 Plot census population vs MP data

[64]:
```
def plot_compare(df, ax, title):
    df["logpop"] = np.log10(df["pobtot"])
    df["loguser"] = np.log10(df["uid"])
    df = df.replace([np.inf, -np.inf], np.nan).dropna()
#     for col in set(df["color"].values):
#         df_thiscol = df[df["color"]==col]
    ax.scatter(df["logpop"].values, df["loguser"].values, color="b", s=15)
    c1, i1, s1, p_value, std_err = stats.linregress(df["logpop"].values, df["loguser"].
↪values)
    ax.plot([0,np.max(df["logpop"])*1.1],[i1,i1+np.max(df["logpop"])*1.1*c1],
            linestyle="-", color="gray")
    ax.set_xlim(np.min(df["logpop"]),np.max(df["logpop"])*1.1)
    ax.set_ylim(0,np.max(df["loguser"])*1.1)
```

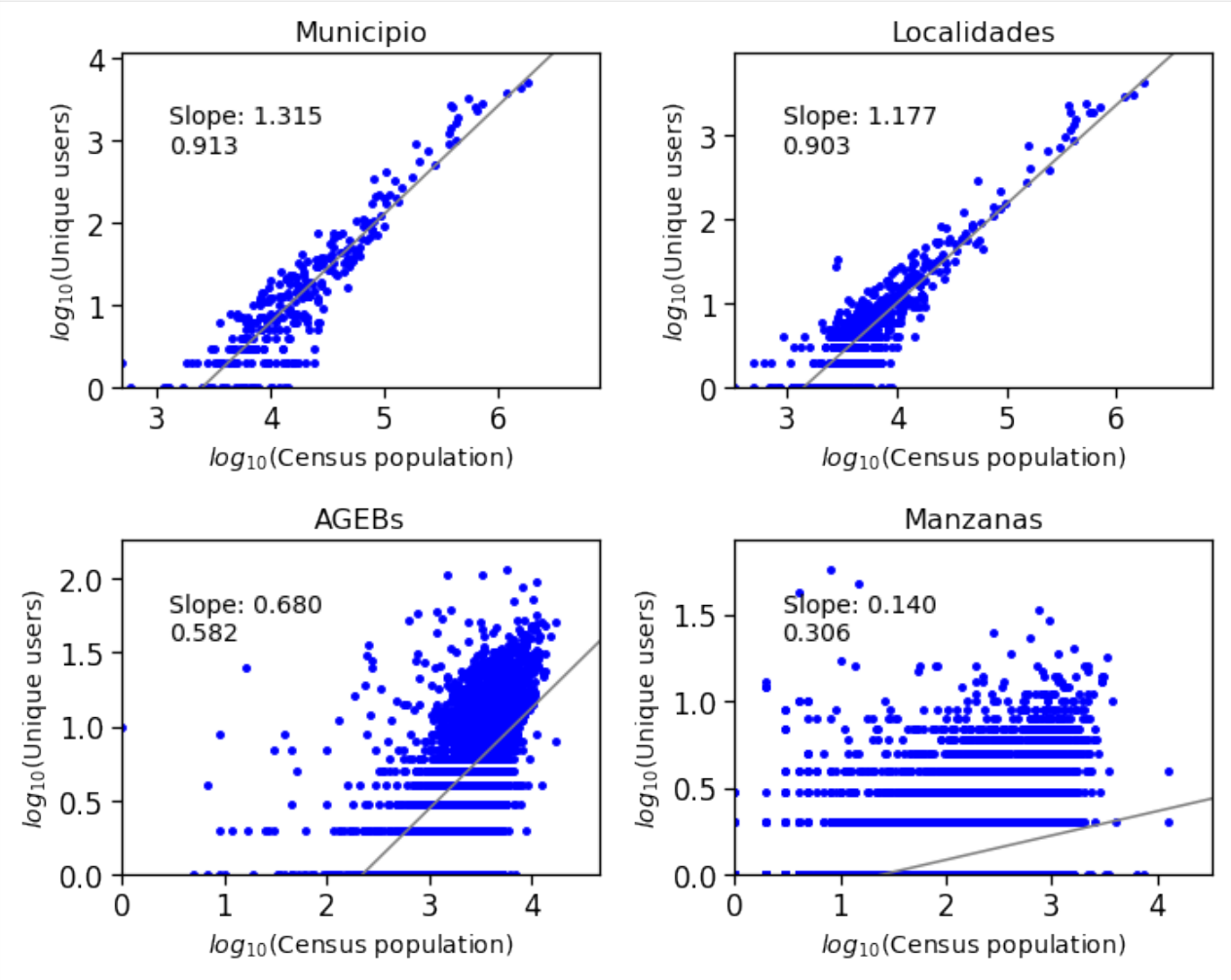
(continues on next page)

(continued from previous page)

```
    ax.set_xlabel(r"$log_{10}$(Census population)", fontsize=14)
    ax.set_ylabel(r"$log_{10}$(Unique users)", fontsize=14)
    ax.annotate("Slope: "+str(c1)[:5]+"\n"+str(s1)[:5], #+utils.stars(p_value),
                        xy=(.1,0.7),
                        xycoords='axes fraction', color="k", fontsize=14)
    ax.set_title(title, fontsize=16)
```

[65]:
```
fig = plt.figure(figsize=(10,8))
gs=GridSpec(2,2)

ax0 = fig.add_subplot(gs[0,0])
plot_compare(mun_ids_pop, ax0, "Municipio")

ax1 = fig.add_subplot(gs[0,1])
plot_compare(loc_ids_pop, ax1, "Localidades")

ax2 = fig.add_subplot(gs[1,0])
plot_compare(ageb_ids_pop, ax2, "AGEBs")

ax3 = fig.add_subplot(gs[1,1])
plot_compare(mza_ids_pop, ax3, "Manzanas")

plt.tight_layout()
# plt.savefig(outpath+"represent_manzana_eq.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```

```
/home/ubi/Sandbox/mobilkit_dask/mobenv/lib/python3.9/site-packages/pandas/core/arraylike.
↪py:274: RuntimeWarning: divide by zero encountered in log10
  result = getattr(ufunc, method)(*inputs, **kwargs)
/home/ubi/Sandbox/mobilkit_dask/mobenv/lib/python3.9/site-packages/pandas/core/arraylike.
↪py:274: RuntimeWarning: divide by zero encountered in log10
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

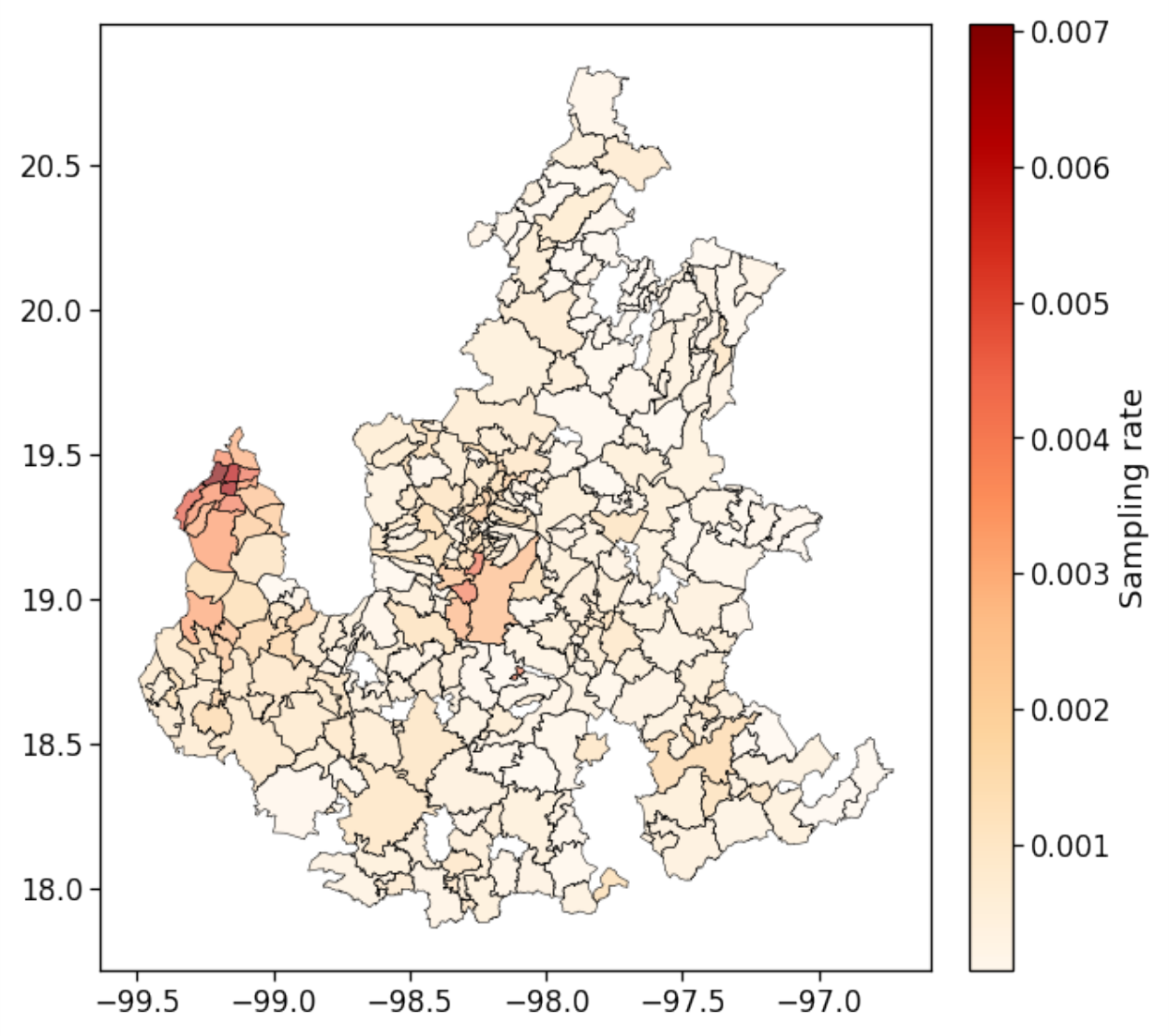


6.2.8 Plot population on map

[66]:
```
mun_ids_pop["rate"] = mun_ids_pop["uid"]/mun_ids_pop["pobtot"]
mun_ids_pop["pcode"] = mun_ids_pop["entmun"].apply(lambda x : "MX"+str(x))

mun_ids_pop_shp = adm2_shp.merge(mun_ids_pop, on="entmun", how="right")
```

[68]:
```
fig,ax = plt.subplots(figsize=(10,10))
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.1)
mun_ids_pop_shp.plot(ax=ax, column='rate', cmap='OrRd', legend=True,
                     cax=cax, legend_kwds={'label': "Sampling rate"}, alpha=0.65)
mun_ids_pop_shp.boundary.plot(ax=ax, color="k", linewidth=0.5)
```

[68]:
```
<AxesSubplot:>
```



[ ]:

6.3 Mobility for resilience: displacement analysis

This notebook shows how to transform raw mobility data to a displacement analsysis using `mobilkit`.

We start loading raw HFLB data using the `mobilkit.loader` module.

Then, we import a shapefile to tessellate data and dynamically analize where people spend time during night before and after a major event (Puebla 2017 earthquake in Mexico). Different stratification (spatial and socio-economic) of the displacement rate are shown.

[1]:
```
%config Completer.use_jedi = False
%matplotlib inline

import numpy as np
```

(continues on next page)

(continued from previous page)

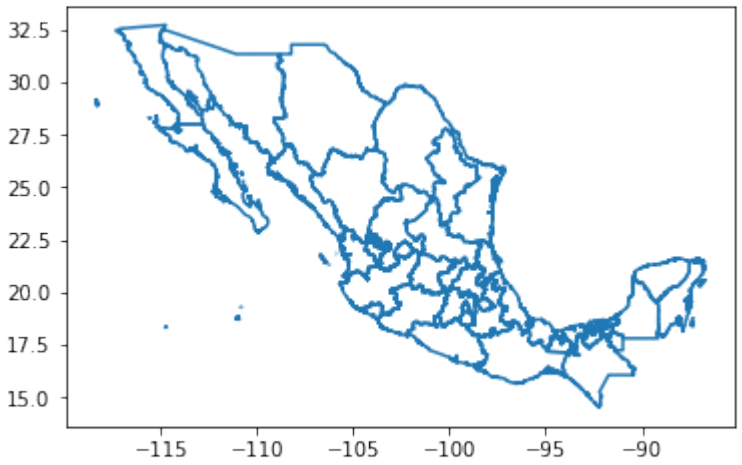
```
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib.gridspec import GridSpec
from matplotlib.dates import DateFormatter

import glob, os
from datetime import datetime as dt
from datetime import timedelta, datetime
from datetime import timezone
import pytz
from math import sin, cos, sqrt, atan2, radians
from scipy.optimize import minimize
from scipy import stats

### import Dask library (https://dask.org/)
import dask
import dask.dataframe as dd
from dask import delayed
from dask.diagnostics import ProgressBar
from dask.distributed import Client, LocalCluster

### import geospatial libraries
import geopandas as gpd
from haversine import haversine
import contextily as ctx
import pyproj

### directory that contains dataset(s) you want to analyze
filepath = "/data/WB_Mexico/gpsdata_eq/testdata_all/"
datapath = "../../data/"
outpath  = "../../results/"
```

[4]:
```
import warnings
warnings.filterwarnings('ignore')
```

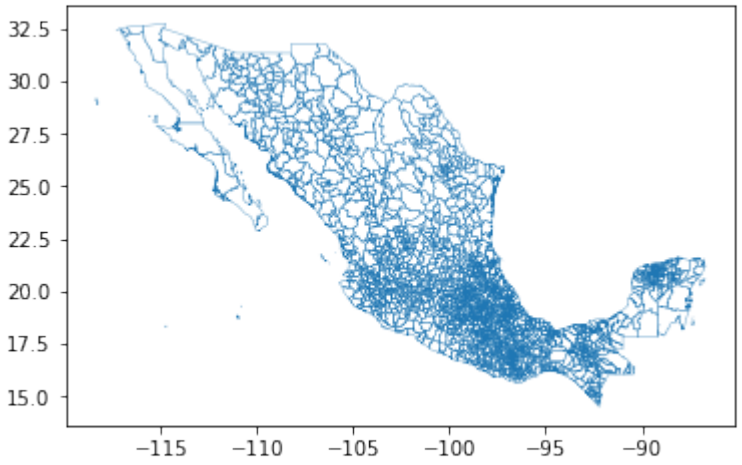
6.3.1 Import external data

Administrative boundary shapefiles

[5]:
```
fig,ax = plt.subplots()
adm1_f = datapath+"spatial/boundaries_shapefiles/mex_admbnda_adm1_govmex/"
adm1_shp = gpd.read_file(adm1_f)
adm1_shp.boundary.plot(ax=ax)
plt.show()
```



[33]:
```
fig,ax = plt.subplots()
adm2_f = datapath+"spatial/boundaries_shapefiles/mex_admbnda_adm2_govmex/"
adm2_shp = gpd.read_file(adm2_f)
adm2_shp = adm2_shp[["ADM2_PCODE","ADM2_ES","geometry"]]
adm2_shp.boundary.plot(ax=ax, linewidth=.3)
plt.show()
```



[34]:
```
adm2_shp.head()
```

[34]:
```
  ADM2_PCODE         ADM2_ES  \
0    MX01001  Aguascalientes
1    MX01002        Asientos
2    MX01003        Calvillo
3    MX01004             Cos
4    MX01005             Jes

                                            geometry
0  POLYGON ((-102.09775 22.02325, -102.09857 22.0...
1  POLYGON ((-101.99941 22.21951, -101.99940 22.2...
2  POLYGON ((-102.57625 21.96778, -102.57626 21.9...
3  POLYGON ((-102.25320 22.37449, -102.25239 22.3...
```

(continues on next page)

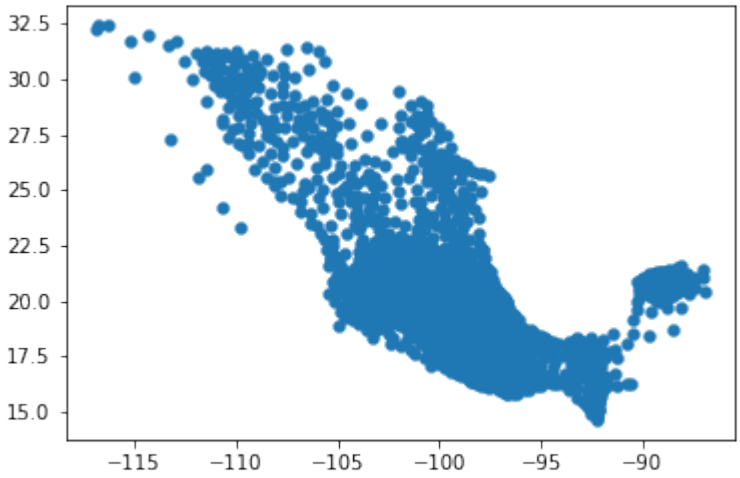
(continued from previous page)

```
4  POLYGON ((-102.31034 22.03716, -102.30653 22.0...
```

[38]:
```
# Turn into a centroid
adm2_shp["centroid"] = adm2_shp.centroid
adm2_shp.geometry = adm2_shp.centroid
```

[40]:
```
fig,ax = plt.subplots()
adm2_shp.plot(ax=ax, linewidth=.3)
```

[40]:
```
<AxesSubplot:>
```



Seismic intensity shapefile

[41]:
```
seismic_shp_f = datapath+"spatial/seismicdata/intensity/"
seismic_shp = gpd.read_file(seismic_shp_f)[["PARAMVALUE","geometry"]]
```

[42]:
```
seismic_shp.tail()
```

[42]:
```
    PARAMVALUE                                           geometry
20         7.0  MULTIPOLYGON (((-99.00292 19.21667, -99.00471 ...
21         7.2  MULTIPOLYGON (((-98.74806 18.85833, -98.74864 ...
22         7.4  MULTIPOLYGON (((-98.67707 18.80000, -98.67940 ...
23         7.6  MULTIPOLYGON (((-98.44767 18.70000, -98.44861 ...
24         7.8  MULTIPOLYGON (((-98.48333 18.39997, -98.48336 ...
```

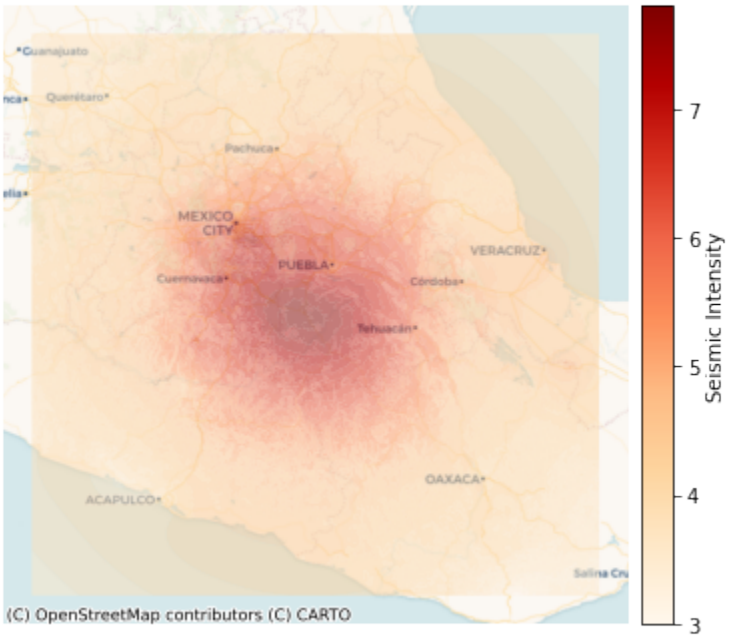
[43]:
```
seismic_shp_hm = seismic_shp.to_crs(epsg=3857)
```

[44]:
```
fig,ax = plt.subplots(1,1,figsize=(6,6))
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.1)
seismic_shp_hm.plot(ax=ax, column='PARAMVALUE', legend=True, cmap='OrRd',
                    cax=cax, legend_kwds={'label': "Seismic Intensity"},
                    zorder=2.5, alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
ctx.add_basemap(ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
plt.show()
```



[45]:
```
adm2_SI = gpd.sjoin(adm2_shp, seismic_shp, how="left", \
                    op='intersects')[["ADM2_PCODE","PARAMVALUE"]]
```

[46]:
```
adm2_SI[adm2_SI["ADM2_PCODE"]=="MX09002"]
```

[46]:
```
    ADM2_PCODE  PARAMVALUE
265    MX09002         6.0
```

Population data

[47]:
```
poppath = datapath+"sociodemographic/populationdata/"
df_pop = pd.DataFrame()
# Load only the states we are interested in
for es in ["09","17","21","29"]:
    pop = poppath+"resultados_ageb_urbana_"+es+"_cpv2010.csv"
    df_pop1 = pd.read_csv(pop)[["entidad","mun","loc","ageb","mza","pobtot"]]
    df_pop = df_pop.append(df_pop1, ignore_index=True)

df_pop = df_pop[(df_pop["mun"]!=0) & (df_pop["loc"]==0)][["entidad","mun","pobtot"]]

df_pop["PCODE"] = df_pop.apply(lambda row : "MX"+str(row["entidad"]).zfill(2)+str(row[
↪"mun"]).zfill(3), axis=1)
```

[50]:
```
df_pop.head()
```

[50]:
```
       entidad  mun   pobtot    PCODE
1            9    2   414711  MX09002
3097         9    3   620416  MX09003
7970         9    4   186391  MX09004
9010         9    5  1185772  MX09005
17648        9    6   384326  MX09006
```

[51]:
```
adm2_SI_pop = adm2_SI.merge(df_pop, left_on="ADM2_PCODE", right_on="PCODE")[["PCODE",
↪"PARAMVALUE","pobtot"]]
```

[52]:
```
adm2_SI_pop.head()
```

[52]:
```
     PCODE  PARAMVALUE   pobtot
0  MX09002         6.0   414711
1  MX09003         6.8   620416
2  MX09004         6.2   186391
3  MX09005         5.4  1185772
4  MX09006         6.6   384326
```

Wealth index data

[53]:
```
wealthidx_f = datapath+"sociodemographic/wealthindex/pca_index_AGEBS_localidades.csv"
wealthidx = pd.read_csv(wealthidx_f, header=0,
                        names = ["index","code","pca","index_pca"])
wealthidx.head()
```

```
/home/ubi/Sandbox/mobilkit_dask/mobenv/lib/python3.9/site-packages/IPython/core/
↪interactiveshell.py:3146: DtypeWarning: Columns (1) have mixed types.Specify dtype␣
↪option on import or set low_memory=False.
  has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
```

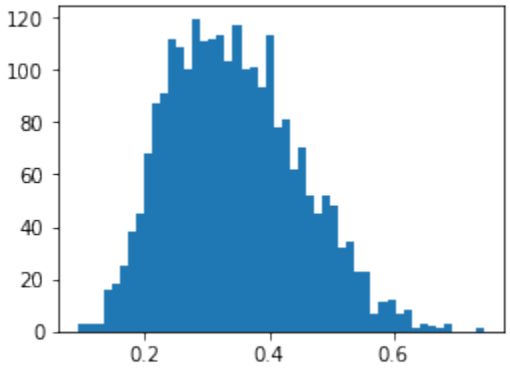
[53]:
```
   index           code       pca  index_pca
0      0  0100100010229 -2.680167   0.147990
1      1  0100100010233 -2.701735   0.146480
2      2  0100100010286 -3.474532   0.092363
3      3  0100100010290 -3.404371   0.097277
4      4  0100100010303 -3.099987   0.118592
```

[54]:
```
wealthidx["PCODE"] = wealthidx["code"].apply(lambda x : "MX"+str(x)[:5])
wealthidx_avg = wealthidx.groupby("PCODE")["index_pca"].mean().reset_index()
```

[56]:
```
wealthidx_avg.head()
```

[56]:
```
     PCODE  index_pca
0  MX01001   0.206558
1  MX01002   0.232278
2  MX01003   0.233212
3  MX01004   0.216361
4  MX01005   0.216503
```

[57]:
```
plt.figure(figsize=(4,3))
plt.hist(wealthidx_avg["index_pca"].values, bins=50)
plt.show()
```



[58]:
```
# Merge this info in the code mapping df
adm2_SI_pop_WI = adm2_SI_pop.merge(wealthidx_avg, on="PCODE")
```

[59]:
```
adm2_SI_pop_WI.head(10)
```

[59]:

	PCODE	PARAMVALUE	pobtot	index_pca
0	MX09002	6.0	414711	0.124537
1	MX09003	6.8	620416	0.106024
2	MX09004	6.2	186391	0.182998
3	MX09005	5.4	1185772	0.140297
4	MX09006	6.6	384326	0.134230
5	MX09007	7.0	1815786	0.157710
6	MX09008	6.0	239086	0.181795
7	MX09009	7.0	130582	0.285495
8	MX09010	6.8	727034	0.135709
9	MX09011	7.4	360265	0.199308

6.3.2 Compute displacement rate

Get valid user IDs

Filter users based on statistic.

[1]:
```
idhome = "data/id_home_3_1.csv"
df_idhome = pd.read_csv(idhome)
df_idhome["home"] = df_idhome["home"].apply(lambda v: [e for e in v.replace("[","")
                                                       .replace("]","")
                                                       .split(" ")
                                                      if len(e)>0])
df_idhome["homelat"] = df_idhome["home"].apply(lambda v: float(v[1]))
df_idhome["homelon"] = df_idhome["home"].apply(lambda v: float(v[0]))
df_idhome = df_idhome[["uid","homelat","homelon"]].copy()
allids = set(df_idhome["uid"].values)
```

[7]:
```
df_idhome.shape
```
[7]:
```
(279541, 3)
```

[9]:
```
len(allids)
```
[9]:
```
279541
```

6.3.3 Extract data of above IDs

We just lazily load the data and then filter on the ids. We get for free the localized datetime column.

If you want to persist these data separated per user at the different steps we show how to do it.

We connect to dask and then load and filter data.

[12]:
```
client = Client(address="127.0.0.1:8786", )
```
[12]:
```
{'tcp://127.0.0.1:38661': {'status': 'OK'},
 'tcp://127.0.0.1:39151': {'status': 'OK'},
 'tcp://127.0.0.1:39553': {'status': 'OK'},
 'tcp://127.0.0.1:40421': {'status': 'OK'},
 'tcp://127.0.0.1:41351': {'status': 'OK'},
 'tcp://127.0.0.1:42609': {'status': 'OK'},
 'tcp://127.0.0.1:44517': {'status': 'OK'},
 'tcp://127.0.0.1:45691': {'status': 'OK'}}
```

[13]:
```
client
```
[13]:
```
<Client: 'tcp://192.168.178.34:8786' processes=8 threads=8, memory=32.00 GB>
```

[14]:
```
tz = pytz.timezone("America/Mexico_City")
alldataf = dd.read_parquet("/data/datiHFLBPARQUET/")
filtered_dataf = mobilkit.stats.filterUsersFromSet(alldataf, allids)
```

[19]:
```
if False:
    # Now we can persist these data as in the original example
    # I prefer to use the parquet format which is faster
    alldataf = "../../results/displacement_selectedids_all_data"
    filtered_dataf.repartition(partition_size="20M").to_parquet(alldataf)
```

[2]:
```
# Now I can quickly reload this first step of selection
alldataf = "../../results/displacement_selectedids_all_data"
filtered_dataf_reloaded = dd.read_parquet(alldataf).repartition(partition_size="200M")
if "datetime" not in filtered_dataf_reloaded.columns:
    # Add datetime column
    import pytz
    tz = pytz.timezone("America/Mexico_City")
    # Filter on dates...
    filtered_dataf_reloaded = mobilkit.loader.filterStartStopDates(filtered_dataf_
↪reloaded,
                                                   start_date="2017-09-04",
                                                     stop_date="2017-10-08",
```

(continues on next page)

(continued from previous page)

```
                                                    tz=tz,)
    filtered_dataf_reloaded = mobilkit.loader.compute_datetime_col(filtered_dataf_
↪reloaded, selected_tz=tz)
```

6.3.4 Get daily displacement distance

All these computing times are obtained on a personal laptop local cluster with:

```
Client
Scheduler: tcp://127.0.0.1:8786
Dashboard: http://127.0.0.1:8787/status
Cluster
Workers: 3
Cores: 3
Memory: 28.00 GB
```

for limited I/O performances. These should scale better on a cluster.

[28]:
```
# Prepare pings adding date and filtering on hour...
df_displacement_ready = mobilkit.temporal.filter_daynight_time(
                                                filtered_dataf_reloaded,
                                                filter_to_h=9,
                                                filter_from_h=21,
                                                previous_day_until_h=4,
                                            )
```

[28]:
```
# We now compute the displacement figures in one line and save it to disk
processed_diplacement = mobilkit.displacement.calc_displacement(df_displacement_ready,
                                                                df_idhome)
```

[29]:
```
# Persist to disk
tic = datetime.now()
if False:
    processed_diplacement.to_parquet("../../results/displacement_selectedids_processed/")
else:
    processed_diplacement = dd.read_parquet("../../results/displacement_selectedids_
↪processed/")
toc = datetime.now()
```

[30]:
```
tot_sec = (toc - tic).total_seconds()
print("Done in %d hours and %.01f minutes!" % (tot_sec//3600, (tot_sec % 3600)/60))
```

```
Done in 2 hours and 21.7 minutes!
```

[31]:
```
# Total number of users and number of pings
stats_df = filtered_dataf_reloaded.groupby("uid").agg("count").compute()
print("Users:", stats_df.shape[0])
print("Pings:", stats_df["lat"].sum())
```

```
Users: 279541
Pings: 318852179
```

6.3.5 Analyze displacement rates

Per-id home location

[60]:
```
# Transform the data in a geodataframe for spatial queries
idhome_gdf = gpd.GeoDataFrame(df_idhome,
                              geometry=gpd.points_from_xy(df_idhome.homelon,
                                                          df_idhome.homelat))
```

[121]:
```
adm2_f = datapath + "spatial/boundaries_shapefiles/mex_admbnda_adm2_govmex/"
adm2_shp = gpd.read_file(adm2_f)
```

[63]:
```
# Spatial join, then I can aggregate by Municipality or other features
id_homecode = gpd.sjoin(idhome_gdf,adm2_shp[["ADM2_PCODE","geometry"]])
id_homecode = id_homecode[["uid","homelon",
                           "homelat","ADM2_PCODE"]].rename(columns={"ADM2_PCODE":"PCODE"}
↪)
```

```
<ipython-input-63-ec4458acad15>:1: 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

  id_homecode = gpd.sjoin(idhome_gdf,adm2_shp)
```

[3]:
```
id_home_feat = id_homecode.merge(adm2_SI_pop_WI, on="PCODE")
```

[66]:
```
muncode_count = id_homecode.groupby("PCODE").count().reset_index()
```

[3]:
```
muncode_rate = muncode_count.merge(adm2_SI_pop_WI, on ="PCODE")
muncode_rate["rate"] = muncode_rate["uid"]/muncode_rate["pobtot"]
```

6.3.6 Macroscopic analysis

Reload previous results and stratify by different user status.

Seismic intensity

[71]:
```
df_disp = dd.read_parquet("results/displacement_selectedids_processed/")
```

[72]:
```
# Now we are working on dask, I port to pandas with .compute()
df_disp2 = df_disp.merge(id_homecode, on="uid", how="left").compute()
```

[74]:
```
df_disp3 = df_disp2.merge(adm2_SI_pop_WI, on="PCODE", how="left")
```

[76]:
```
# Helper function to determine the Seismic intensity level
def categorizeSI(si):
    if si>=7:
        r = 7
    elif si>=6.5:
        r = 6.5
    elif si>=6:
        r = 6
    elif si >=5:
        r = 5
    elif si >=4:
        r = 4
    else:
        r = 0
    return r
```

[77]:
```
df_disp3["SI_cat"] = df_disp3["PARAMVALUE"].apply(lambda x : categorizeSI(x))
```

Compute displacement rates

[4]:
```
df_disp4 = df_disp3[df_disp3["lng"]!=0].copy()
```

[80]:
```
dist = "mindist"
df_disp4["500m"] = df_disp4[dist].apply(lambda x : 1 if x>0.5 else 0)
df_disp4["1km"] = df_disp4[dist].apply(lambda x : 1 if x>1 else 0)
df_disp4["3km"] = df_disp4[dist].apply(lambda x : 1 if x>3 else 0)
df_disp4["5km"] = df_disp4[dist].apply(lambda x : 1 if x>5 else 0)
df_disp4["10km"] = df_disp4[dist].apply(lambda x : 1 if x>10 else 0)
```

[5]:
```
si_count = df_disp4[df_disp4["date"]==dt(2017,9, 3)]\
                                    .groupby('SI_cat')\
                                    .agg("count").reset_index()
```

Displacement plot by SI

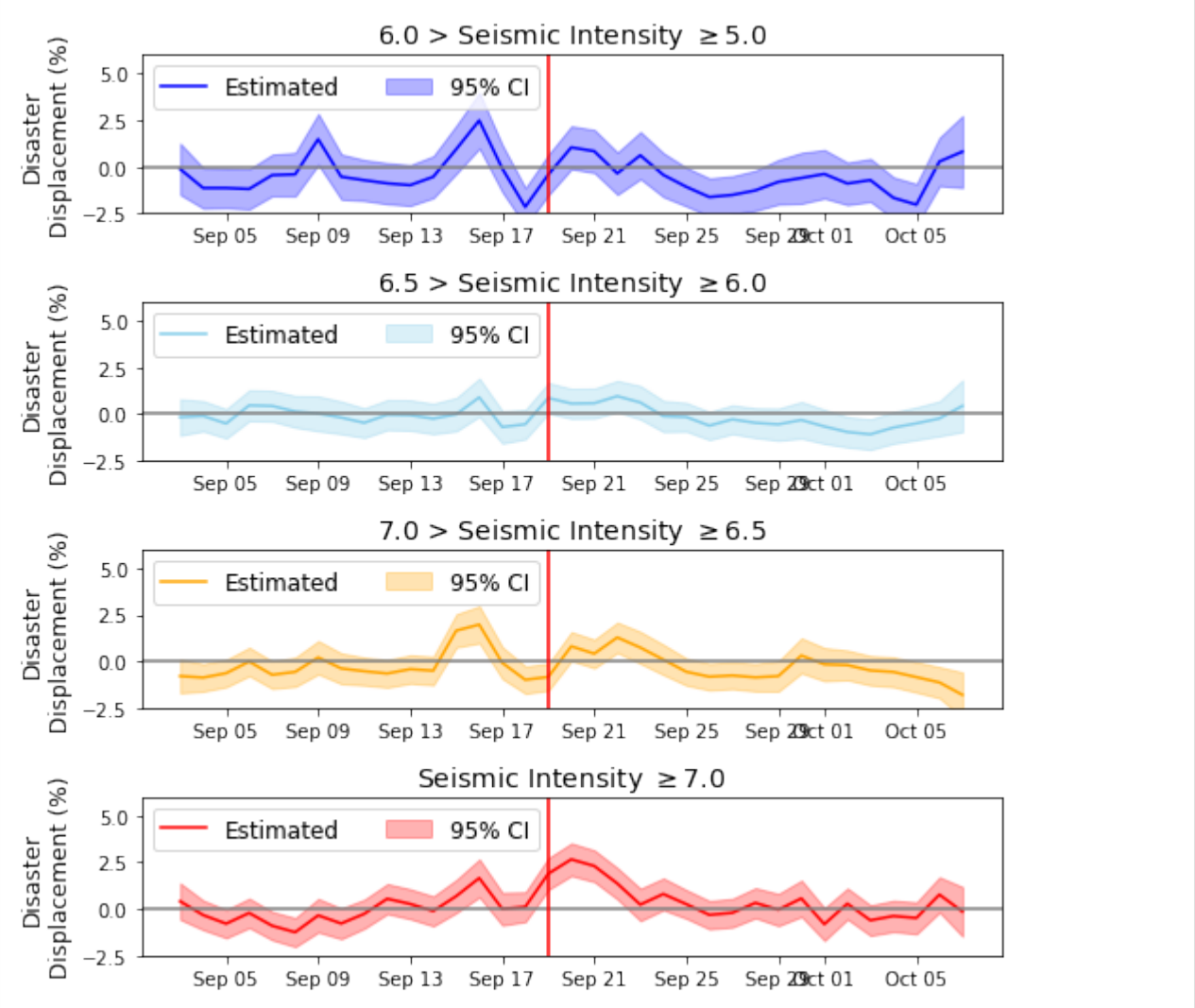
[84]:
```
sis = sorted(set(df_disp4["SI_cat"]))
sis = [5.0, 6.0, 6.5, 7.0]

cms = plt.get_cmap("jet",len(sis))

scale = "500m"

df_this = df_disp4[df_disp4["SI_cat"]==0]
date_disp = df_this.groupby('date').mean().reset_index()
date_disp["date_dt"] = date_disp["date"].values# apply(lambda x : dt.strptime(str(x), '%Y
↪%m%d'))
baseline = date_disp["500m"].values
```

[88]:
```
from scipy.optimize import minimize

def fit_baseline(data,baseline):

    def axb(p,x):
        return p[0]*x

    def errortot(data, baseline):
        tot = 0
        for i in np.arange(15):
            tot = tot + (baseline[i]-data[i])**2
        return tot

    x0 = np.array([1])
    res = minimize(lambda p: errortot(axb(p, data), baseline), x0=x0, method='Powell')

    return res.x
```

[137]:
```
def plotforSI(df_disp_se,si,ax,color, category, label, ylab, colname, baseline):
    df_this = df_disp_se[df_disp_se[category]==si]
    date_count = df_this.groupby('date').count().reset_index()[["date","uid"]]
    date_std = df_this.groupby('date').std().reset_index()[["date",colname]]
    date_std = date_std.rename(columns= {colname:"std"})
    date_disp = df_this.groupby('date').mean().reset_index()
    date_disp["date_dt"] = date_disp["date"].values # .apply(lambda x : dt.
↪strptime(str(x), '%Y%m%d'))
    date_disp["youbi"] = date_disp["date_dt"].apply(lambda x : x.weekday())
    date_disp = date_disp.merge(date_count, on="date")
    date_disp = date_disp.merge(date_std, on="date")

    data = date_disp[colname].values
    a = fit_baseline(data, baseline)
    print(a.shape, data.shape, baseline.shape)
    res = (a*data-baseline)*100
    ax.plot(date_disp["date_dt"],res, color=color, label="Estimated")
    date_disp["error"] = date_disp.apply(lambda x : 196*np.sqrt((x[colname]*(1-
↪x[colname]))/x["uid"]), \
                                         axis=1)

    ax.fill_between(date_disp["date_dt"],res-date_disp["error"].values, \
                    res+date_disp["error"].values,
                    color=color, alpha=0.3, label="95% CI")
    ax.xaxis.set_major_formatter(DateFormatter('%b %d'))
    ax.axhline(0, color="gray")
    # ax.set_xticks(["20170905","20170915","20170925","20171005"])
    ax.set_ylim(-2.5,6)
    ax.set_ylabel(ylab[0], fontsize=12)
    ax.axvline(datetime(2017,9,19), color="red")
    ax.legend(fontsize=12, ncol=5, loc="upper left")
    ax.set_title(label, fontsize=14)
```

[138]:
```
fig=plt.figure(figsize=(7.5,9))
gs=GridSpec(5,1)

res_si = {}

category = "SI_cat"
ylabs = ["Disaster\nDisplacement (%)", "$\Delta D$"]
colors = ["blue", "skyblue", "orange", "red"]
titles = ["6.0 > Seismic Intensity "+r"$\geq$"+"5.0",
          "6.5 > Seismic Intensity "+r"$\geq$"+"6.0",
          "7.0 > Seismic Intensity "+r"$\geq$"+"6.5",
          "Seismic Intensity "+r"$\geq$"+"7.0"]
for si,i in zip(sis,np.arange(len(sis))):
    ax = fig.add_subplot(gs[i,0])
    plotforSI(df_disp4, si, ax, colors[i], category, titles[i], ylabs, scale, baseline)

plt.tight_layout()
# plt.savefig("C:/users/yabec/desktop/displacement_si.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```

```
(1,) (35,) (35,)
(1,) (35,) (35,)
(1,) (35,) (35,)
(1,) (35,) (35,)
```



Displacement rate by wealth index

[6]:
```
df_disp_se_5 = df_disp4[df_disp4["SI_cat"]>=6.5]
```

[92]:
```
aaa = df_disp_se_5[df_disp_se_5["date"]==dt(2017,9,8)].groupby("index_pca")["500m"].
↪mean().reset_index()
bbb = df_disp_se_5[df_disp_se_5["date"]==dt(2017,9,8)].groupby("index_pca")["500m"].
↪count().reset_index()

aaa.shape, bbb.shape
```

[92]:
```
((106, 2), (106, 2))
```

[114]:
```
x = aaa["index_pca"].values
y = aaa["500m"].values
z = bbb["500m"].values
newz = []
```

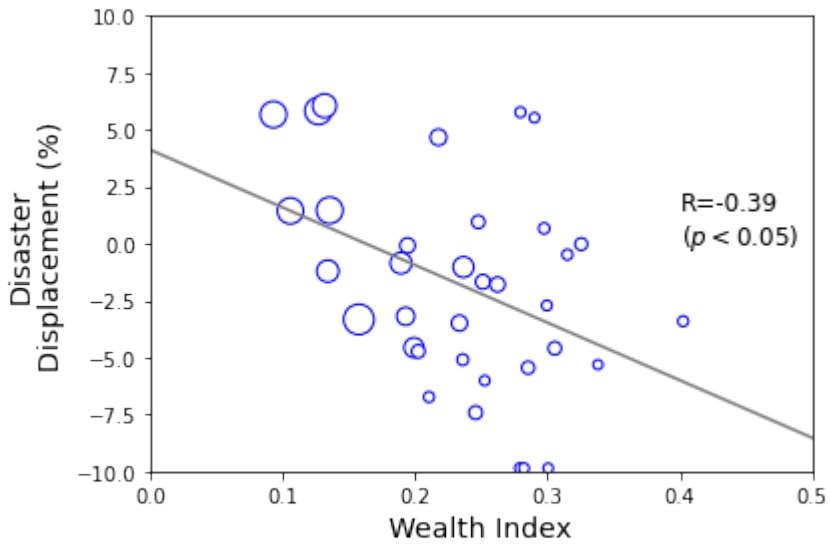
(continues on next page)

(continued from previous page)

```
xx = []
yy = []
for k,j,i in zip(x,y,z):
    if i > 20:
        xx.append(k)
        yy.append(j*100-baseline[15]*100)
        newz.append(np.sqrt(i)*5)
```

[115]:
```
plt.scatter(xx, yy, s=newz, edgecolor="b", facecolor="white")
c1, i1, s1, p_value, std_err = stats.linregress(xx,yy)
print(c1, i1, s1, p_value, std_err)
plt.plot([0,1],[i1,i1+c1], linestyle="-", color="gray")
plt.annotate("R="+str(s1)[:5]+"\n($p<0.05$)", xy=(0.4,0), fontsize=12)
plt.ylim(-10,10)
plt.xlim(0,0.5)
plt.xlabel("Wealth Index", fontsize=14)
plt.ylabel("Disaster\nDisplacement (%)", fontsize=14)

plt.tight_layout()
# plt.savefig("C:/users/yabec/desktop/wealth_disp.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```

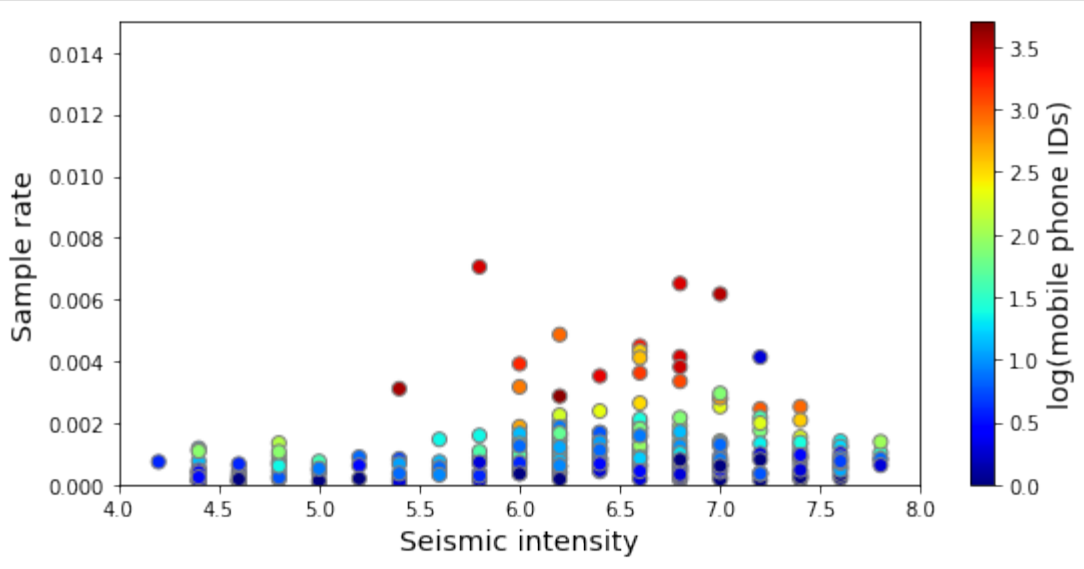
```
-25.29461166237592 4.122102681557249 -0.39968759624164824 0.01736848907416504 10.
↪098447012463001
```



Look at high damage + sample rate areas in detail

[116]:
```
### merge with disaster damage data (below)
muncode_rate["idlog"] = muncode_rate["uid"].apply(lambda x : np.log10(x))
```

[118]:
```
fig=plt.figure(figsize=(8,4))
ax = fig.add_subplot(1, 1, 1)
muncode_rate.plot.scatter("PARAMVALUE","rate",c='idlog',
                          colormap='jet', edgecolor="gray", s=50,ax=ax)
ax.set_xlim(4,8)
ax.set_ylim(0,0.015)
# ax.set_xticklabels([4,5,6,7,8])
ax.set_xlabel("Seismic intensity", fontsize=14)
ax.set_ylabel("Sample rate", fontsize=14)

f = plt.gcf()
cax = f.get_axes()[1]
cax.set_ylabel('log(mobile phone IDs)', fontsize=14)

plt.tight_layout()
# plt.savefig("C:/users/yabec/desktop/samplerate_si.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```



[4]:
```
target = muncode_rate[(muncode_rate["PARAMVALUE"]>=6.5) & (muncode_rate["uid"]>=30)]
```

[120]:
```
targetcodes = target["PCODE"].values
targetcodes = ['MX09015', 'MX09003', 'MX09010', 'MX09007', 'MX09014', 'MX09017']
```

[124]:
```
names = []
for t in targetcodes:
    name = adm2_shp[adm2_shp["ADM2_PCODE"]==t]["ADM2_ES"].values[0]
    SI = target[target["PCODE"]==t]["PARAMVALUE"].values[0]
    ids = target[target["PCODE"]==t]["uid"].values[0]
```

(continues on next page)

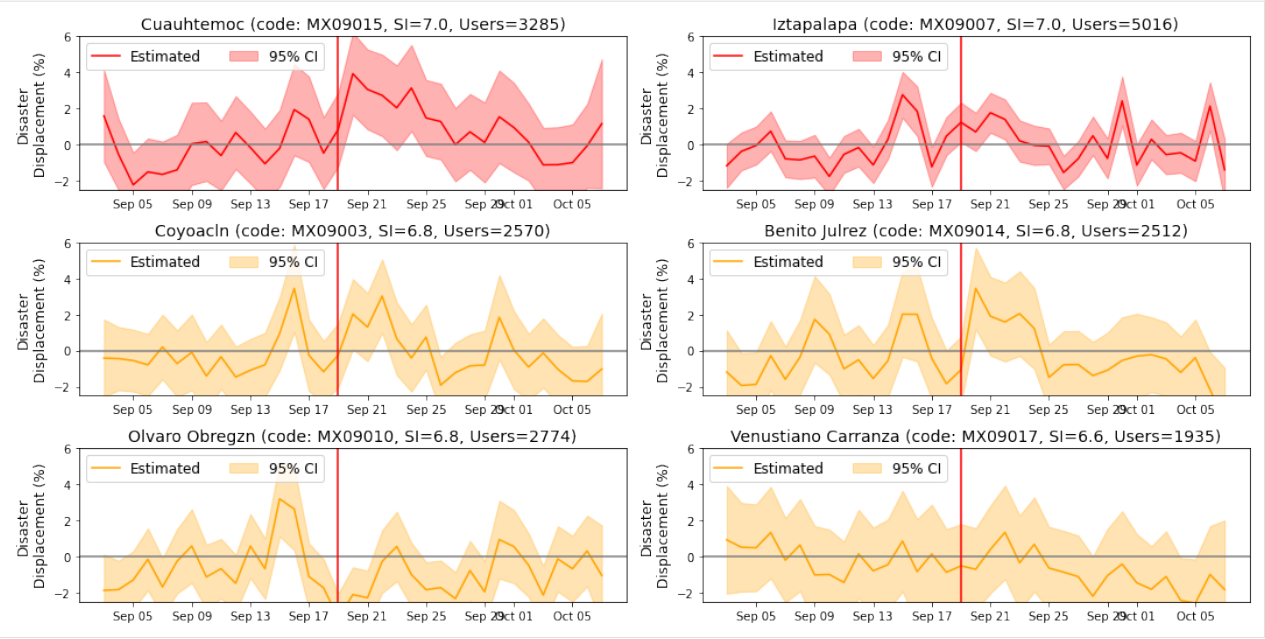
(continued from previous page)

```
    if "Cua" in name:
        name = "Cuauhtemoc"
    elif "lvaro" in name:
        name = "Olvaro Obregzn"
    names.append(name+" (code: "+t+", SI="+str(SI)+", Users="+str(ids)+")")

names
```

[124]:
```
['Cuauhtemoc (code: MX09015, SI=7.0, Users=3285)',
 'Coyoacln (code: MX09003, SI=6.8, Users=2570)',
 'Olvaro Obregzn (code: MX09010, SI=6.8, Users=2774)',
 'Iztapalapa (code: MX09007, SI=7.0, Users=5016)',
 'Benito Julrez (code: MX09014, SI=6.8, Users=2512)',
 'Venustiano Carranza (code: MX09017, SI=6.6, Users=1935)']
```

[139]:
```
fig=plt.figure(figsize=(15,2.5*3))
gs=GridSpec(3,2)

res_si = {}

category = "PCODE"
ylabs = ["Disaster\nDisplacement (%)", "$\Delta D$"]
colors = ["red", "orange", "orange", "red", "orange", "orange", "orange"]
titles = names

for i,pcode in enumerate(targetcodes):
    x,y = i, 0
    if i>2:
        y = 1
        x = i - 3
#     print(x,y)
    ax = fig.add_subplot(gs[x,y])
    plotforSI(df_disp4, pcode, ax, colors[i], category, titles[i], ylabs, scale,␣
↪baseline)

plt.tight_layout()
# plt.savefig("C:/users/yabec/desktop/displacement_places.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```

```
(1,) (35,) (35,)
(1,) (35,) (35,)
(1,) (35,) (35,)
(1,) (35,) (35,)
(1,) (35,) (35,)
(1,) (35,) (35,)
```



6.3.7 Zoom to Areas of interest

Cuahtlemoc

[180]:
```
date = dt(2017,9,20)
df_disp4_cua = df_disp4[(df_disp4["PCODE"]=="MX09015") & (df_disp4["date"]==date)].copy()
df_disp4_cua["distance"] = df_disp4_cua.apply(lambda row: np.log10(max(.001,
                        haversine([row["homelat_x"],row["homelon_x"]],
                                  [row["lat"],row["lng"]]))),axis=1)

# df_disp4_cua = df_disp4_cua[(df_disp4_cua["distance"]>2)]
df_disp4_cua = df_disp4_cua[(df_disp4_cua["distance"].between(1e-3, 1000))]

# df_disp4_cua = df_disp4_cua[((df_disp4_cua["distance"]>0.5) & (df_disp4_cua["distance"]
↪<1.5))]

# df_disp4_cua = df_disp4_cua[((df_disp4_cua["distance"]>0.5) & (df_disp4_cua["distance"]
↪<1.5)) | (df_disp4_cua["distance"]<-1)]

df_disp4_cua_gdf = gpd.GeoDataFrame(df_disp4_cua, geometry=gpd.points_from_xy(df_disp4_
↪cua.lng, df_disp4_cua.lat))
df_disp4_cua_gdf = df_disp4_cua_gdf[["uid","geometry"]]
```

[181]:
```
df_disp4_cua_gdf.columns
```

[181]:
```
Index(['uid', 'geometry'], dtype='object')
```

[182]:
```
df_disp4_cua_to = gpd.sjoin(df_disp4_cua_gdf,adm2_shp)
```

```
<ipython-input-182-bc7533da0982>:1: UserWarning: CRS mismatch between the CRS of left␣
↪geometries and the CRS of right geometries.
```

(continues on next page)

(continued from previous page)

```
Use `to_crs()` to reproject one of the input geometries to match the CRS of the other.

Left CRS: None
Right CRS: EPSG:4326

  df_disp4_cua_to = gpd.sjoin(df_disp4_cua_gdf,adm2_shp)
```

[183]:
```
targetcode_count = df_disp4_cua_to.groupby("ADM2_PCODE")["uid"].count().reset_index()
targetcode_count["idlog"] = targetcode_count["uid"].apply(lambda x : np.log10(x))
```

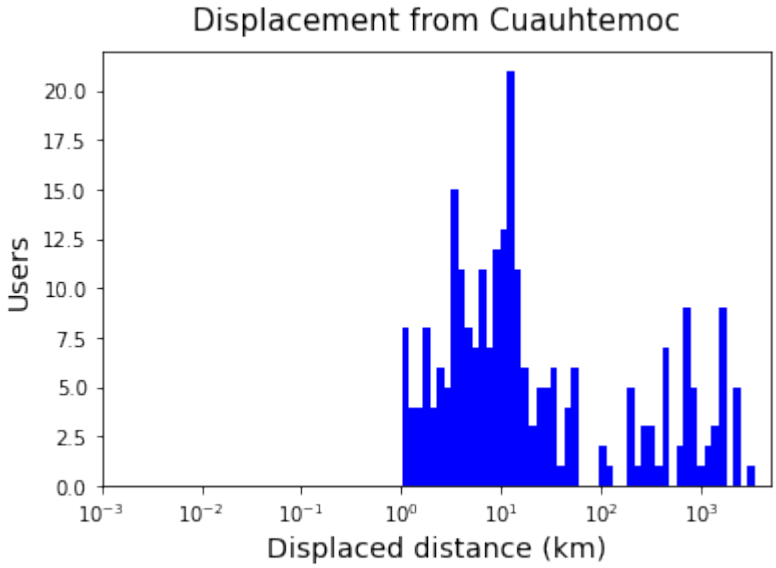
[184]:
```
mun_ids_pop_shp = adm2_shp.merge(targetcode_count, on="ADM2_PCODE", how="right")
mun_ids_pop_shp = mun_ids_pop_shp.to_crs(epsg=3857)
```

[185]:
```
fig,ax = plt.subplots(figsize=(20,10))
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.1)
mun_ids_pop_shp.plot(ax=ax, column='idlog', cmap='OrRd', legend=True,
                     cax=cax, legend_kwds={'label': "Users (log10)"}, alpha=.6)
ctx.add_basemap(ax, source=ctx.providers.Stamen.TonerLite)
# plt.savefig("C:/users/yabec/desktop/displacement_10km.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```

Distance of displacement

[186]:
```
df_disp4_cua["distance"] = df_disp4_cua.apply(lambda row: np.log10(haversine([row[
↪"homelat_x"],row["homelon_x"]],
                                                                              [row["lat"],
↪row["lng"]])),axis=1)
```

[187]:
```
fig,ax = plt.subplots(figsize=(6,4))

ax.hist(df_disp4_cua["distance"].values, bins=50, color="b")
ax.set_xticks([-3,-2,-1,0,1,2,3])
ax.set_xticklabels(["$10^{-3}$","$10^{-2}$","$10^{-1}$","$10^{0}$",
                    "$10^{1}$","$10^{2}$","$10^{3}$"])
ax.set_xlabel("Displaced distance (km)", fontsize=14)
ax.set_ylabel("Users", fontsize=14)
ax.set_title("Displacement from Cuauhtemoc", fontsize=15, pad=10)

# plt.savefig("C:/users/yabec/desktop/displacement_distance.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```



[ ]:

6.4 Mobility for resilience: POI visit rate analysis

This notebook shows how to transform raw mobility data to temporal profiles of POIs visit rate using `mobilkit`.

We start loading raw HFLB data using the `mobilkit.loader` module.

Then, we import a shapefile of POIs to follow their visit rates in time.

[1]:
```
%config Completer.use_jedi = False
%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
### import libraries
import numpy as np
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
from matplotlib.dates import DateFormatter
from scipy import stats
import pickle
import glob, os
from datetime import datetime as dt
from datetime import timedelta, timezone
from collections import Counter
from collections import OrderedDict
import pytz
import geopandas as gpd
import dask.dataframe as dd
import dask.array as da
from dask.distributed import Client, LocalCluster
from copy import copy
from scipy.stats import ttest_1samp
import contextily as ctx
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib.gridspec import GridSpec

import mobilkit
```

6.4.1 Data import

Connect to client and load the data.

[5]:
```
client = Client(address="127.0.0.1:8786", )
client.upload_file("dist/mobilkit-0.1-py3.9.egg")
```

[5]:
```
{'tcp://127.0.0.1:35587': {'status': 'OK'},
 'tcp://127.0.0.1:35817': {'status': 'OK'},
 'tcp://127.0.0.1:36843': {'status': 'OK'},
 'tcp://127.0.0.1:38985': {'status': 'OK'},
 'tcp://127.0.0.1:42163': {'status': 'OK'},
 'tcp://127.0.0.1:42947': {'status': 'OK'},
 'tcp://127.0.0.1:43653': {'status': 'OK'},
 'tcp://127.0.0.1:45023': {'status': 'OK'}}
```

[6]:
```
client
```

[6]:
```
<Client: 'tcp://192.168.178.34:8786' processes=8 threads=8, memory=32.00 GB>
```

[1]:
```
idhome = "../../data/id_home_3_1.csv"
df_idhome = pd.read_csv(idhome)
df_idhome["home"] = df_idhome["home"].apply(lambda v: [e for e in v.replace("[","")
```

(continues on next page)

(continued from previous page)

```
                                                    .replace("]","")
                                                    .split(" ")
                                                   if len(e)>0])
df_idhome["homelat"] = df_idhome["home"].apply(lambda v: float(v[1]))
df_idhome["homelon"] = df_idhome["home"].apply(lambda v: float(v[0]))
df_idhome = df_idhome[["uid","homelat","homelon"]].copy()
allids = set(df_idhome["uid"].values)
```

[9]:
```
df_idhome.shape
```

[9]:
```
(279541, 3)
```

POI locations

mobilkit uses KDTrees on a transformed projection EPSG 6362 (ideal for Mexico, change it accordingly in your case) that enables to compute distances in meters in the Mexican area using the euclidean distance.

[10]:
```
df_poi_loc = pd.read_csv("data/POI_shelters.csv")
df_poi_loc = df_poi_loc.rename(columns={"LATITUD": "poilat", "LONGITUD": "poilon"})
df_poi_loc["radius"] /= 1000.  # In KM!!!
df_poi_loc.head()
```

[10]:
```
      type                                             NOMBRE     poilat  \
0  shelter   Universidad del Valle de Mexico Campus San Angel  19.336474
1  shelter                                Venustiano Carranza  19.419172
2  shelter                                        Azcapotzalco  19.483777
3  shelter      Universidad del Valle de Mexico Sede Coyoacan  19.311669
4  shelter  Universidad Nacional Autonoma de Mexico Casa U...  19.422911

      poilon  radius
0 -99.192650     0.1
1 -99.112927     0.1
2 -99.184528     0.1
3 -99.138898     0.2
4 -99.160724     0.1
```

[11]:
```
df_poi_loc["type"].value_counts()
```

[11]:
```
shelter    31
Name: type, dtype: int64
```

Raw pings

[2]:
```
# Now I can quickly reload this first step of selection
alldataf = "../../results/displacement_selectedids_all_data"
filtered_dataf_reloaded = dd.read_parquet(alldataf).repartition(partition_size="50M")
if "datetime" not in filtered_dataf_reloaded.columns:
    # Add datetime column
    import pytz
    tz = pytz.timezone("America/Mexico_City")
```

(continues on next page)

(continued from previous page)

```
    # Filter on dates...
    filtered_dataf_reloaded = mobilkit.loader.filterStartStopDates(filtered_dataf_
↪reloaded,
                                                   start_date="2017-09-04",
                                                   stop_date="2017-10-08",
                                                   tz=tz,)
    filtered_dataf_reloaded = mobilkit.loader.compute_datetime_col(filtered_dataf_
↪reloaded, selected_tz=tz)
```

6.4.2 Compute POI visit figures

Estimate visit counts to POIs

[13]:
```
if False:
    # Compute
    tic = dt.now()
    joined_pings, poi_visit_results = mobilkit.spatial.compute_poi_visit(
                                                     df_pings=filtered_dataf_reloaded,
                                                     df_homes=df_idhome,
                                                     df_POIs=df_poi_loc,
                                                   from_crs="EPSG:4326", to_crs="EPSG:
↪6362",
                                                   min_home_dist_km=.2, visit_time_
↪bin="1H",
                                                   lat_lon_tol_box=.02)
    toc = dt.now()
    joined_pings.to_parquet("../../results/poi_pings_processed", partition_on="time_bin")
    poi_visit_results.to_pickle("../../results/poi_pings_results.pkl")
else:
    # Reload
    joined_pings = dd.read_parquet("../../results/poi_pings_processed")
    poi_visit_results - pd.read_pickle("../../results/poi_pings_results.pkl")
```

[15]:
```
tot_sec = (toc - tic).total_seconds()
tot_users_stats = joined_pings.groupby("uid").agg({"lat": "count"}).compute()
n_users = tot_users_stats.shape[0]
n_pings = tot_users_stats["lat"].sum()
print("Tot users:", n_users)
print("Tot pings:", n_pings)
print("Done in %d hours and %.01f minutes!" % (tot_sec//3600, (tot_sec % 3600)/60))
```

```
Tot users: 11802
Tot pings: 209458
Done in 0 hours and 28.4 minutes!
```

6.4.3 Plot visit temporal profiles

[18]:
```
df_poi = poi_visit_results.copy()
df_poi["date_dt"] = df_poi["time_bin"].copy()
df_poi["day"] = df_poi["date_dt"].dt.floor("1d")
df_poi["visits"] = df_poi["num_users"].copy()
df_poi["name"] = df_poi["NOMBRE"].copy()
```

Trend analysis

[19]:
```
def get_trend_compare(df_poi, how, what):
    df_shelter = df_poi[df_poi[how]==what]
    shlter_agg = df_shelter.groupby("date_dt").sum().reset_index()
    shlter_agg["youbi_time"] = shlter_agg["date_dt"].apply(lambda x : str(x.weekday())+"_
↪"+str(x.hour))
    shlter_agg_usual = shlter_agg# .iloc[:14*24]
    shlter_agg_usual["youbi_time"] = shlter_agg_usual["date_dt"].apply(lambda x : str(x.
↪weekday())+"_"+str(x.hour))
    shlter_agg_usual_avg = shlter_agg_usual.groupby('youbi_time').mean().reset_index()[[
↪"youbi_time","visits"]]
    shlter_agg_usual_sem = shlter_agg_usual.groupby('youbi_time').std().reset_index()[[
↪"youbi_time","visits"]]
    avg_trend = shlter_agg_usual_avg.rename(columns={"visits": "avgtrend"})
    sem_trend = shlter_agg_usual_sem.rename(columns={"visits": "semtrend"})
    avg = shlter_agg.merge(avg_trend, on="youbi_time")
    avg = avg.merge(sem_trend, on="youbi_time")
    avg = avg.sort_values(by=['date_dt'])
    return avg
```

[20]:
```
def plot_trend_compare(avg, how, what, showtrend, ax):
    if showtrend==True:
        trendmax = np.max(avg["avgtrend"])
        ax.plot(avg["date_dt"], avg["visits"]/trendmax, color="orange", label="data")
        ax.plot(avg["date_dt"], avg["avgtrend"]/trendmax, color="skyblue", label="trend")
        ax.fill_between(avg["date_dt"],
                        (avg["avgtrend"]-avg["semtrend"])/trendmax,
                        (avg["avgtrend"]+avg["semtrend"])/trendmax,
                        color="skyblue", alpha=.5, label="trend (95% CI)")
        ax.annotate("Puebla Eq.", xy=(dt(2017,9,19,18), .9),
                    fontsize=14, ha="left", color="red")
    else:
        ax.plot(avg["date_dt"], avg["visits"], color="orange", label="data")
        ax.annotate("Puebla Eq.", xy=(dt(2017,9,19,18), np.max(avg["visits"])*.9),
                    fontsize=14, ha="left", color="red")

    ax.xaxis.set_major_formatter(DateFormatter('%b %d'))
    # ax.set_xticks(["20170905","20170912","20170919","20170926","20171003","20171010"])
    # ax.set_xlim("20170904 12:00:00","20171011")
    ax.set_ylim(0)
    ax.set_ylabel("Visit Index", fontsize=14)
    ax.axvline(dt(2017,9,19,13), color="red")
```

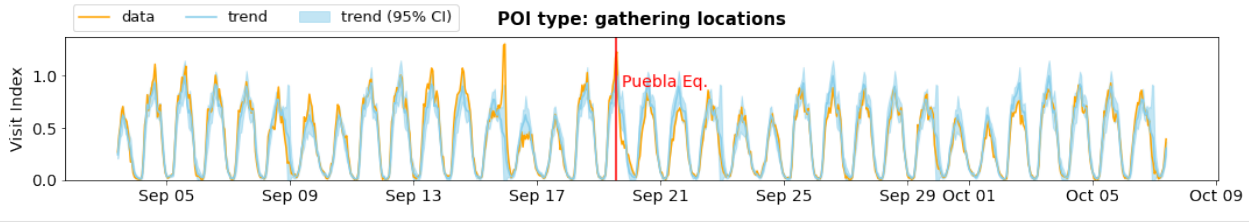
(continues on next page)

(continued from previous page)

```
    if what == "shelter":
        what = "gathering locations"
    if how=="type":
        ax.set_title("POI type: "+what, fontsize=15, pad=10, weight='bold')
    elif how=="name":
        ax.set_title("POI name: "+what, fontsize=15, pad=10, weight='bold')
    ax.tick_params(axis='x', labelsize=14)
    ax.tick_params(axis='y', labelsize=14)
    ax.legend(fontsize=13, ncol=3, loc="upper left", bbox_to_anchor=(0,1.28))
```

Aggregate trends for all POI types

[21]:
```
cats = ['airport', 'hospital', 'market', 'park', 'shelter']
cats = ["shelter",]
how = "type"

fig=plt.figure(figsize=(15,2.8*len(cats)))
gs=GridSpec(len(cats),1)

for i,c in enumerate(cats):
    ax = fig.add_subplot(gs[i,0])
    avg = get_trend_compare(df_poi, how, c)
    plot_trend_compare(avg, how, c, True, ax)

plt.tight_layout()
# plt.savefig("C:/users/yabec/desktop/visittrend_all.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```



Who came to shelter locations?

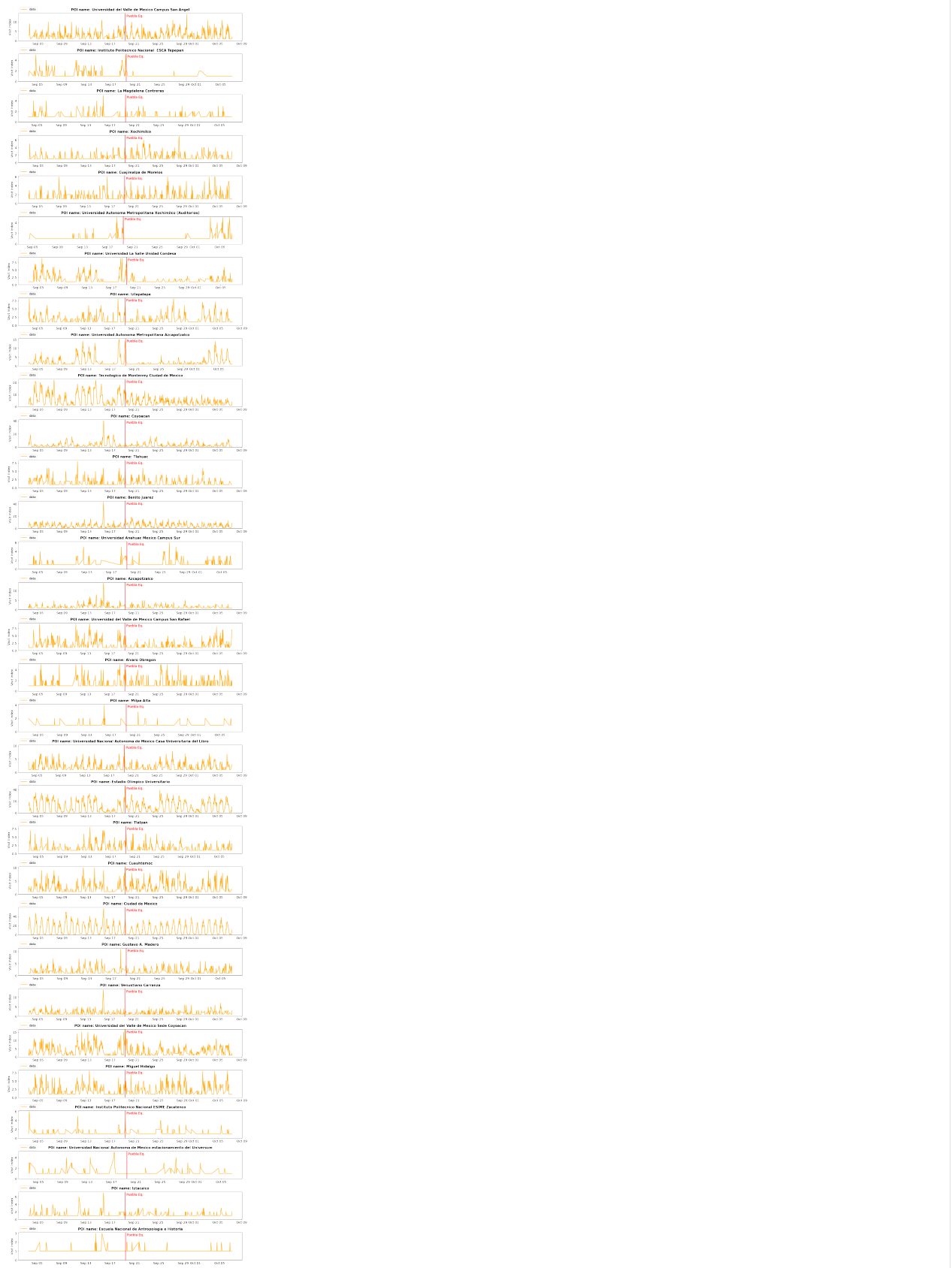
[24]:
```
shelters = set(df_poi[df_poi["type"]=="shelter"]["name"])
```

[26]:
```
cats = shelters
how = "name"

fig=plt.figure(figsize=(15,2.5*len(cats)))
gs=GridSpec(len(cats),1)

for i,c in enumerate(cats):
    ax = fig.add_subplot(gs[i,0])
```

(continues on next page)

(continued from previous page)

```
    avg = get_trend_compare(df_poi, how, c)
    plot_trend_compare(avg, how, c, False, ax)

plt.tight_layout()
# plt.savefig("C:/users/yabec/desktop/visittrend_all.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```


POI name: Universidad del Valle de Mexico Campus San Angel
POI name: Instituto Politecnico Nacional - ESCA Tepepan
POI name: La Magdalena Contreras
POI name: Xochimilco
POI name: Cuajimalpa de Morelos
POI name: Universidad Autonoma Metropolitana Xochimilco (Auditorios)
POI name: Universidad La Salle Unidad Cuidema
POI name: Iztapalapa
POI name: Universidad Autonoma Metropolitana Azcapotzalco
POI name: Tecnologico de Monterrey Ciudad de Mexico
POI name: Coyoacan
POI name: Tlahuac
POI name: Benito Juarez
POI name: Universidad Anahuac Mexico Campus Sur
POI name: Azcapotzalco
POI name: Universidad del Valle de Mexico Campus San Rafael
POI name: Alvaro Obregon
POI name: Milpa Alta
POI name: Universidad Nacional Autonoma de Mexico Casa Universitaria del Libro
POI name: Estadio Olimpico Universitario
POI name: Tlalpan
POI name: Cuauhtemoc
POI name: Ciudad de Mexico
POI name: Gustavo A. Madero
POI name: Venustiano Carranza
POI name: Universidad del Valle de Mexico Sede Coyoacan
POI name: Miguel Hidalgo
POI name: Instituto Politecnico Nacional ESIME Zacatenco
POI name: Universidad Nacional Autonoma de Mexico enfacinamiento del Universum
POI name: Iztacalco
POI name: Escuela Nacional de Antropologia e Historia

Shelter origin

[ ]:
```
shelterloc = df_poi.copy() # pd.read_csv("D:/WB_Mexico/spatialdata/POIdata_new/POI_
↪shelters.csv")
shelterloc["date"] = shelterloc["date_dt"].values
```

[ ]:
```
df_shel = shelterloc.copy()
df_shel["day"] = df_shel["date_dt"].dt.floor("1d")
```

[31]:
```
top = df_shel.groupby("name").sum().reset_index().sort_values("num_users",␣
↪ascending=False)["name"].values
top
```

[31]:
```
array(['Ciudad de Mexico', 'Estadio Olimpico Universitario',
       'Tecnologico de Monterrey Ciudad de Mexico', 'Benito Juarez',
       'Coyoacan', 'Universidad del Valle de Mexico Sede Coyoacan',
       'Universidad del Valle de Mexico Campus San Angel', 'Cuauhtemoc',
       'Universidad Autonoma Metropolitana Azcapotzalco',
       'Universidad Nacional Autonoma de Mexico Casa Universitaria del Libro',
       'Miguel Hidalgo',
       'Universidad del Valle de Mexico Campus San Rafael',
       'Venustiano Carranza', 'Iztapalapa', 'Gustavo A. Madero',
       'Tlalpan', 'Alvaro Obregon', 'Tlahuac',
       'Universidad La Salle Unidad Condesa', 'Azcapotzalco',
       'Cuajimalpa de Morelos', 'Xochimilco', 'Iztacalco',
       'Universidad Anahuac Mexico Campus Sur', 'La Magdalena Contreras',
       'Instituto Politecnico Nacional  ESCA Tepepan',
       'Instituto Politecnico Nacional ESIME Zacatenco',
       'Universidad Autonoma Metropolitana Xochimilco (Auditorios)',
       'Universidad Nacional Autonoma de Mexico estacionamiento del Universum',
       'Escuela Nacional de Antropologia e Historia', 'Milpa Alta'],
      dtype=object)
```

[34]:
```
id_home_feat = df_idhome.copy()
```

[35]:
```
from haversine import haversine
def getvisitinfo(tmp_shelter, date):
    befday = df_shel[df_shel["day"].isin(list(date))]
    befday_sub = befday[befday["name"]==tmp_shelter["name"]]
    allids = []
    for ids in befday_sub["users"]:
        for i in ids:
            if i!="":
                allids.append(i)
    allids_set = list(set(allids))
    ids_here = id_home_feat[id_home_feat["uid"].isin(allids_set)].copy()
    lat, lon = tmp_shelter["poilat"], tmp_shelter["poilon"]
    if ids_here.shape[0] > 0:
        ids_here = ids_here.assign(distance=ids_here.apply(lambda row: haversine([row[
↪"homelat"],row["homelon"]],
                                                                  [lat, lon]), axis=1))
    else:
```

(continues on next page)

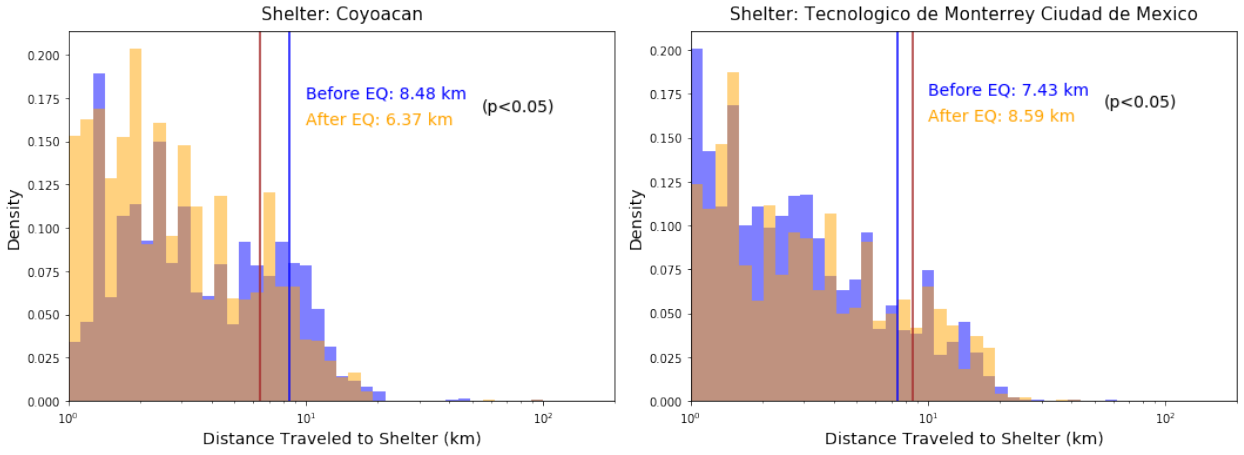
(continued from previous page)

```
        ids_here["distance"] = []
    return ids_here
```

[36]:
```
date1 = [dt.strptime(s, "%Y%m%d") for s in ["20170912","20170913","20170914",
                                            "20170915","20170916","20170917","20170918"]]
date2 = [dt.strptime(s, "%Y%m%d") for s in ["20170919","20170920","20170921",
                                            "20170922","20170923","20170924","20170925"]]

variable = "distance"
```

[258]:
```
fig=plt.figure(figsize=(15,5.5))
gs=GridSpec(1,2)

i=0
for shelter in set(befday["name"]):
    latlon = shelterloc[shelterloc["NOMBRE"]==shelter][["LATITUD","LONGITUD"]]
    lat = latlon["LATITUD"].values[0]
    lon = latlon["LONGITUD"].values[0]
    bef = getvisitinfo(shelter,date1)
    aft = getvisitinfo(shelter,date2)

    maxdist = np.max((np.max(bef[variable].values),np.max(aft[variable].values)))
    res = stats.ttest_ind(bef[variable].values, aft[variable].values, equal_var = False)
    if res[1]<0.05:
        ax = fig.add_subplot(gs[0,i])
#         fig,ax = plt.subplots(figsize=(7,4))
        ax.hist(bef[variable].values,bins=np.logspace(0, int(np.log10(maxdist)), 40),
                density=True,alpha=.5, color="b")
        ax.axvline(np.mean(bef[variable].values), color="b")
        ax.hist(aft[variable].values,bins=np.logspace(0, int(np.log10(maxdist)), 40),
                density=True,alpha=.5, color="orange")
        ax.axvline(np.mean(aft[variable].values), color="brown")
        ax.set_xlabel("Distance Traveled to Shelter (km)", fontsize=14)
        ax.set_xlim(1,200)
        ax.set_ylabel("Density", fontsize=14)
        ax.set_title("Shelter: "+shelter, fontsize=15, pad=10)
        ax.annotate("Before EQ: "+str(np.mean(bef[variable].values))[:4]+" km",
                    xy=(10,0.175), color="b", fontsize=14)
        ax.annotate("After EQ: "+str(np.mean(aft[variable].values))[:4]+" km",
                    xy=(10,0.16), color="orange", fontsize=14)
        ax.annotate("(p<0.05)",
            xy=(55,0.1675), color="k", fontsize=14)

#         print(,np.std(bef[variable].values))
#         print(np.mean(aft[variable].values),np.std(aft[variable].values))
#         print("--")
        ax.set_xscale("log")
        i+=1
plt.tight_layout()
# plt.savefig("C:/users/yabec/desktop/shelter_dist.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```

```
C:\Users\yabec\Anaconda3\lib\site-packages\ipykernel_launcher.py:11:␣
↪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: http://pandas.pydata.org/pandas-docs/stable/
↪indexing.html#indexing-view-versus-copy
  # This is added back by InteractiveShellApp.init_path()
```



By economic deprovation

[38]:
```
variable = "index_pca"

def getshelters(variable):
    df = pd.DataFrame()
    for shelter in set(befday["name"]):
        latlon = shelterloc[shelterloc["NOMBRE"]==shelter][["LATITUD","LONGITUD"]]
        lat = latlon["LATITUD"].values[0]
        lon = latlon["LONGITUD"].values[0]
        bef = getvisitinfo(shelter,date1)
        aft = getvisitinfo(shelter,date2)
        dist = np.mean(aft[variable].values)
        df = df.append({"shelter":shelter, "dist":dist, "lon":lon, "lat":lat}, ignore_
↪index=True)
    df_gdf = gpd.GeoDataFrame(df, geometry=gpd.points_from_xy(df.lon, df.lat),
                              crs= {"init": "epsg:4326"}).to_crs(epsg=3857)
    return df_gdf
```

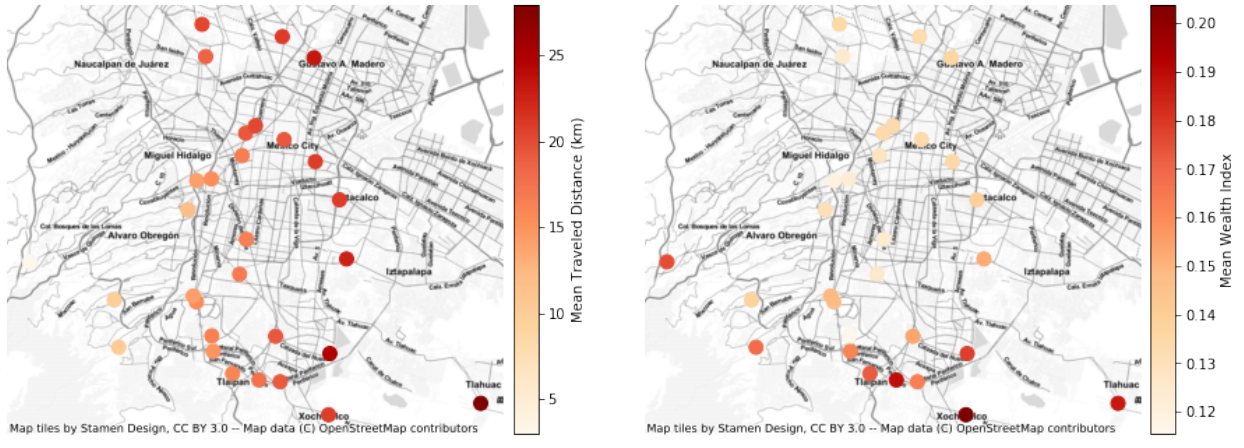
[302]:
```
fig=plt.figure(figsize=(15,8))
gs=GridSpec(1,2)

ax = fig.add_subplot(gs[0,0])
divider = make_axes_locatable(ax)
df_gdf = getshelters("distance")
cax = divider.append_axes("right", size="5%", pad=0.1)
df_gdf.plot(ax=ax, column='dist', cmap='OrRd', legend=True, markersize=100,
```

(continues on next page)

(continued from previous page)

```
                    cax=cax, legend_kwds={'label': "Mean Traveled Distance (km)"},␣
↪alpha=1)
ctx.add_basemap(ax, source=ctx.providers.Stamen.TonerLite)
ax.set_axis_off()

ax2 = fig.add_subplot(gs[0,1])
divider = make_axes_locatable(ax2)
df_gdf = getshelters("index_pca")
cax = divider.append_axes("right", size="5%", pad=0.1)
df_gdf.plot(ax=ax2, column='dist', cmap='OrRd', legend=True, markersize=100,
                    cax=cax, legend_kwds={'label': "Mean Wealth Index"}, alpha=1)
ctx.add_basemap(ax2, source=ctx.providers.Stamen.TonerLite)
ax2.set_axis_off()

# plt.savefig("C:/users/yabec/desktop/shelters_distance.png",
#             dpi=300, bbox_inches='tight', pad_inches=0.05)
plt.show()
```

```
C:\Users\yabec\Anaconda3\lib\site-packages\ipykernel_launcher.py:11:␣
↪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: http://pandas.pydata.org/pandas-docs/stable/
↪indexing.html#indexing-view-versus-copy
  # This is added back by InteractiveShellApp.init_path()
C:\Users\yabec\Anaconda3\lib\site-packages\ipykernel_launcher.py:11:␣
↪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: http://pandas.pydata.org/pandas-docs/stable/
↪indexing.html#indexing-view-versus-copy
  # This is added back by InteractiveShellApp.init_path()
```



[ ]:

[ ]:

6.5 Mobility for resilience: population density analysis

This notebook shows how to transform raw mobility data to population density maps using `mobilkit`.

We start loading raw HFLB data using the `mobilkit.loader` module.

[1]:
```
%config Completer.use_jedi = False
%matplotlib inline

import os
import csv
import gzip
import numpy as np
import pandas as pd
from math import sin, cos, sqrt, atan2, radians
import copy
import pickle
from datetime import datetime as dt
from datetime import timedelta
from datetime import timezone
import pytz
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
import matplotlib.colors as mcolors
import mpl_toolkits.axes_grid1
import matplotlib.gridspec as gridspec
import mpl_toolkits
import dask
from dask import dataframe as dd
from dask.distributed import Client

import mobilkit
```

6.5.1 Load raw pings for selected IDs

[5]:
```
# client.close()
client = Client(address="127.0.0.1:8786", )
client.upload_file("dist/mobilkit-0.1-py3.9.egg")
```

[5]:
```
{'tcp://127.0.0.1:44845': {'status': 'OK'},
 'tcp://127.0.0.1:45415': {'status': 'OK'},
 'tcp://127.0.0.1:46085': {'status': 'OK'},
 'tcp://127.0.0.1:46423': {'status': 'OK'}}
```

[6]:
```
client
```

[6]:
```
<Client: 'tcp://192.168.178.34:8786' processes=4 threads=4, memory=24.00 GB>
```

[1]:
```
alldataf = "results/displacement_selectedids_all_data/"
filtered_dataf_reloaded = dd.read_parquet(alldataf) # .repartition(partition_size="200M")
if "datetime" not in filtered_dataf_reloaded.columns:
    # Add datetime column
    import pytz
    tz = pytz.timezone("America/Mexico_City")
    # Filter on dates...
    filtered_dataf_reloaded = mobilkit.loader.filterStartStopDates(filtered_dataf_
↪reloaded,
                                                   start_date="2017-09-04",
                                                   stop_date="2017-09-25",
                                                   tz=tz,)
    filtered_dataf_reloaded = mobilkit.loader.compute_datetime_col(filtered_dataf_
↪reloaded, selected_tz=tz)
```

6.5.2 Compute day- and night-time locations

First I compute all the day and night time, then I use the function to compute the locations of each users for each day during day and night.

[2]:
```
# Prepare pings adding day- night- stuff
df_with_dn_time = mobilkit.temporal.filter_daynight_time(
                                              filtered_dataf_reloaded,
                                              filter_from_h=-1,
                                              filter_to_h=25, # To keep all the events
                                              previous_day_until_h=4,
                                              daytime_from_h=8,
                                              daytime_to_h=22,
                                          )
```

[9]:
```
# Filter to ROI
df_with_dn_time_roi = mobilkit.spatial.filter_to_box(df_with_dn_time,
                                                     minlon=-99.8761, maxlat=19.7661,
                                                     maxlon=-97.9208, minlat=18.7507)
```

[10]:
```
# Sample for speedup
df_sampled = df_with_dn_time_roi.sample(frac=.2)
```

[3]:
```
result_density = mobilkit.spatial.compute_population_density(df_sampled, maxpoints=50,␣
↪max_iter=100)
```

[13]:
```
result_density.to_pickle("results/population_density.pkl")
```

6.5.3 Plot density maps

[14]:
```
result_density = pd.read_pickle("results/population_density.pkl")
```

[15]:
```
normaldays = [dt.strptime(s, "%Y%m%d") for s in ["20170904","20170905","20170906",
                                                 "20170907","20170908","20170911",
                                                "20170912","20170913","20170914",
↪"20170915"]]
```

[16]:
```
# Prepare some annotations, just in case...
map_annotations = {
    "CDMX": dict(xy=(-98.85,19.5), xycoords='data', color="white", fontsize=14),
    "Puebla": dict(xy=(-98.5,18.8), xycoords='data', color="white", fontsize=14),
    "Toluca": dict(xy=(-99.75,19.5), xycoords='data', color="white", fontsize=14),
    "Cuernavaca": dict(xy=(-99.4,18.65), xycoords='data', color="white", fontsize=14),
}
```

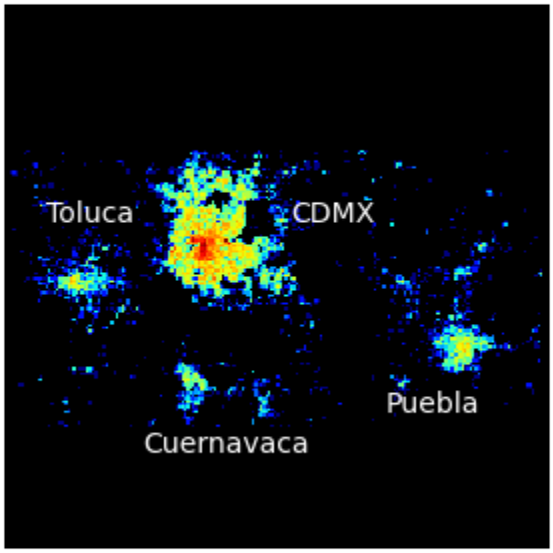
Daytime population density

[17]:
```
from importlib import reload
reload(mobilkit)
reload(mobilkit.viz)
reload(mobilkit.spatial)
```

[17]:
```
<module 'mobilkit.spatial' from '/home/ubi/Sandbox/mobilkit_dask/mobilkit/spatial.py'>
```

[18]:
```
df = result_density.reset_index().copy(deep=True)
```

[19]:
```
date = dt(2017,9,20)
daytime = True

df = df[(df["date"]==date) & (df["daytime"]==daytime)]

fig, ax = plt.subplots(1, 1, figsize=(5,5))
ax.set_aspect("equal")
heatmap, ax = mobilkit.viz.plot_density_map(df["lat"], df["lng"], (19.3, -98.9), 200, 1,␣
↪ax,
                                    annotations=map_annotations)
```



[20]:
```
# Select parameters for the analysis
bins = 200
center = (19.3, -98.9)
radius = 1
```

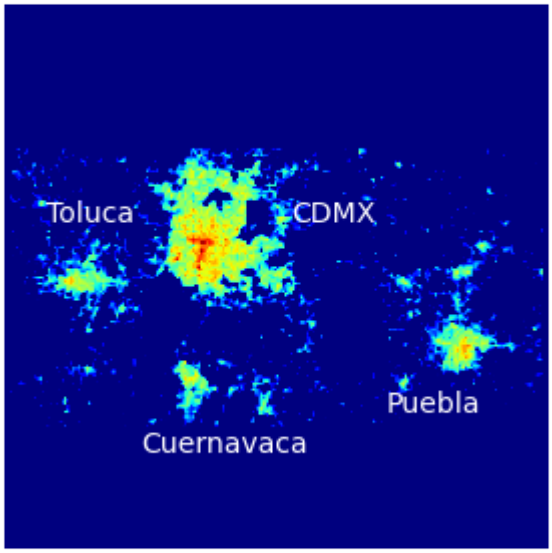
[21]:
```
results_day = mobilkit.spatial.stats_density_map(result_density, normaldays,␣
↪daytime=True,
                                                 bins=bins, center=center,␣
↪radius=radius)

results_night = mobilkit.spatial.stats_density_map(result_density, normaldays,␣
↪daytime=False,
                                                 bins=bins, center=center,␣
↪radius=radius)
```

Standard deviation

[22]:
```
mobilkit.viz.shori_density_map(results_day["std"], results_day["x_bins"], results_day["y_
↪bins"],
                               ax=None, annotations=map_annotations,
                               vmin=1e-4, vmax=results_day["std"].max())
```

```
/home/ubi/Sandbox/mobilkit_dask/mobilkit/viz.py:100: MatplotlibDeprecationWarning:␣
↪default base will change from np.e to 10 in 3.4.  To suppress this warning specify the␣
↪base keyword argument.
  divnorm = mcolors.SymLogNorm(linthresh=1.0, linscale=1.0, vmin=vmin, vmax=vmax)
```

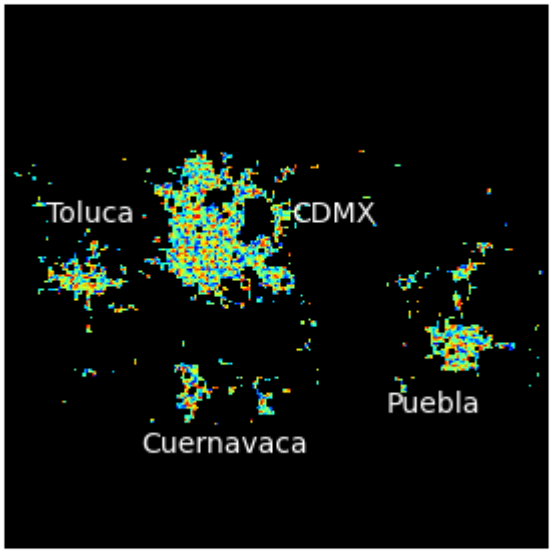
[22]:
```
<matplotlib.image.AxesImage at 0x7f1999c0a670>
```



z-scores

[23]:
```
# select the day to plot
mobilkit.viz.shori_density_map(results_day["zsc"][0], results_day["x_bins"], results_day[
↪"y_bins"],
                               ax=None, annotations=map_annotations,
                               vmin=-3, vmax=3)
```

[23]:
```
<matplotlib.image.AxesImage at 0x7f19559761c0>
```



Compute the zscore in a given day

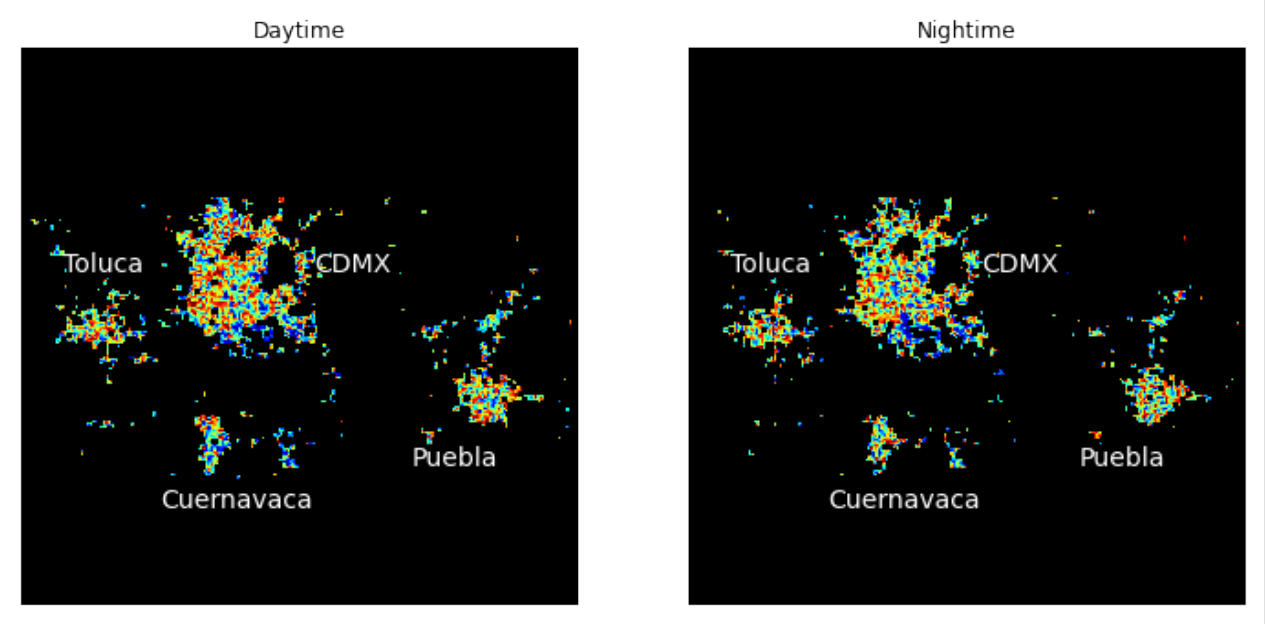
I just compute the density map for that day and compute its z-score w.r.t. the analysis on the "normal" days.

[24]:
```
date_selected = dt(2017, 9, 19)
density_earthquake_d, xybins = mobilkit.spatial.stack_density_map(result_density,␣
↪dates=[date_selected],
                                                                  center=center, bins=bins,␣
↪radius=radius)

density_earthquake_n, xybins = mobilkit.spatial.stack_density_map(result_density,␣
↪dates=[date_selected], daytime=False,
                                                                  center=center, bins=bins,␣
↪radius=radius)
```

[25]:
```
z_score_earthq_d = (density_earthquake_d - results_day["avg"]) / results_day["std"]
z_score_earthq_n = (density_earthquake_n - results_night["avg"]) / results_night["std"]
```

[26]:
```
fig, axs = plt.subplots(1,2,figsize=(12,6))

ax = axs[0]
ax.set_title("Daytime")
mobilkit.viz.shori_density_map(z_score_earthq_d[0], results_day["x_bins"], results_day[
↪"y_bins"],
                               ax=ax, annotations=map_annotations,
                               vmin=-3, vmax=3)


ax = axs[1]
ax.set_title("Nightime")
mobilkit.viz.shori_density_map(z_score_earthq_n[0], results_day["x_bins"], results_day[
↪"y_bins"],
                               ax=ax, annotations=map_annotations,
                               vmin=-3, vmax=3)
```

[26]:
```
<matplotlib.image.AxesImage at 0x7f1976bbd280>
```



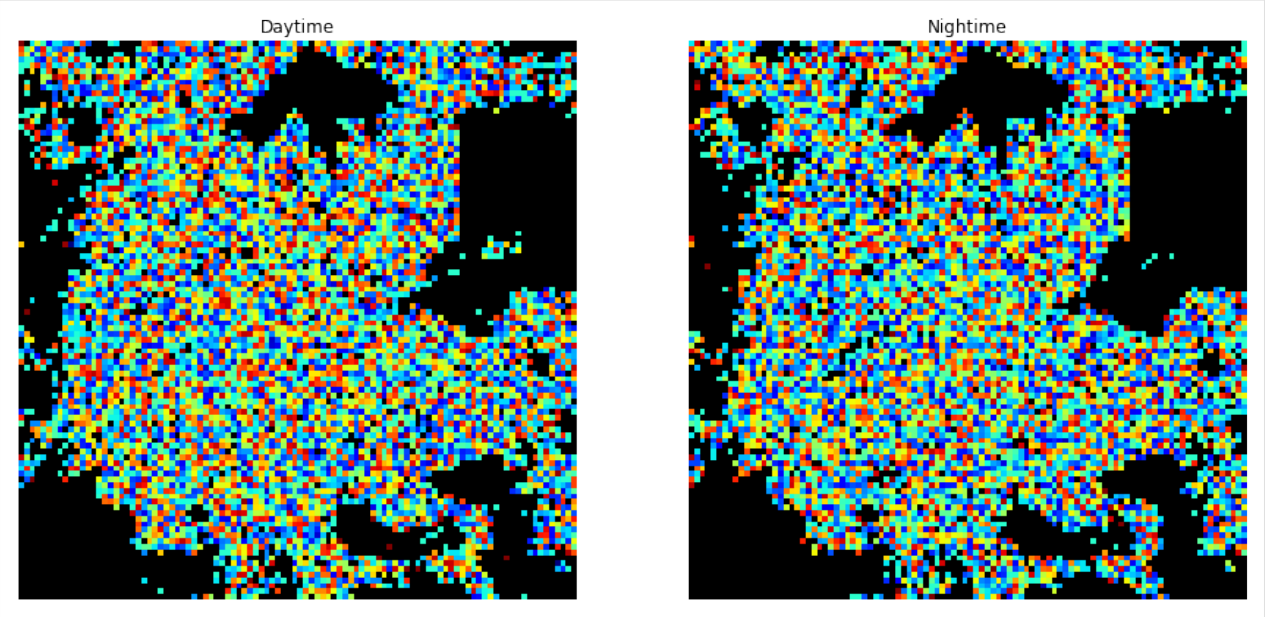
Zoom in CDMX

[27]:
```
center = (19.43, -99.13)
radius = .2
bins = int(500*radius)
```

[28]:
```
results_day_mxc = mobilkit.spatial.stats_density_map(result_density, normaldays,␣
↪daytime=True,
                                                     bins=bins, center=center,␣
↪radius=radius)

results_night_mxc = mobilkit.spatial.stats_density_map(result_density, normaldays,␣
↪daytime=False,
                                                     bins=bins, center=center,␣
↪radius=radius)
```

[29]:
```
fig = plt.figure(figsize=(15,10))
gs = gridspec.GridSpec(1,2)
ax = fig.add_subplot(gs[0,0])
ax.set_title("Daytime")
mobilkit.viz.shori_density_map(results_day_mxc["zsc"][0], results_day_mxc["x_bins"],␣
↪results_day_mxc["y_bins"],
                               ax=ax, vmin=-3, vmax=+3)

ax = fig.add_subplot(gs[0,1])
ax.set_title("Nightime")
mobilkit.viz.shori_density_map(results_night_mxc["zsc"][0], results_night_mxc["x_bins"],␣
↪results_night_mxc["y_bins"],
                               ax=ax, vmin=-3, vmax=+3)
```

[29]:
```
<matplotlib.image.AxesImage at 0x7f197c50ce80>
```



Compare day and night

[30]:
```
# Compare event day
date_selected = dt(2017, 9, 19)
density_earthquake_mxc_d, xybins_mxc_d = mobilkit.spatial.stack_density_map(result_
↪density, dates=[date_selected],
                                                        center=center, bins=bins,␣
↪radius=radius)

density_earthquake_mxc_n, xybins_mxc_n = mobilkit.spatial.stack_density_map(result_
↪density, dates=[date_selected],
                                                                                ␣
↪daytime=False,
                                                                                ␣
↪center=center, bins=bins, radius=radius)
```

[31]:
```
z_score_earthq_mxc_d = (density_earthquake_mxc_d - results_day_mxc["avg"]) / results_day_
↪mxc["std"]
z_score_earthq_mxc_n = (density_earthquake_mxc_n - results_night_mxc["avg"]) / results_
↪night_mxc["std"]
```

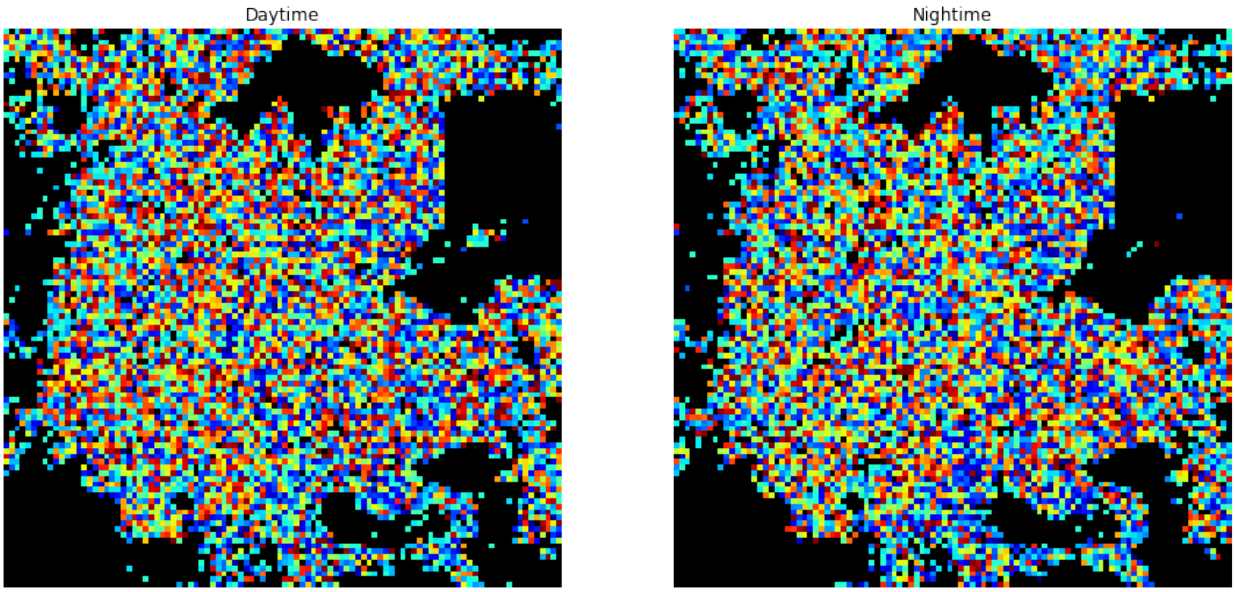
[32]:
```
fig = plt.figure(figsize=(15,10))
gs = gridspec.GridSpec(1,2)
ax = fig.add_subplot(gs[0,0])
ax.set_title("Daytime")
mobilkit.viz.shori_density_map(z_score_earthq_mxc_d[0], results_day_mxc["x_bins"],␣
↪results_day_mxc["y_bins"],
                               ax=ax, vmin=-3, vmax=3)

ax = fig.add_subplot(gs[0,1])
```

(continues on next page)

(continued from previous page)

```
ax.set_title("Nightime")
mobilkit.viz.shori_density_map(z_score_earthq_mxc_n[0], results_night_mxc["x_bins"],␣
↪results_night_mxc["y_bins"],
                               ax=ax, vmin=-3, vmax=3)
```

[32]:
```
<matplotlib.image.AxesImage at 0x7f1999cf0ca0>
```



Zoom in Puebla

[33]:
```
center = (19.03, -98.2)
radius = .2
bins = int(500*radius)
```

[34]:
```
results_day_puebla = mobilkit.spatial.stats_density_map(result_density, normaldays,␣
↪daytime=True,
                                                         bins=bins, center=center,␣
↪radius=radius)

results_night_puebla = mobilkit.spatial.stats_density_map(result_density, normaldays,␣
↪daytime=False,
                                                         bins=bins, center=center,␣
↪radius=radius)
```

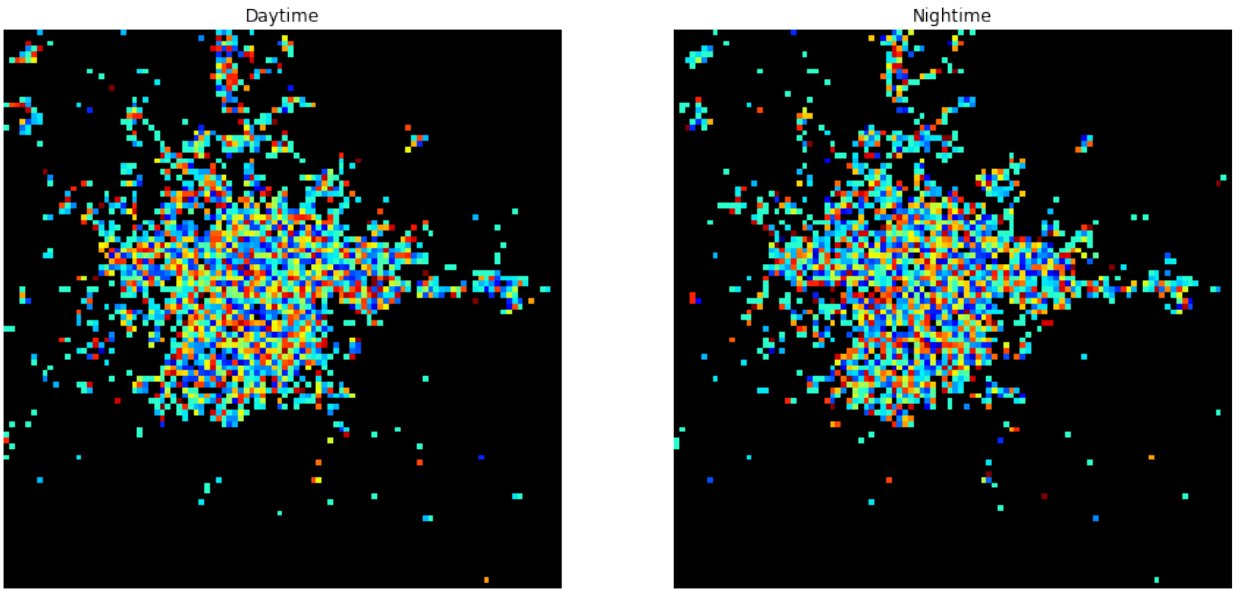
[35]:
```
fig = plt.figure(figsize=(15,10))
gs = gridspec.GridSpec(1,2)
ax = fig.add_subplot(gs[0,0])
ax.set_title("Daytime")
mobilkit.viz.shori_density_map(results_day_puebla["zsc"][0],
                               results_day_puebla["x_bins"], results_day_puebla["y_bins
↪"],
                               ax=ax, vmin=-3, vmax=+3)
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot(gs[0,1])
ax.set_title("Nightime")
mobilkit.viz.shori_density_map(results_night_puebla["zsc"][0],
                               results_night_puebla["x_bins"], results_night_puebla["y_
↪bins"],
                               ax=ax, vmin=-3, vmax=+3)
```

[35]:
```
<matplotlib.image.AxesImage at 0x7f19554309d0>
```



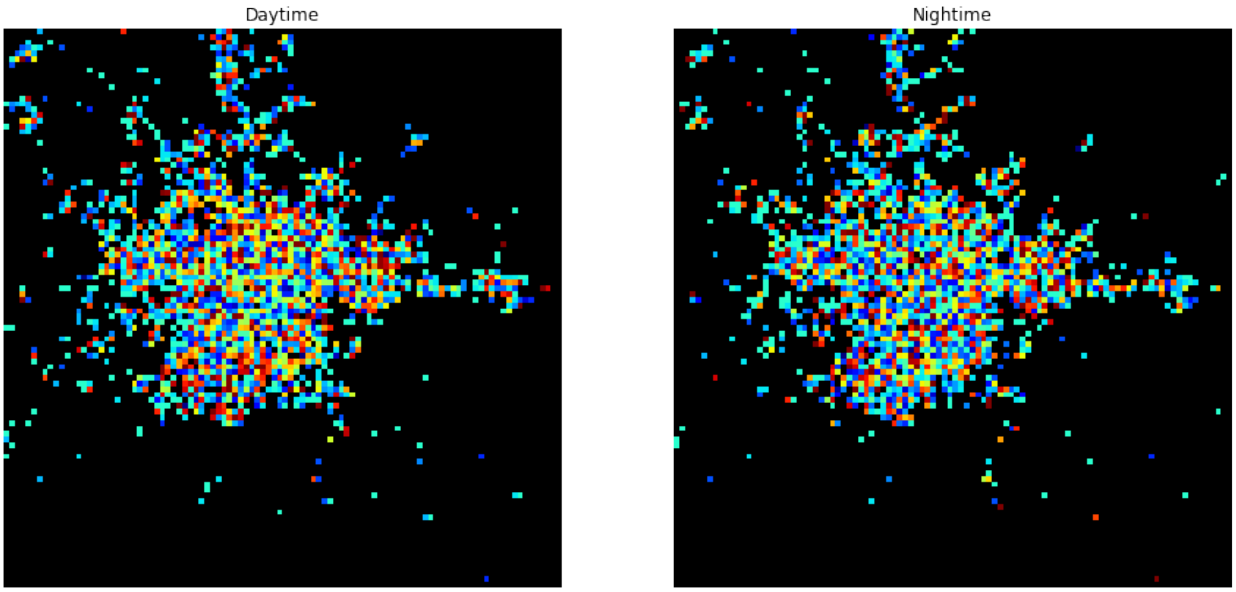
Compare day and night

[42]:
```
# Compare event day
date_selected = dt(2017, 9, 19)
density_earthquake_puebla_d, xybins_puebla_d = mobilkit.spatial.stack_density_map(result_
↪density, dates=[date_selected],
                                                                  center=center, bins=bins,␣
↪radius=radius)

density_earthquake_puebla_n, xybins_puebla_n = mobilkit.spatial.stack_density_map(result_
↪density, dates=[date_selected],
                                                                                        ␣
↪daytime=False,
                                                                                        ␣
↪center=center, bins=bins, radius=radius)
```

[43]:
```
z_score_earthq_puebla_d = (density_earthquake_puebla_d - results_day_puebla["avg"]) /␣
↪results_day_puebla["std"]
z_score_earthq_puebla_n = (density_earthquake_puebla_n - results_night_puebla["avg"]) /␣
↪results_night_puebla["std"]
```

[44]:
```
fig = plt.figure(figsize=(15,10))
gs = gridspec.GridSpec(1,2)
ax = fig.add_subplot(gs[0,0])
ax.set_title("Daytime")
mobilkit.viz.shori_density_map(z_score_earthq_puebla_d[0], results_day_puebla["x_bins"],␣
↪results_day_puebla["y_bins"],
                               ax=ax, vmin=-3, vmax=3)

ax = fig.add_subplot(gs[0,1])
ax.set_title("Nightime")
mobilkit.viz.shori_density_map(z_score_earthq_puebla_n[0], results_night_puebla["x_bins
↪"], results_night_puebla["y_bins"],
                               ax=ax, vmin=-3, vmax=3)
```

[44]:
```
<matplotlib.image.AxesImage at 0x7f195591ce20>
```



[ ]:

6.6 Urban spatial structure: the Mumbai example

In this notebook we show how to extract information on the daily commuting patterns of people starting from raw location based mobility data.

Specifically we start by loading raw data, we then filter them and observe the main features of the data (from both a collective and user based perspective).

We conduct all of our anlyses tessellating our area of interest with an hexagonal grid (with a side of about 450m).

For each grid cell we then infer some key observables linked to the users' mobility patterns, that is: - its land use (i.e., the predominant vocation); - the number of residents (workers) whose home (workplace) falls within the cell; - the average distance, time and speed of commuting for people residing there.

We will save the checkpoints of the procedure to speed up following refinements of the procedure.

We start with some generic imports to facilitate our analyses.

Note on data

The data used in this notebook have been provided by Quadrant within the **Resilient Urban Planning Analysis Using Smartphone Location Data** project of the The World Bank / Global Facility for Disaster Reduction and Recovery (GFDRR) - contract number 7204724.

Quadrant (An Appen Company) is a global leader in mobile location data, POI data, and corresponding compliance services. Quadrant provides anonymised location data and location-based business solutions that are fit for purpose, authentic, easy to use, and simple to organise. We offer data for almost all countries in the world, with hundreds of millions of unique devices and tens of billions of events per month, allowing our clients to perform location analyses, derive location-based intelligence, and make well-informed business decisions. Our data is gathered directly from first party opt-in mobile devices through a server-to-server integration with trusted publisher partners, delivering genuine and reliable raw GPS data unlike other location data sources relying heavily on Bidstream. Our consent management platform, QCMP, ensures that our data is compliant with applicable consent and opt-out provisions of data privacy laws governing the collection and use of location data.

[1]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import geopandas as gpd
import matplotlib.pyplot as plt
from datetime import datetime
import sys
import os
from shutil import rmtree
from dask.distributed import Client
from dask import dataframe as dd
import dask
import seaborn as sns
import statsmodels.api as sm
import pytz
import psutil
import multiprocessing
sns.set_context('notebook', font_scale=1.5)

dask.__version__
```

[1]:
```
'2022.04.1'
```

We also import `mobilkit` as `mk` to gain access to all the librarie's capabilities. We import some constant names of columns from the `dask_schemas` submodule as they will be useful later on.

[2]:
```
import mobilkit as mk
from mobilkit.spatial import haversine_pairwise
from mobilkit.dask_schemas import (latColName, lonColName,
                                   uidColName, zidColName,
                                   accColName, utcColName,
                                   dttColName, durColName,
                                   locColName, ldtColName,
                                   medLatColName, medLonColName)
from mobilkit.viz import compareLinePlot
```

6.6.1 Configuration

Here we need to specifiy the details of our analysis, such as where the input files are found and the name to give to our analysis.

We also set some variables needed to localize the timestamps, compute home and work locations and correctly compute the distances in the local projection.

Analysis input

[3]:
```
# The name of the subfolder in the output folder where to put checkpoints and
# save results
city_name = 'mumbai'
# The timezone to use to pass from timestamp to datetime
timezone = 'Asia/Kolkata'

# We limit the analysis to this date range
start_date = datetime(2022,  2, 28)
end_date   = datetime(2022,  4,  1)

# The hours to count as home and work:
# We set the initial and final time in 24h format using a float notation
# For instance 17:45 is 17.75
homeHours=(22, 8)
workHours=(10,18)

# This is the grid file to be used to tessellate data
aoi_grid_file = 'data/AOI_bbox/aoi_Mumbai_GRID.geoJSON'
```

Spatial details

[4]:
```
# The local projection to us when buffering the home and offices (to compute
# their approximated density), together with the city center
local_EPSG = 24381
homeWorkBufferMeters = 500
center_of_city = np.array([18.928453,72.829736]).reshape(1,2)

# When dealing with a policentric city we may want to specify the different
# business districts of the city
multiple_cbd_latlon = np.array([
        [18.928453,72.829736], # Central Business District,
        [19.013895,72.837438], # Extended Business District,
        [19.067498,72.865451], # New Business District,
        [19.165406,72.858306], # Suburban Business District (west),
        [19.151227,72.954612], # Suburban Business District (east),
        [19.054251,73.027987], # Peripheral Business District (Navi Mumbai),
        [19.216505,72.977984], # Peripheral Business District (Thane),
        [19.255434,73.046656], # Bhilwara Town,
])

# Custom stop detection keyowrds that will be passed to mk.spatial.findStops
```

(continues on next page)

(continued from previous page)

```
# See the function's documentation for details.
custom_stay_locations_kwds = {
    'minutes_for_a_stop': 10.0,
    'spatial_radius_km': 0.2,
    'no_data_for_minutes': 60*10,
}
```

Running options

[5]:
```
# If True it will recompute all the checkpoints (if already present)
recomputeAll = False

# The raw files' pattern are and where to save the results/output.
RAW_FILES_PATTERN = '/mnt/data_two/mob_data/IN_mumbai/year=*/month=*/day=*/*_*_*_*_
↪bucket-*'
OUT_ROOT_FOLDER  = '/mnt/data/resilience_analyses'
```

OSRM backend

Here, if we also want to compute "theoretical" commuting times and distances, we set the details of the Open Source Routing Machine.

Thanks to its dockerized implementation, it is pretty straightforward to set up a local OSRM server by following the instructions of the OSRM Backend.

You start by downloading the OpenStreetMaps databse for your area of interest from GeoFabrik.

```
wget http://download.geofabrik.de/europe/germany/berlin-latest.osm.pbf
```

Then, with the `osrm-cli` tools installed, prepare the data:

```
osrm-extract -p your_profile berlin.osm.pbf
osrm-partition berlin.osrm
osrm-customize berlin.osrm
```

Finally, run a docker container to be queried for routing duration/distances.

```
sudo docker run -t -i -p 5000:5000 -v "${PWD}:/data" osrm/osrm-backend osrm-routed --
↪algorithm=MLD /data/berlin-latest.osrm
```

[6]:
```
# OSRM BACKEND CONFIG
max_trip_duration_s = 10800
# Compose the base query
SERVER = "http://localhost"
PORT = 5000
PROFILE = "car"

osrm_url = f"{SERVER}:{PORT}/table/v1/{PROFILE}/"
```

6.6.2 Start of computation

We start by creating the output folders and loading the needed shapefiles and datasets.

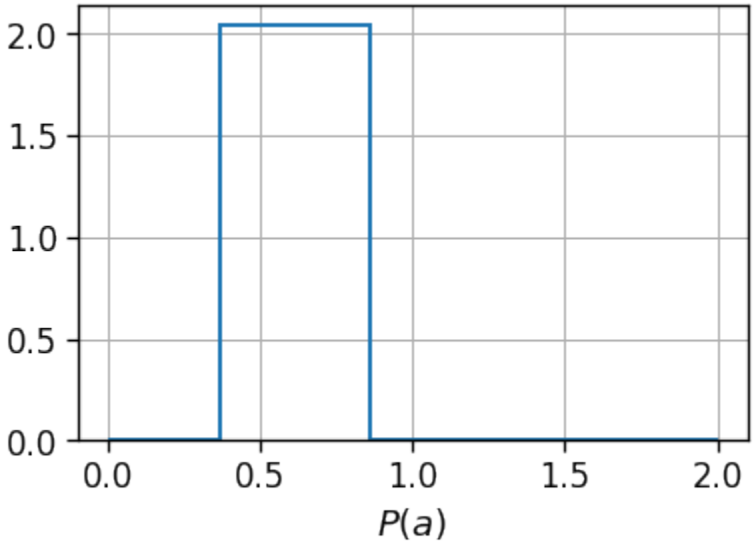
[7]:
```
# Create the output folders and path
OUT_DIR = os.path.join(OUT_ROOT_FOLDER, city_name)
os.makedirs(OUT_DIR, exist_ok=True)
out_grid_cells_file = os.path.join(OUT_DIR, 'gdf_landuse_rog.gpkg')
OUT_DIR_FIG = os.path.join(OUT_DIR, 'figures')
os.makedirs(OUT_DIR_FIG, exist_ok=True)
```

Load grid

Read the grid's geo-dataframe and project it to the local projection. We also plot the distribution of the area (in km^2) for the regular grid we selected to check it is regular.

[9]:
```
gdf_aoi_grid = gpd.read_file(aoi_grid_file)
gdf_aoi_grid_local = gdf_aoi_grid.to_crs(epsg=local_EPSG)

(gdf_aoi_grid_local.area/1e6).hist(bins=np.geomspace(.001, 2, 10),
                                   density=True,
                                   histtype='step',
                                   lw=2,
                                   label='Hex Grid'
                                  )
plt.xlabel(r'Cell area ($km^2$) - $a$')
plt.xlabel(r'$P(a)$')

print('Grig mean sqkm',
      (gdf_aoi_grid_local.area/1e6).mean(),
      'and median', (gdf_aoi_grid_local.area/1e6).median())
```

```
Grig mean sqkm 0.4871392896713538 and median 0.48713928967140174
```



Create *Dask* client

Here we create and connect to a dask client. We just specify where we want Dask to store tmp files (the `tmp_dask_dir`) and the memory limit per worker.

[8]:
```
tmp_dask_dir = '/mnt/dask_workplace/'
dask.config.set({'temporary_directory': tmp_dask_dir})
os.environ["DASK_TEMPORARY_DIRECTORY"] = tmp_dask_dir
```

[9]:
```
n_proc = int(multiprocessing.cpu_count() / 2) - 1
mem_per_proc = psutil.virtual_memory().total * 1.3 / 1e9 / n_proc
client = Client(
    memory_limit='%.01fG'%mem_per_proc,
    n_workers=n_proc,
    threads_per_worker=2
                )
client
```

```
/home/enrico/.local/share/virtualenvs/jupyter-EWzvnNR1/lib/python3.8/site-packages/
↪distributed/node.py:177: UserWarning: Port 8787 is already in use.
Perhaps you already have a cluster running?
Hosting the HTTP server on port 41527 instead
  warnings.warn(
2022-05-17 16:40:06,600 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-8g5lkup8', purging
2022-05-17 16:40:06,600 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-2e1e4vyf', purging
2022-05-17 16:40:06,601 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-yfmqukvb', purging
2022-05-17 16:40:06,601 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-7qz48yox', purging
2022-05-17 16:40:06,601 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-mzb427cg', purging
2022-05-17 16:40:06,601 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-47a8md3a', purging
2022-05-17 16:40:06,602 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-159b263l', purging
2022-05-17 16:40:06,602 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-stgtktpm', purging
2022-05-17 16:40:06,602 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-tx41os8t', purging
2022-05-17 16:40:06,602 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-gddlhakk', purging
2022-05-17 16:40:06,603 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-t048ojgk', purging
2022-05-17 16:40:06,603 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-m6q5h2w9', purging
```

[9]:
```
<Client: 'tcp://127.0.0.1:34233' processes=23 threads=46, memory=162.80 GiB>
```

Load raw data

We use the `dask.dataframe.read_parquet` to easily read the raw data.

Then we leverage on the `mk.loader.compute_datetime_col` to localize the timestamp to out selected timezone `timezone`.

We also compute a `day` column that will be used to partition the output files.

[10]:
```
df_raw = dd.read_parquet(RAW_FILES_PATTERN)
```

[11]:
```
df = df_raw.dropna(how='any',
                   subset=['timestamp','quad_id',
                           'latitude','longitude',
                           'horizontal_accuracy'])\
            .query('horizontal_accuracy<=70')

df[utcColName] = df['timestamp']/1000
df = mk.loader.compute_datetime_col(df, selected_tz=pytz.timezone(timezone))

df = df.rename(columns={
    'latitude': latColName,
    'longitude': lonColName,
    'quad_id': uidColName,
    'horizontal_accuracy': accColName
})

df = df[
    (df[dttColName] >= start_date)
    & (df[dttColName] < end_date)
][[dttColName, latColName, lonColName,
   uidColName, utcColName, accColName]].copy()

df['day'] = df['datetime'].dt.floor('1d').dt.strftime('%Y%m%d')
df.head(0)
```

[11]:
```
Empty DataFrame
Columns: [datetime, lat, lng, uid, UTC, acc, day]
Index: []
```

Filter date range

We filter the raw pings to be within the selected time range.

[12]:
```
df_pings = df[df[dttColName].between(start_date,end_date)].copy()
```

Computing users and pings count

To check the temporal dependence of the number of pings and/or users per day we can use the `mk.temporal.computeVolumeProfile`. The latter returns the daily volume of active pings/users seen in the dataset. To get a normalized (0-1) count simply pass `normalized=True` to the function.

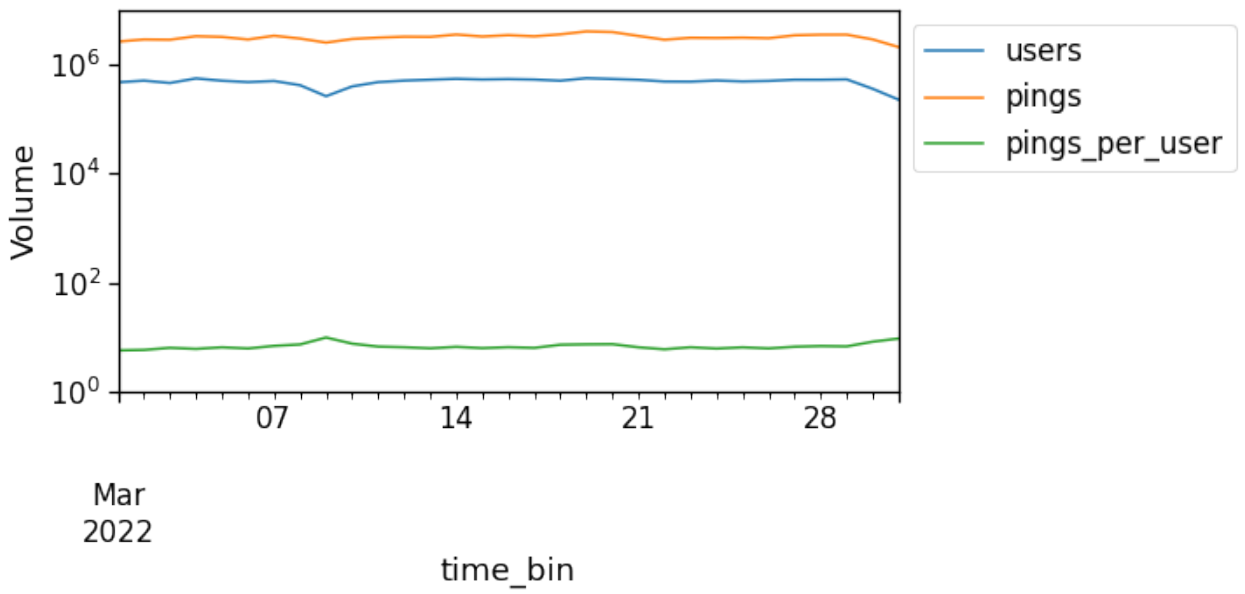
As one can see, the volume of daily users and pings is highly volatile, so that we have to apply normalization techniques in the following, for instance when computing land use.

[16]:
```
path_ping_user_volume
```

[16]:
```
'/mnt/data/resilience_analyses/mumbai/ping_users_volume.pkl'
```

[17]:
```
!rm /mnt/data/resilience_analyses/mumbai/ping_users_volume.pkl
```

[22]:
```
path_ping_user_volume = os.path.join(OUT_DIR, 'ping_users_volume.pkl')
if recomputeAll and os.path.exists(path_ping_user_volume):
    os.remove(path_ping_user_volume)
if not os.path.exists(path_ping_user_volume):
    df_volume = mk.temporal.computeVolumeProfile(df_pings, what='both', normalized=False)
    pd.to_pickle(df_volume, path_ping_user_volume)
fig, ax = plt.subplots(1,1,figsize=(8,4))
df_volume = pd.read_pickle(path_ping_user_volume)
df_volume.plot(ax=ax)
plt.semilogy()
plt.ylim(1e0, 1e7)
plt.ylabel('Volume')
plt.legend(loc=2, bbox_to_anchor=[1,1])
plt.savefig(os.path.join(OUT_DIR_FIG, 'user_ping_volume.pdf'))
```



Select valid users

Another important task is to check the representativeness of users. As we might have users that features too few pings (or that are seen only for one-two days), we want to focus the analysis on the users featuring a rich statistics.

To this end, we first compute the users' stats using `mk.stats.userStats`. The latter computes, for each user: - his number of raw pings; - the number of spanned days (i.e., the number of days between their first and last appearance of the user in the dataset); - the number of active days (i.e., the number of different days that the user has been active).

To visualize the impact of the threshold we apply on these counters, we can use the `mk.stats.plotUsersHist` function, that plots for us a nive 2d-histogram outlining the distribution of the users' number of pings, days, etc, plus the counters of users in the upper-left (ul) part of the histgram and so on.

[14]:
```
path_users_stats = os.path.join(OUT_DIR, 'users_stats.pkl')
if recomputeAll and os.path.exists(path_users_stats):
    os.remove(path_users_stats)
if not os.path.exists(path_users_stats):
    users_stats_df = mk.stats.userStats(df_pings).compute()
    pd.to_pickle(users_stats_df, path_users_stats)
users_stats_df = pd.read_pickle(path_users_stats)
users_stats_df.head(0)
```

[14]:
```
Empty DataFrame
Columns: [uid, min_day, max_day, pings, daysActive, daysSpanned, pingsPerDay, avg]
Index: []
```

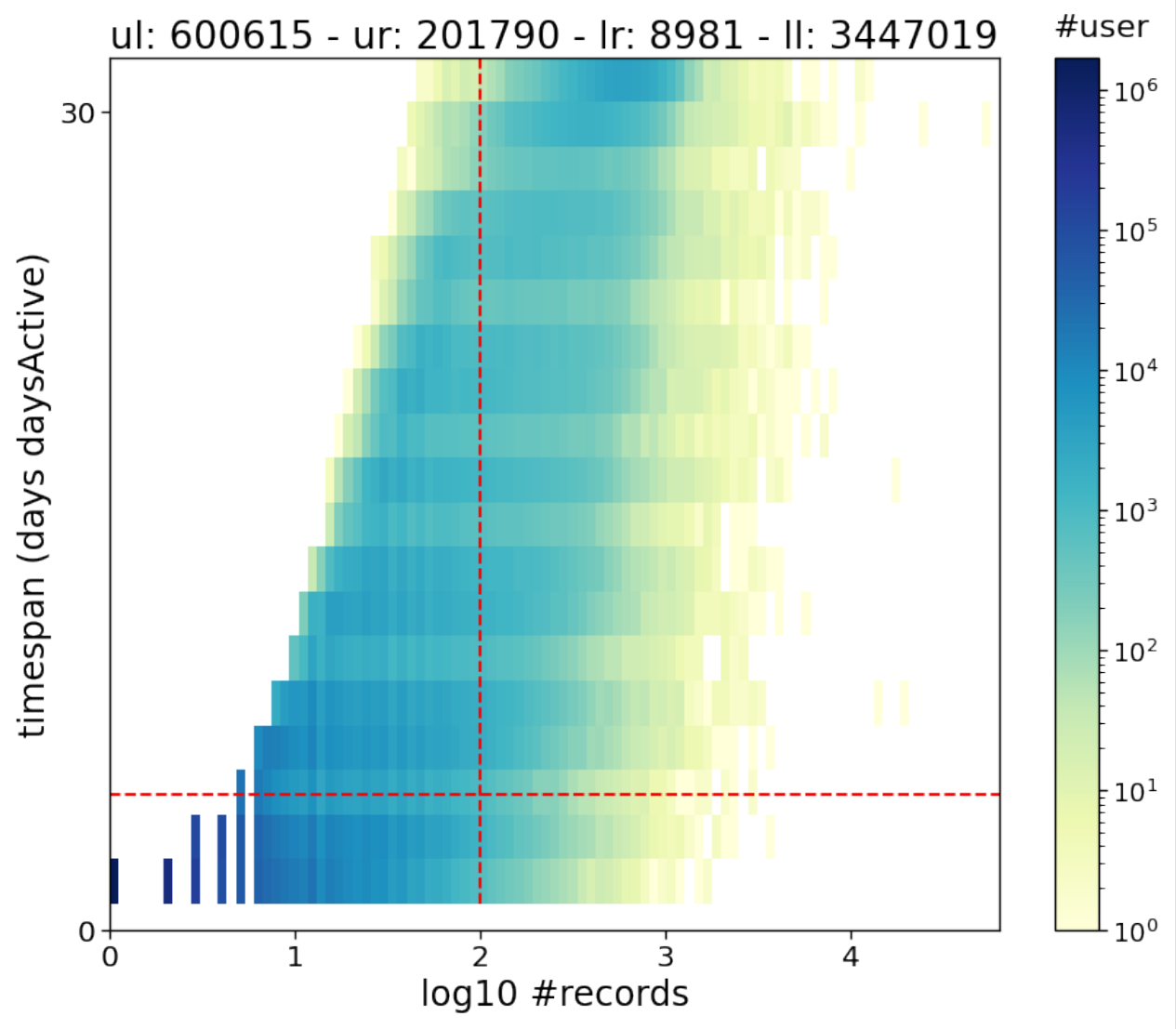
[15]:
```
# Select the threshold of minimum pings and active days for a user to be kept
min_user_pings = 100
min_user_days = 5
```

[16]:
```
mk.stats.plotUsersHist(users_stats_df,
                               min_pings=min_user_pings,
```

(continues on next page)

(continued from previous page)

```
                                    min_days=min_user_days)
plt.savefig(os.path.join(OUT_DIR_FIG, 'user_2dhist.pdf'))
```



[17]:
```
valid_users = list(users_stats_df.query("daysActive >= @min_user_days & pings > @min_
↪user_pings")["uid"])
df_pings_filtered = mk.stats.filterUsersFromSet(df_pings,
                                                users_set=valid_users)
```

6.6.3 Compute the needed tables

We will first compute all the event tables needed. These are: - the `stops_df_tessellated` where we transform raw pings in users' stops (i.e., stays of users long at least x minutes and localized in space); these stops also feature a duration, that is, an initial and final datetime. Also, each stop is assigned to a grid cell by looking at its mean point. To find the stops we leverage the traditional method of: > Hariharan, R. & Toyama, K. Project Lachesis: Parsing and Modeling Location Histories., > in Geographic Information Science 106–124 (Springer Berlin Heidelberg, 2004),

that is, we look for groups of pings close enough in space for at least a given duration to be labelled as stops. `mobilkit` leverages on the stop-detection implementation of `scikit-mobility` under the hood, conveniently parallelizing the per-user stop-detection work. - the `exploded_stops_df` contains the same information but each stop is replicated once for every hour that it touches. For instance, a stop starting at 09:12 and ending at 11:06 will be replicated at 09:12, 10:12, 11:12, so that we have an idea of the users and number of stops found in each area at each moment of the day. - then we compute the user locations, i.e., we group the stops close enough in meaningful locations. These will be encoded in the `users_stop_locs_df` that contains and additonal column `loc_ID` telling the location id of the stop and the `lat_medoid,lng_medoid` columns, containing the latitude and longitude of the medoid (i.e., the real stop point that minimizes the sum of distances to the other stops of the group). Again, in the default implementation we use the DBSCAN method as suggested in the *Project Lachesis*. An alternative solution is to use the `infostop` method as described in: > Aslak, U. & Alessandretti, L. Infostop: Scalable stop-location detection in multi-user mobility data. (2020). > arXiv 2003.14370. - the `user_locs_stats_hw_separated` will contain the home and work hours stats of each location, such as the number of hours that the user spent there, how many times we have seen him at the location for each single hour in the night and day hour ranges. Also, the fraction of day/night hours during which the location was the most visited (both in terms of stop duration and count) are reported. The procedure is as found in

> Lenormand, M., Louail, T., Barthelemy, M. & Ramasco, J. J. Is spatial information in ICT data reliable? > arXiv 1609.03375 (2016). - the `df_hw_locs_pd` table is a compressed, wide format table reporting, for each user, its home and work location stats and coordinates. It will be used in the following to compute the spatial and per-user statistics.

Compute stops and locations

The whole procedure to pass from users' pings to stops and locations is represented below (data from the Microsoft's GeoLife dataset).

A We start with the user pings color-coded with their original time (from blue, eldest, to yellow, most recent).

B Stops (coloured big circles) are found when some pings are close enough in space (within 200 meters one to the others in our case) for at least a minimum duration of time (10 minutes in this example). Also in this case the time sptamp of the stops is color-coded to show that the user is coming back multiple times in different moments in one venue.

C The stops falling close enough (within 100 meters one to the other in this example) are then linked and clusters of stops are found using the DBSCAN algorithm. These groups of stops represent the venues (or typical locations) of the user.

D Finally, each location is mapped to its centroid (the black-bordered dots), i.e., the points that minimizes the distance with respect to all the other points of the cluster of stops.

Compute stops

Here we start by computing the stops of the users and assigning them to a cell grid (tessellation).

We compute the stays using the `custom_stay_locations_kwds` argument.

```
custom_stay_locations_kwds = {
    'minutes_for_a_stop': 10.0,
    'spatial_radius_km': 0.2,
    'no_data_for_minutes': 600
}
```

In particular, we use the Project Lachesis method defined in > Hariharan, R. & Toyama, K. Project Lachesis: Parsing and Modeling Location Histories. > In Geographic Information Science 106–124 (Springer Berlin Heidelberg, 2004).

to extract the stops from the sequence of positions of the users. Specifically we define a stop when a user stays for at least 10 minutes in a radius of up to 200 meters. Moreover, we allow the data to have hole of up to 10 hours (meaning that if a user is seen at a location, no data are present during the next 10 hours max and then the user is still seen at the location we count the stop to be a valid one).

We also pass the reference to the shapefile to tessellate the stops location, i.e., assign each stop's centroid to a cell of the grid.

The resulting dataframe reports, for each stop, the latitude and longitude of the stop centroid, the start and end time of the stay and the user id plus the corresponding tile id.

[18]:
```
stops_df_tessellated = mk.spatial.findStops(df_pings_filtered,
                                            tesselation_shp=aoi_grid_file,
                                            filterAreas=True,
                                            stay_locations_kwds=custom_stay_locations_
↪kwds,
                                           )
stops_df_tessellated
```

```
/home/enrico/.local/share/virtualenvs/jupyter-EWzvnNR1/lib/python3.8/site-packages/
↪geopandas/array.py:275: ShapelyDeprecationWarning: The array interface is deprecated␣
↪and will no longer work in Shapely 2.0. Convert the '.coords' to a numpy array instead.
  return GeometryArray(vectorized.points_from_xy(x, y, z), crs=crs)
```

[18]:
```
Dask DataFrame Structure:
                     lat      lng     uid        datetime leaving_datetime duration tile_
↪ID
npartitions=310
                   float64  float64  object  datetime64[ns]   datetime64[ns]  float64  ␣
↪int64
                       ...      ...     ...             ...              ...      ...    ␣
↪...
...                    ...      ...     ...             ...              ...      ...    ␣
↪...
                       ...      ...     ...             ...              ...      ...    ␣
↪...
                       ...      ...     ...             ...              ...      ...    ␣
↪...
Dask Name: getitem, 27855 tasks
```

[19]:
```
# Code to save the dataframe to disk
path_stops_tessellated = os.path.join(OUT_DIR, 'dataHFLB_stops_tessellated_parquet')
if recomputeAll and os.path.exists(path_stops_tessellated):
    rmtree(path_stops_tessellated)
if not os.path.exists(path_stops_tessellated):
    stops_df_tessellated['day'] = stops_df_tessellated[dttColName].dt.floor('1d').dt.
↪strftime('%Y%m%d')
    stops_df_tessellated.to_parquet(path_stops_tessellated,
                                    compression='gzip',
                                    write_index=False,
                                    partition_on='day',
                                    overwrite=True)
stops_df_tessellated = dd.read_parquet(path_stops_tessellated)
```

Explode the stops

Some functions of `mobilkit` require a single ping (or stop) to be repeated multiple times at a given frequency, for instance to compute the visit profiles of areas.

Here we expand the stops falling in more than one hour to be repeated, i.e., the **exploded** stops. This means that each stop instead of reporting the start and end time is duplicated once for every given interval the stop touches (in this case every hour).

This view is useful if we want to quickly compute the number of users found in a given area at a given hour.

[20]:
```
exploded_stops_df = mk.spatial.expandStops(stops_df_tessellated,
                                           freq='1h',
                                           explode_stop=True)
exploded_stops_df
```

[20]:
```
Dask DataFrame Structure:
                     lat      lng     uid        datetime duration tile_ID              ␣
↪day   stops
npartitions=9610
                   float64  float64  object  datetime64[ns]  float64   int64 ␣
↪category[known]  object
                       ...      ...     ...             ...      ...     ...              ␣
↪...     ...
...                    ...      ...     ...             ...      ...     ...              ␣
↪...     ...
                       ...      ...     ...             ...      ...     ...              ␣
↪...     ...
                       ...      ...     ...             ...      ...     ...              ␣
↪...     ...
Dask Name: expand_stops_partition, 19220 tasks
```

[21]:
```
path_stops_exploded_tessellated = os.path.join(OUT_DIR, 'dataHFLB_stops_tessellated_
↪exploded_parquet')
if recomputeAll and os.path.exists(path_stops_exploded_tessellated):
    rmtree(path_stops_exploded_tessellated)
if not os.path.exists(path_stops_exploded_tessellated):
    exploded_stops_df['day'] = exploded_stops_df[dttColName].dt.floor('1d').dt.strftime('
↪%Y%m%d')
    exploded_stops_df.to_parquet(path_stops_exploded_tessellated,
                                 compression='gzip',
                                 write_index=False,
                                 partition_on='day',
                                 overwrite=True)
exploded_stops_df = dd.read_parquet(path_stops_exploded_tessellated)
exploded_stops_df = exploded_stops_df.repartition(npartitions=200)
exploded_stops_df.head(0)
```

[21]:
```
Empty DataFrame
Columns: [lat, lng, uid, datetime, duration, tile_ID, stops, day]
Index: []
```

Compute the users' locations

To detect the home and workplace of the users, we group their stops in meaningful locations.

Locations are clusters of stops (or stays) that are close enough to supposedly belong to one venue of the user. Home and work are the two most important ones but other locations can be found in a user's mobility diary.

`mobilkit` offers two ways to find locations: - using a `dbscan` unsupervised clustering algorithm, as in the Lachesis project; - using the `infostop` package, that groups the stops of a user by linking them in a weighted network. > Aslak, U. & Alessandretti, L. Infostop: Scalable stop-location detection in multi-user mobility data. arXiv 2003.14370 (2020).

In the following we use the `dbscan` approach, where we link stops distant up to 100 meters and using clusters with a n=1 core.

The result is a dataframe reporting the latitude and longitude of the medoid of the locations of each user, plus a unique identifier of each location-user couple.

[22]:
```
users_stop_locs_df = mk.spatial.computeUsersLocations(
                            stops_df_tessellated.repartition(npartitions=200),
                            method='dbscan',
                            link_dist=100,
                            min_stops_count=1,
                            return_locations=False)
users_stop_locs_df
```

[22]:
```
Dask DataFrame Structure:
                  index      lat      lng     uid        datetime leaving_datetime␣
↪duration tile_ID              day loc_ID lng_medoid lat_medoid
npartitions=200
                  int64  float64  float64  object  datetime64[ns]   datetime64[ns] ␣
↪float64   int64  category[known]  int64    float64    float64
                    ...      ...      ...     ...             ...              ...     ␣
↪...     ...              ...    ...        ...        ...
...                 ...      ...      ...     ...             ...              ...     ␣
↪...     ...              ...    ...        ...        ...
                    ...      ...      ...     ...             ...              ...     ␣
↪...     ...              ...    ...        ...        ...
                    ...      ...      ...     ...             ...              ...     ␣
↪...     ...              ...    ...        ...        ...
Dask Name: reset_index, 18685 tasks
```

[23]:
```
path_users_stop_locs = os.path.join(OUT_DIR, 'users_stop_locs_df')
if recomputeAll and os.path.exists(path_users_stop_locs):
    rmtree(path_users_stop_locs)

if not os.path.exists(path_users_stop_locs):
    users_stop_locs_df['day'] = users_stop_locs_df[dttColName].dt.floor('1d').dt.
↪strftime('%Y%m%d')
    users_stop_locs_df.to_parquet(path_users_stop_locs,
                                  compression='gzip',
                                  write_index=False,
                                  partition_on='day',
                                  overwrite=True)
users_stop_locs_df = dd.read_parquet(path_users_stop_locs).repartition(npartitions=200)
users_stop_locs_df.head(0)
```

[23]:
```
Empty DataFrame
Columns: [index, lat, lng, uid, datetime, leaving_datetime, duration, tile_ID, loc_ID,␣
↪lng_medoid, lat_medoid, day]
Index: []
```

Compute statistics on the users' locations

We compute the number of times a user visits each location, how long it stays there and for how many home- and work-time hours we observe him at the location.

Following *Lenormand, et al.*, we then check how the number of users with a valid home or work location changes when we raise the minimum statistical requirements (i.e., number of hours passed at home, number of active days, etc.).

We also compute the δ_h fraction of home- and work-hours for which the single location is the most visited one.

[24]:
```
user_locs_stats_hw_separated = mk.stats.stopsToHomeWorkStats(users_stop_locs_df,
                                                             force_different=True)
```

[25]:
```
path_user_locs_stats_hw_separated = os.path.join(OUT_DIR, 'user_locs_stats_hw_separated.
↪pkl')
if recomputeAll and os.path.exists(path_user_locs_stats_hw_separated):
    os.remove(path_user_locs_stats_hw_separated)
if not os.path.exists(path_user_locs_stats_hw_separated):
    user_locs_stats_hw_separated = user_locs_stats_hw_separated.compute()
    pd.to_pickle(user_locs_stats_hw_separated, path_user_locs_stats_hw_separated)
user_locs_stats_hw_separated = pd.read_pickle(path_user_locs_stats_hw_separated)
user_locs_stats_hw_separated.head(0)
```

[25]:
```
Empty DataFrame
Columns: [uid, loc_ID, lat_medoid, lng_medoid, home_day_count, home_hour_count, home_per_
↪hour_count, home_per_hour_duration, work_day_count, work_hour_count, work_per_hour_
↪count, work_per_hour_duration, home_count, work_count, home_duration, work_duration,␣
↪tot_seen_home_hours, tot_seen_home_days, tot_seen_work_hours, tot_seen_work_days, tot_
↪seen_hours, tot_seen_days, tot_stop_count, tot_stop_time, frac_home_count, frac_work_
↪count, frac_home_duration, frac_work_duration, home_delta_count, work_delta_count,␣
↪home_delta_duration, work_delta_duration, isHome, isWork]
Index: []

[0 rows x 34 columns]
```

Plot survival users

Here we check how many users feature a sufficient number of hours spent at home (work) during nighttime (daytime) and how many users feature a sufficient δ_h fraction of night (day) hours when the home (work) location happens to be the most visited one.

We check for different number of hours, days and delta the fraction of users surviving w.r.t. the total number when no filtering is applied.

We plot the number of users that we find in the dataset with at least two active days per home and work location, varying the minimum number of hours that we want to observe the user there (color of the line, see legend for the values) and varying the δ_h fraction of day/night hours when a location is the most visited one to be considered a valid work/home location.

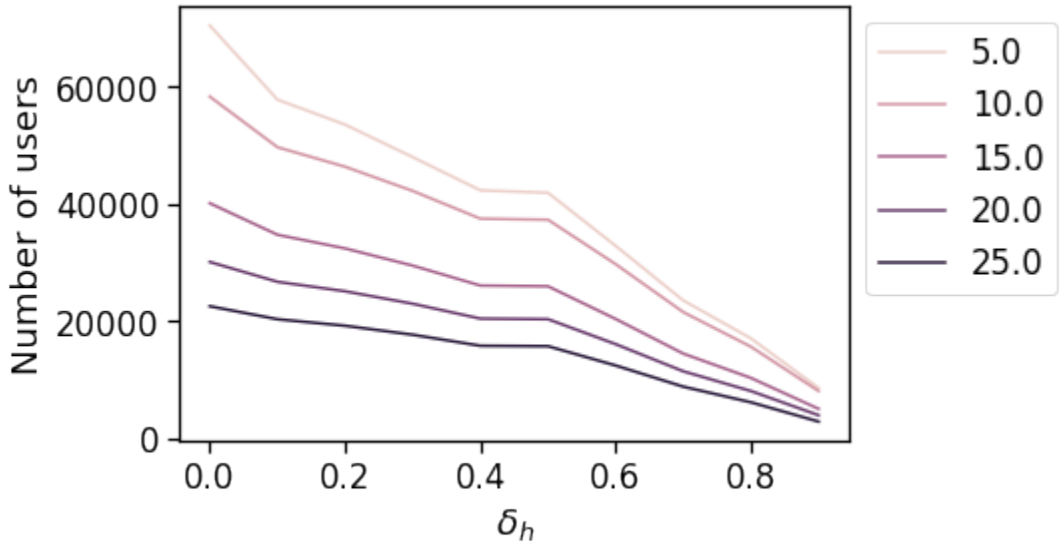
[26]:
```
n_days = list(range(2,15,2))
n_hours = list(range(5,26,5))
min_delta = list(np.arange(0, 1.,.1))
```

[27]:
```
path_out_df_hw_locs = os.path.join(OUT_DIR, 'out_df_hw_locs.pkl')
path_df_cnt_hw_locs = os.path.join(OUT_DIR, 'out_df_cnt_hw_locs.pkl')
if recomputeAll and os.path.exists(path_out_df_hw_locs):
    os.remove(path_out_df_hw_locs)
if recomputeAll and os.path.exists(path_df_cnt_hw_locs):
    os.remove(path_df_cnt_hw_locs)

if (not os.path.exists(path_out_df_hw_locs)) or (not os.path.exists(path_df_cnt_hw_
↪locs)):
    out_df_hw_locs, df_cnt_hw_locs = mk.stats.computeHomeWorkSurvival(user_locs_stats_hw_
↪separated,
                                        min_day_counts=n_days,
                                        min_hour_counts=n_hours,
                                        min_delta_durations=min_delta,
                                        limit_hw_locs=True,
                                        loc_col=locColName,
                                       )
    pd.to_pickle(out_df_hw_locs, path_out_df_hw_locs)
    pd.to_pickle(df_cnt_hw_locs, path_df_cnt_hw_locs)

out_df_hw_locs = pd.read_pickle(path_out_df_hw_locs)
df_cnt_hw_locs = pd.read_pickle(path_df_cnt_hw_locs)

df_cnt_hw_locs.head(0)
```

[27]:
```
Empty DataFrame
Columns: [tot_duration, n_days, n_hours, delta_count, delta_duration, n_users, with_home_
↪users, with_work_users, home_work_same_area_users, home_work_same_area_users_frac]
Index: []
```

[28]:
```
fig, ax = plt.subplots(1,1,figsize=(6,4))
sns.lineplot(x='delta_duration', y='n_users',
             data=df_cnt_hw_locs.query('n_days == 2'),
             hue='n_hours'
            )
plt.legend(loc=2, bbox_to_anchor=[1,1])
plt.xlabel(r"$\delta_h$")
plt.ylabel("Number of users")
```

[28]:
```
Text(0, 0.5, 'Number of users')
```



Convert the home work locations from long to wide format

[23]:
```
path_df_hw_locs_pd = os.path.join(OUT_DIR, 'df_hw_locs_pd.pkl')
if recomputeAll and os.path.exists(path_df_hw_locs_pd):
    os.remove(path_df_hw_locs_pd)

if not os.path.exists(path_df_hw_locs_pd):
    df_hw_locs_pd = mk.stats.compressLocsStats2hwTable(user_locs_stats_hw_separated)
    pd.to_pickle(df_hw_locs_pd, path_df_hw_locs_pd)

df_hw_locs_pd = pd.read_pickle(path_df_hw_locs_pd)
df_hw_locs_pd.head(0)
```

[23]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings]
Index: []
```

Limit the analysis to users with sufficient home/work stats

[30]:
```
min_pings_home_work = 5
```

[31]:
```
print("We start with", df_hw_locs_pd.shape[0], "users...")
clean_df_hw_locs_pd = df_hw_locs_pd.query('home_pings >= @min_pings_home_work & work_
↪pings >= @min_pings_home_work').copy(deep=True)
good_users_hw = clean_df_hw_locs_pd.index
print("... and we end with", clean_df_hw_locs_pd.shape[0], "users.")
```

```
We start with 187710 users...
... and we end with 91790 users.
```

6.6.4 Spatial and travel indicators

Now we start computing some indicators both for each area and for the single users: - for the area-based stats, in order to lower noise we will buffer the users home location of `homeWorkBufferMeters` meters and assign the user stat to all the cell grids touched by the buffer. Then we compute the min,max,mean,std,count stats for all the users assigned to an area. - for the user-based stats, we keep growing a `user_stats_table_df` which initially is a merge of the total per-user pings/days tables and his home/work stats. Then we will later add the stats on home/work travel time, radius of gyration etc.

Spatial distribution of users

Later we will compute the zonal statistics of urban spatial structure by looking at the typical travel/commuting features of the users residing in a given area. To this end, we compute once for all: - the users found residing in each area. - the distance of each user's home from the CBD closest to its workplace location. This allows us to check the functioning of the selected indicators on the distance from BD canceling out the policentric nature of a city. To do so, we pass the `multiple_cbd_latlon` array to the `mk.spatial.user_dist_cbds` that computes the distance of the user's workplace w.r.t. each and every BD of the policentric city. Then, the function saves in the `closest_cbd_dist` column the user's home euclidean distance from the reference BD.

[32]:
```
users_buffered_per_area = mk.stats.computeBufferStat(
                                gdf_stat=clean_df_hw_locs_pd.reset_index()[['lat_home',
↪'lng_home', uidColName]],
                                gdf_grid=gdf_aoi_grid,
                                column=uidColName,
                                aggregation=set,
                                lat_name='lat_home',
                                lon_name='lng_home',
                                local_EPSG=local_EPSG,
                                buffer=500)

# Normalize the number of users found to live in each area
users_buffered_per_area['nUsers'] = users_buffered_per_area[uidColName].apply(len)
users_buffered_per_area['nUsers'] /= users_buffered_per_area['nUsers'].sum()
```

```
/home/enrico/.local/share/virtualenvs/jupyter-EWzvnNR1/lib/python3.8/site-packages/
↪geopandas/array.py:275: ShapelyDeprecationWarning: The array interface is deprecated␣
↪and will no longer work in Shapely 2.0. Convert the '.coords' to a numpy array instead.
  return GeometryArray(vectorized.points_from_xy(x, y, z), crs=crs)
```

[33]:
```
# I also compute the distance from the different BDs of each valide user's home.
clean_df_hw_locs_pd = mk.spatial.user_dist_cbds(clean_df_hw_locs_pd, multiple_cbd_latlon)
clean_df_hw_locs_pd.head(0)
```

[33]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings, closest_cbd_idx, closest_cbd_dist]
Index: []
```

[34]:
```
user_stats_table_df = clean_df_hw_locs_pd.join(
                                            users_stats_df.set_index(uidColName),
                                            how='inner')
user_stats_table_df.head(0)
```

[34]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings, closest_cbd_idx, closest_cbd_dist, min_day, max_day, pings,␣
↪daysActive, daysSpanned, pingsPerDay, avg]
Index: []
```

Daily total traveled distance

We compute the per-user-day total travel distance.

Here we use the pings as they contain all the trajectories of the users, so we do not limit the user's trips to the as crows fly distance between their stops locations but we include the actual path.

We also compute the daily ROG computed on the positions of the user for a particular day.

[35]:
```
path_TTD_user_day = os.path.join(OUT_DIR, 'TTD_user_day.pkl')
if recomputeAll and os.path.exists(path_TTD_user_day):
    os.remove(path_TTD_user_day)
if not os.path.exists(path_TTD_user_day):
    result_ttd = mk.spatial.totalUserTravelDistance(df_pings_filtered,
                                                    doROG=True,
                                                    freq='1d').compute()
    pd.to_pickle(result_ttd, path_TTD_user_day)
result_ttd = pd.read_pickle(path_TTD_user_day)
result_ttd.head(0)
```

[35]:
```
Empty DataFrame
Columns: [ttd, rog, nPings]
Index: []
```

[36]:
```
# Merge the average daily rog and ttd keeping the days with a minimum of pings
user_stats_table_df = user_stats_table_df.join(
    result_ttd.query('nPings>5').groupby(level=0)[['ttd','rog']].agg('mean'),
    how='left',
)
user_stats_table_df.head(0)
```

[36]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings, closest_cbd_idx, closest_cbd_dist, min_day, max_day, pings,␣
↪daysActive, daysSpanned, pingsPerDay, avg, ttd, rog]
Index: []
```

[37]:
```
perAreaDailyTTD = mk.stats.userBasedBufferedStat(result_ttd, users_buffered_per_area,
                                        stat_col='ttd')
```

[38]:
```
perAreaDailyROG = mk.stats.userBasedBufferedStat(result_ttd, users_buffered_per_area,
                                        stat_col='rog')
```

[39]:
```
perAreaDailyTTD.head(0)
```

[39]:
```
Empty DataFrame
Columns: [ttd_min, ttd_max, ttd_mean, ttd_std, ttd_count]
Index: []
```

Radius of Gyration

We compute the per-user ROG w.r.t. his home and his mean position for the whole duration of the data.

In this case this will be a monthly ROG for the users. Below we will show how to compute the daily ROG.

[40]:
```
rogs_user_pd = mk.spatial.compute_ROG(
                                    exploded_stops_df,
                                    which='both',
                                    df_hw_locs=df_hw_locs_pd)
```

[41]:
```
path_users_rog = os.path.join(OUT_DIR, 'user_rog.pkl')
if recomputeAll and os.path.exists(path_users_rog):
    os.remove(path_users_rog)
if not os.path.exists(path_users_rog):
    rogs_user_pd = rogs_user_pd.compute()
    rogs_user_pd.to_pickle(path_users_rog)
rogs_user_pd = pd.read_pickle(path_users_rog)
rogs_user_pd.head(0)
```

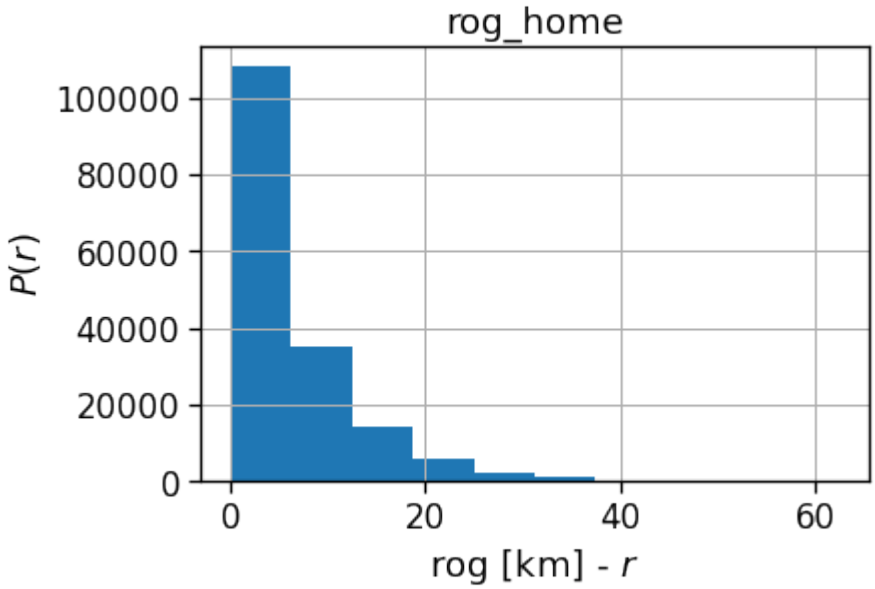
[41]:
```
Empty DataFrame
Columns: [n_pings, rog_home, com_home_lat, com_home_lng, rog_total, com_total_lat, com_
↪total_lng]
Index: []
```

[42]:
```
# Merge the average rog w.r.t. home and c.o.m.
user_stats_table_df = user_stats_table_df.join(rogs_user_pd.query('n_pings > 5'),
                                               how='left')
user_stats_table_df.head(0)
```

[42]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings, closest_cbd_idx, closest_cbd_dist, min_day, max_day, pings,␣
↪daysActive, daysSpanned, pingsPerDay, avg, ttd, rog, n_pings, rog_home, com_home_lat,␣
↪com_home_lng, rog_total, com_total_lat, com_total_lng]
Index: []

[0 rows x 27 columns]
```

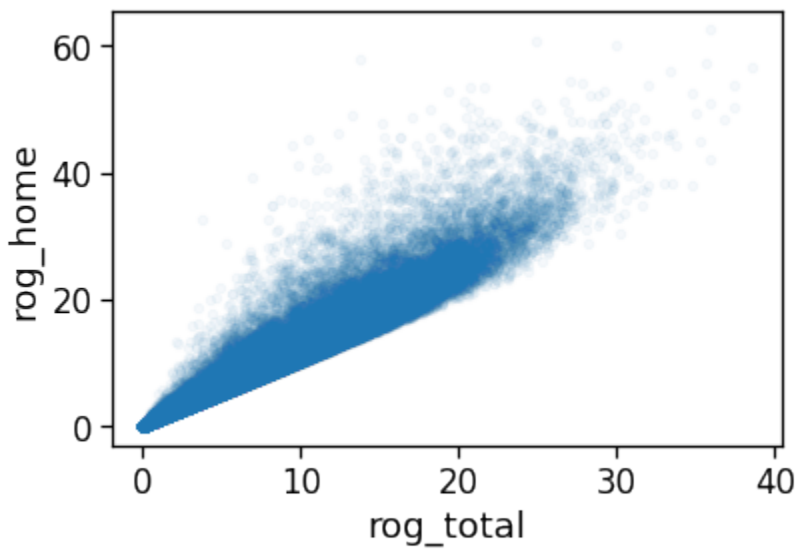
[43]:
```
rogs_user_pd[['rog_home']].hist()
plt.xlabel(r'rog [km] - $r$')
plt.ylabel(r'$P(r)$')
```

[43]:
```
Text(0, 0.5, '$P(r)$')
```



[44]:
```
rogs_user_pd.plot('rog_total', 'rog_home',
                  alpha=.04, kind='scatter')
```

[44]:
```
<AxesSubplot:xlabel='rog_total', ylabel='rog_home'>
```



Number of houses/offices

We: - project the user home/work locations to the local projection given by `local_EPSG`; - buffer their location by `homeWorkBufferMeters` meters to simulate a kernel density (with flat kernel over a circle); - count the number of offices and homes found in each cell and compute their absolute ratio;

NOTE >When determining the home location of a user, please consider that some data providers, like Cuebiq, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy. > >This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern. > >However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

[45]:
```
home_counts = mk.stats.computeBufferStat(
                                gdf_stat=clean_df_hw_locs_pd.reset_index()
                                              [['lat_home','lng_home', uidColName]],
                                gdf_grid=gdf_aoi_grid,
                                column=uidColName,
                                aggregation='nunique',
                                lat_name='lat_home',
                                lon_name='lng_home',
                                local_EPSG=local_EPSG,
                                buffer=500)
home_counts = home_counts.rename(columns={uidColName: 'n_homes'})['n_homes'].to_dict()
```

```
/home/enrico/.local/share/virtualenvs/jupyter-EWzvnNR1/lib/python3.8/site-packages/
↪geopandas/array.py:275: ShapelyDeprecationWarning: The array interface is deprecated␣
↪and will no longer work in Shapely 2.0. Convert the '.coords' to a numpy array instead.
  return GeometryArray(vectorized.points_from_xy(x, y, z), crs=crs)
```

[46]:
```
work_counts = mk.stats.computeBufferStat(
                                gdf_stat=clean_df_hw_locs_pd.reset_index()
                                              [['lat_work','lng_work', uidColName]],
                                gdf_grid=gdf_aoi_grid,
                                column=uidColName,
                                aggregation='nunique',
                                lat_name='lat_work',
                                lon_name='lng_work',
                                local_EPSG=local_EPSG,
                                buffer=500)
work_counts = work_counts.rename(columns={uidColName: 'n_works'})['n_works'].to_dict()
```

```
/home/enrico/.local/share/virtualenvs/jupyter-EWzvnNR1/lib/python3.8/site-packages/
↪geopandas/array.py:275: ShapelyDeprecationWarning: The array interface is deprecated␣
↪and will no longer work in Shapely 2.0. Convert the '.coords' to a numpy array instead.
  return GeometryArray(vectorized.points_from_xy(x, y, z), crs=crs)
```

Length and duration of commute

We want to compute the time of commuting starting from the original stops (the ones with the leaving datetime).

For each user we compute: - the straight line distance between home and work location; - the distance based on the actual time from departing home arriving in work cell and viceversa; - the "traffic-free" solution based on a osrm backend telling the travel time by car on the OSM street graph.

In order to have a running osrm-backend server you can use `docker` (see here for details):

```
wget http://download.geofabrik.de/europe/germany/berlin-latest.osm.pbf
osrm-extract -p your_profile berlin.osm.pbf
osrm-partition berlin.osrm
osrm-customize berlin.osrm
sudo docker run -t -i -p 5000:5000 -v "${PWD}:/data" osrm/osrm-backend osrm-routed --
↪algorithm=MLD /data/berlin-latest.osrm
```

[47]:
```
df_hw_locs_pd.head(0)
```

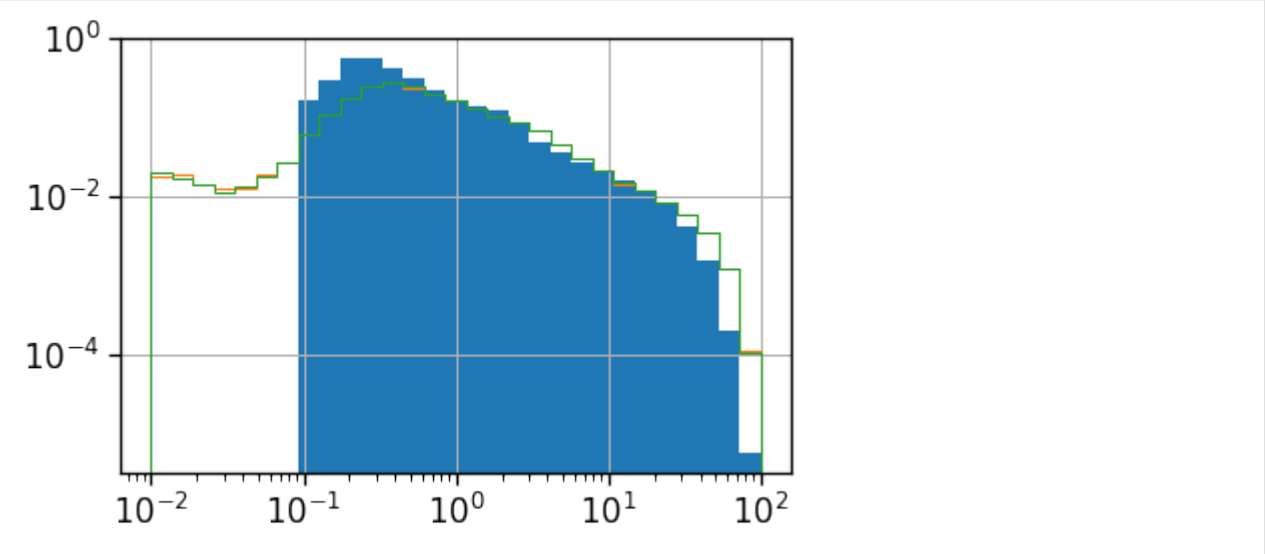
[47]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings]
Index: []
```

[48]:
```
path_df_user_home_work_distance_time = os.path.join(OUT_DIR, 'users_home_work_times_
↪distance.pkl')
if recomputeAll and os.path.exists(path_df_user_home_work_distance_time):
    os.remove(path_df_user_home_work_distance_time)
if not os.path.exists(path_df_user_home_work_distance_time):

    clean_df_hw_locs_pd = mk.stats.computeUserHomeWorkTripTimes(clean_df_hw_locs_pd,
                                                        osrm_url=osrm_url,
                                                        direction='both',
                                                        what='duration,distance',
                                                       )
    pd.to_pickle(clean_df_hw_locs_pd, path_df_user_home_work_distance_time)
clean_df_hw_locs_pd = pd.read_pickle(path_df_user_home_work_distance_time)
```

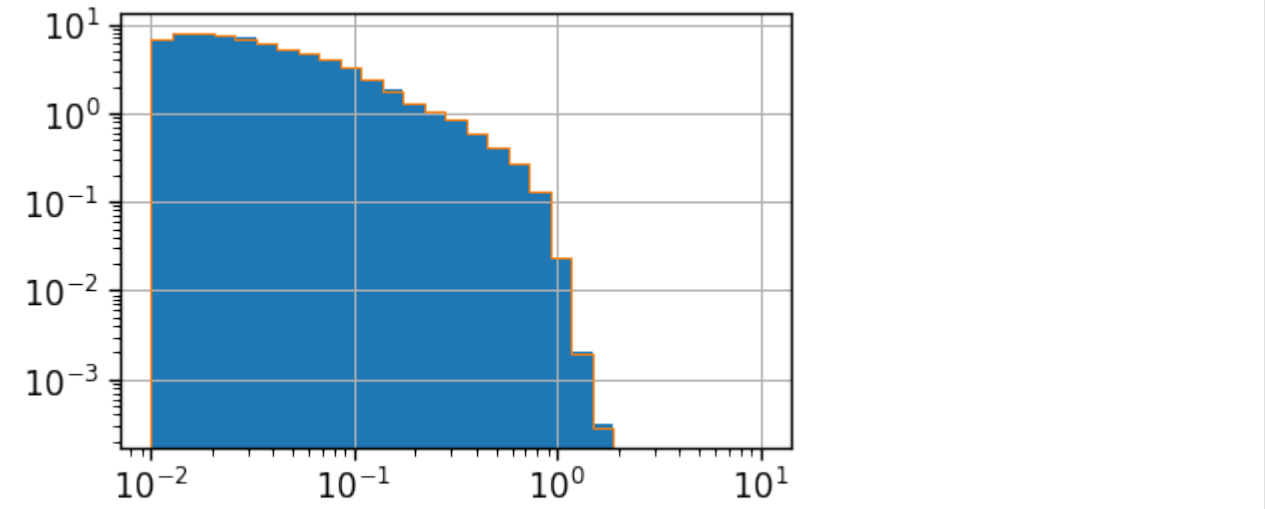
[49]:
```
clean_df_hw_locs_pd.head(0)
```

[49]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings, closest_cbd_idx, closest_cbd_dist, home_work_straight_dist, home_
↪work_osrm_time, home_work_osrm_dist, work_home_osrm_time, work_home_osrm_dist]
Index: []
```

[50]:
```
clean_df_hw_locs_pd['home_work_straight_dist'].hist(bins=np.logspace(-2,2,30),␣
↪density=True)
(clean_df_hw_locs_pd['home_work_osrm_dist']/1000).hist(bins=np.logspace(-2,2,30),␣
↪density=True,
                                        histtype='step', color='C1')
(clean_df_hw_locs_pd['work_home_osrm_dist']/1000).hist(bins=np.logspace(-2,2,30),␣
↪density=True,
                                        histtype='step', color='C2')
plt.loglog();
```



[51]:
```
(clean_df_hw_locs_pd['home_work_osrm_time']/3600).hist(bins=np.logspace(-2,1,30),␣
↪density=True)
(clean_df_hw_locs_pd['work_home_osrm_time']/3600).hist(bins=np.logspace(-2,1,30),␣
↪density=True,
                                        histtype='step', color='C1')
plt.loglog();
```



Real commuting duration

These are instead the real time from home to work (or the other way around).

These are computed by looking at the time-ordered sequence of the stops of the user, looking at all the time when the user was at home and then at work within a given time window.

[52]:
```
user_time_trips_hw = mk.stats.userRealHomeWorkTimes(df_stops=users_stop_locs_df,
                                                    home_work_locs=clean_df_hw_locs_pd,
                                                    direction='both',
                                                    location_col=locColName)
user_time_trips_hw
```

[52]:
```
Dask DataFrame Structure:
                time_trips_hw speed_trips_hw time_trips_wh speed_trips_wh home_work_
↪straight_dist home_work_osrm_time work_home_osrm_time home_work_osrm_dist work_home_
↪osrm_dist
npartitions=200
                       object         object        object         object               ␣
↪ float64             float64             float64             float64            ␣
↪float64
                          ...            ...           ...            ...               ␣
↪     ...                  ...                 ...                 ...                ␣
↪...
...                       ...            ...           ...            ...               ␣
↪     ...                  ...                 ...                 ...                ␣
↪...
                          ...            ...           ...            ...               ␣
↪     ...                  ...                 ...                 ...                ␣
↪...
                          ...            ...           ...            ...               ␣
↪     ...                  ...                 ...                 ...                ␣
↪...
Dask Name: per_user_real_home_work_times, 16676 tasks
```

[53]:
```
path_df_user_timetrips_hw = os.path.join(OUT_DIR, 'users_time_trips_home_work.pkl')
if recomputeAll and os.path.exists(path_df_user_timetrips_hw):
    os.remove(path_df_user_timetrips_hw)
if not os.path.exists(path_df_user_timetrips_hw):

    user_time_trips_hw = user_time_trips_hw.compute()

    pd.to_pickle(user_time_trips_hw, path_df_user_timetrips_hw)
user_time_trips_hw = pd.read_pickle(path_df_user_timetrips_hw)
```

[54]:
```
user_time_trips_hw.head(0)
```

[54]:
```
Empty DataFrame
Columns: [time_trips_hw, speed_trips_hw, time_trips_wh, speed_trips_wh, home_work_
↪straight_dist, home_work_osrm_time, work_home_osrm_time, home_work_osrm_dist, work_
↪home_osrm_dist]
Index: []
```

[55]:
```
user_stats_table_df.head(0)
```

[55]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings, closest_cbd_idx, closest_cbd_dist, min_day, max_day, pings,␣
↪daysActive, daysSpanned, pingsPerDay, avg, ttd, rog, n_pings, rog_home, com_home_lat,␣
↪com_home_lng, rog_total, com_total_lat, com_total_lng]
Index: []

[0 rows x 27 columns]
```

[56]:
```
# Merge the new columns computed time
user_stats_table_df = user_stats_table_df.join(user_time_trips_hw, how='left')
for c in ['time_trips_hw', 'speed_trips_hw', 'time_trips_wh', 'speed_trips_wh']:
    if c in user_stats_table_df.columns:
        user_stats_table_df[c] = user_stats_table_df[c].apply(lambda l: None if l is␣
↪None or len(l)==0 else np.mean(l))
user_stats_table_df.head(0)
```

[56]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings, closest_cbd_idx, closest_cbd_dist, min_day, max_day, pings,␣
↪daysActive, daysSpanned, pingsPerDay, avg, ttd, rog, n_pings, rog_home, com_home_lat,␣
↪com_home_lng, rog_total, com_total_lat, com_total_lng, time_trips_hw, speed_trips_hw,␣
↪time_trips_wh, speed_trips_wh, home_work_straight_dist, home_work_osrm_time, work_home_
↪osrm_time, home_work_osrm_dist, work_home_osrm_dist]
Index: []

[0 rows x 36 columns]
```

[57]:
```
path_df_timetrip_stats = os.path.join(OUT_DIR, 'time_trips_stats_home_work.pkl')
if recomputeAll and os.path.exists(path_df_timetrip_stats):
    os.remove(path_df_timetrip_stats)
if not os.path.exists(path_df_timetrip_stats):

    time_trips_stats_df = mk.stats.computeTripTimeStats(df_trip_times=user_time_trips_hw,
                                        df_hw_locs=clean_df_hw_locs_pd,
                                        gdf_grid=gdf_aoi_grid,
                                        local_EPSG=local_EPSG,
                                        buffer_m=500)

    pd.to_pickle(time_trips_stats_df, path_df_timetrip_stats)
time_trips_stats_df = pd.read_pickle(path_df_timetrip_stats)
```

[58]:
```
time_trips_stats_df.head(0)
```

[58]:
```
Empty GeoDataFrame
Columns: [uid, time_trips_hw_min, time_trips_hw_max, time_trips_hw_avg, time_trips_hw_
↪std, time_trips_wh_min, time_trips_wh_max, time_trips_wh_avg, time_trips_wh_std, speed_
↪trips_hw_min, speed_trips_hw_max, speed_trips_hw_avg, speed_trips_hw_std, speed_trips_
↪wh_min, speed_trips_wh_max, speed_trips_wh_avg, speed_trips_wh_std, home_work_straight_
↪dist_min, home_work_straight_dist_max, home_work_straight_dist_avg, home_work_straight_
↪dist_std, home_work_osrm_time_min, home_work_osrm_time_max, home_work_osrm_time_avg,␣
↪home_work_osrm_time_std, work_home_osrm_time_min, work_home_osrm_time_max, work_home_
↪osrm_time_avg, work_home_osrm_time_std, home_work_osrm_dist_min, home_work_osrm_dist_
```

(continues on next page)

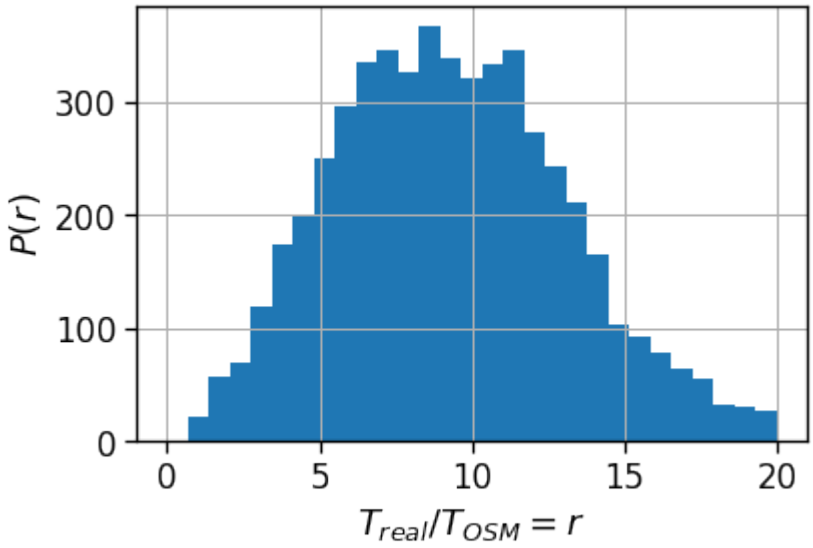
(continued from previous page)

```
↪max, home_work_osrm_dist_avg, home_work_osrm_dist_std, work_home_osrm_dist_min, work_
↪home_osrm_dist_max, work_home_osrm_dist_avg, work_home_osrm_dist_std]
Index: []

[0 rows x 37 columns]
```

[59]:
```
# Compute the average time fraction
time_trips_stats_df['avg_realT_frac_osmT'] = time_trips_stats_df.apply(lambda r:
                                                    r['time_trips_hw_avg'] / max(0.1, r[
↪'home_work_osrm_time_avg']/3600.)
                                                        if not (
                                                            pd.isna(r['time_trips_hw_avg
↪'])
                                                            or pd.isna(r['home_work_osrm_
↪time_avg'])
                                                        ) else None, axis=1)
user_stats_table_df['avg_realT_frac_osmT'] = user_stats_table_df.apply(lambda r:
                                                    r['time_trips_hw'] / max(0.1, r[
↪'home_work_osrm_time']/3600.)
                                                        if not (
                                                            pd.isna(r['time_trips_hw'])
                                                            or pd.isna(r['home_work_osrm_
↪time'])
                                                        ) else None, axis=1)
```

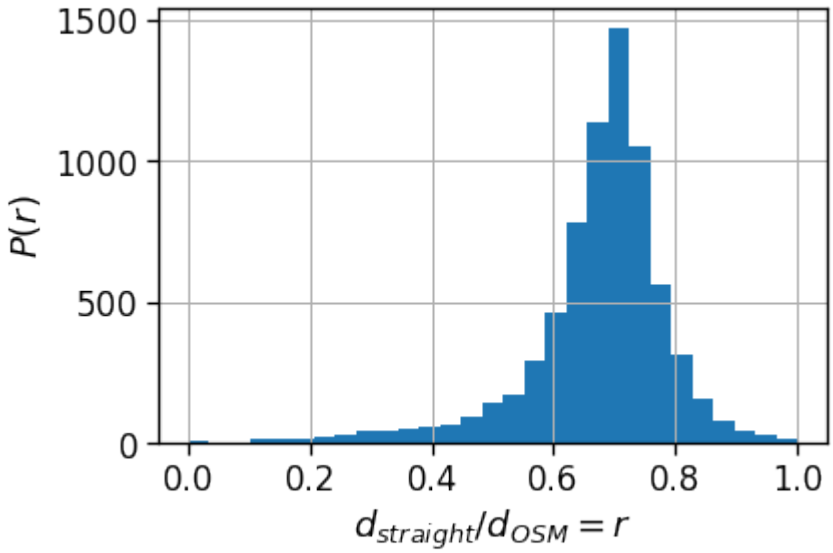
[60]:
```
# Compute the average distance fraction
time_trips_stats_df['avg_realD_frac_osmD'] = time_trips_stats_df.apply(lambda r:
                                                    r['home_work_straight_dist_avg'] /␣
↪max(0.1, r['home_work_osrm_dist_avg']/1000.)
                                                        if not (
                                                            pd.isna(r['home_work_
↪straight_dist_avg'])
                                                            or pd.isna(r['home_work_osrm_
↪dist_avg'])
                                                        ) else None, axis=1)
user_stats_table_df['avg_realD_frac_osmD'] = user_stats_table_df.apply(lambda r:
                                                    r['home_work_straight_dist'] / max(0.
↪1, r['home_work_osrm_dist']/1000.)
                                                        if not (
                                                            pd.isna(r['home_work_
↪straight_dist'])
                                                            or pd.isna(r['home_work_osrm_
↪dist'])
                                                        ) else None, axis=1)
```

[61]:
```
time_trips_stats_df['avg_realT_frac_osmT'].hist(bins=np.linspace(0,20,30))
plt.xlabel(r'$T_{real} / T_{OSM} = r$')
plt.ylabel(r'$P(r)$')
plt.savefig(os.path.join(OUT_DIR_FIG, 'avg_realT_frac_osmT_hist.pdf'))
```



Also the distribution of the ratio between the as-crows-fly and the "theoretical one" shows that only few people have access to a nearly straight home-work commuting route.

[62]:
```
time_trips_stats_df['avg_realD_frac_osmD'].hist(bins=np.linspace(0,1.,30))
plt.xlabel(r'$d_{straight} / d_{OSM} = r$')
plt.ylabel(r'$P(r)$')
plt.savefig(os.path.join(OUT_DIR_FIG, 'avg_realD_frac_osmD_hist.pdf'))
```



6.6.5 Urban spatial structure

In this section we investigate the spatial structure of the city. In particular, we inspect the spatial functioning of some relevant observables and study the their spatial dependence on their distance to the Central Business District (CBD) or to the user's reference Business District (BD, see below for definition).

We will compute: - the daily user's Radius Of Gyration (ROG) and Total Traveled Distance (TTD); - the ratio between the number of homes and office locations found in each cell grid; - the mean home-work commuting length and duration per user and per cell, also with respect to the traffic-free scenario computed with OSRM.

However, we start with another perspective, the land use detection.

Compute areas profiles

To compute the land use of single areas we refer to the work of Toole et al. > Toole, J. L., Ulm, M., González, M. C. & Bauer, D. Inferring land use from mobile phone activity. > in Proceedings of the ACM SIGKDD International Workshop on Urban Computing - UrbComp 12 (ACM Press, 2012).

i.e., we start computing the activity of the grid cells (i.e., the number of people observed in a given cell at each hour of the day in specific days) and we later normalize this volume to cancel out the fluctuactions in the number of users/pings registered in the data.

In this case we want to detect the weekdays landuse of an area, by looking at: - weekdays only (Monday to Friday); - keep under consideration areas with sufficient statistics only.

We compute the hourly occupation rate $a(c,h,d) = N_u(c,h,d)/N_u^{TOT}(d)$ of a cell c with respect to the total volume of users observed on that day $N_u^{TOT}(d)$.

[63]:
```
profile_period = "day"
profile_timeBin = "H"
signal_column = "frac_users"
weekdays = [0,1,2,3,4]
min_area_covered_hours = 72
min_area_observed_pings = 500
```

Rename the columns

We are interested in the time bin when the stop is present, so we overwrite the `datetime` column name with the `stops` one (the latter contains the info about the repetition of the stop.)

[64]:
```
exploded_stops_df[dttColName] = exploded_stops_df['stops']
```

Compute the profiles

For each day we compute the hourly profile of the single grid cell.

[65]:
```
total_profiles_df = mk.temporal.computeTemporalProfile(exploded_stops_df,
                                                       timeBin=profile_timeBin,
                                                       weekdays=weekdays,
                                                       byArea=True,
                                                       profile=profile_period,
                                                       normalization="total")
total_profiles_df
```

```
/home/enrico/.local/share/virtualenvs/jupyter-EWzvnNR1/lib/python3.8/site-packages/dask/
↪dataframe/core.py:3935: UserWarning:
You did not provide metadata, so Dask is running your function on a small dataset to␣
↪guess output types. It is possible that Dask will guess incorrectly.
To provide an explicit output types or to silence this message, please provide the␣
↪`meta=` keyword, as described in the map or apply function that you are using.
  Before: .apply(func)
  After:  .apply(func, meta=('users', 'int64'))

  warnings.warn(meta_warning(meta))
```

[65]:
```
Dask DataFrame Structure:
                           day tile_ID               H  users  pings tot_users tot_pings␣
↪frac_users frac_pings pings_per_user profile_hour
npartitions=10
                datetime64[ns]   int64  datetime64[ns]  int64  int64     int64     int64␣
↪   float64    float64        float64        int64
                           ...     ...             ...    ...    ...       ...       ...␣
↪       ...        ...            ...          ...
...                        ...     ...             ...    ...    ...       ...       ...␣
↪       ...        ...            ...          ...
                           ...     ...             ...    ...    ...       ...       ...␣
↪       ...        ...            ...          ...
                           ...     ...             ...    ...    ...       ...       ...␣
↪       ...        ...            ...          ...
Dask Name: assign, 15765 tasks
```

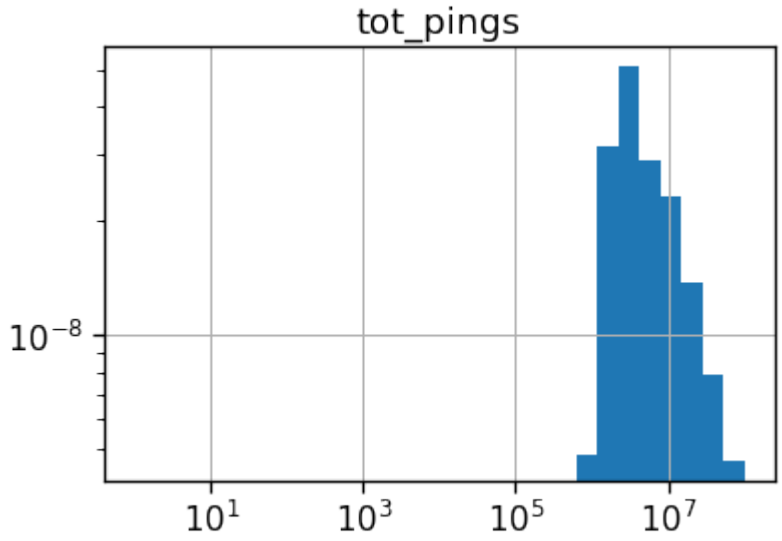
[66]:
```
path_total_profiles = os.path.join(OUT_DIR, 'total_profiles.pkl')
if recomputeAll and os.path.exists(path_total_profiles):
    os.remove(path_total_profiles)
if not os.path.exists(path_total_profiles):
    total_profiles_df = total_profiles_df.compute()
    total_profiles_df.to_pickle(path_total_profiles)
total_profiles_df = pd.read_pickle(path_total_profiles)
```

[67]:
```
total_profiles_df.head(0)
```

[67]:
```
Empty DataFrame
Columns: [day, tile_ID, H, users, pings, tot_users, tot_pings, frac_users, frac_pings,␣
↪pings_per_user, profile_hour]
Index: []
```

[68]:
```
# Visualize the number of pings per area
total_profiles_df.groupby('tile_ID').agg({'tot_pings': sum}).hist(bins=np.logspace(0, 8,␣
↪30), density=True)
plt.loglog()
```

[68]:
```
[]
```



[69]:
```
# We select the good areas to be the one with sufficient hours covered
# and with at least some stops observed
good_areas = set(total_profiles_df.groupby('tile_ID')
                                  .agg({'tot_pings': sum, 'H': 'nunique'})
                                     .query('H > @min_area_covered_hours & tot_pings >
↪ @min_area_observed_pings').index)
len(good_areas)
```

[69]:
```
4998
```

[70]:
```
# Then I clean the profiles to keep only valid areas and compute the residual activities.
↪..
cleaned_profiles = total_profiles_df.query('tile_ID in @good_areas').copy(deep=True)
residual_activities, mappings =  mk.temporal.computeResiduals(cleaned_profiles,
                                                              signal_column=signal_column,
                                                              profile=profile_period)
```

Infer land use

We now cluster the per-areas profiles of users presence fraction using a hierarchical clustering with a Ward linking and a cosine distance.

We inspect the Calinski-Harabaz score to find a local maximum to select the number of clusters that bast partition the data.

In this case we use 6 clusters.

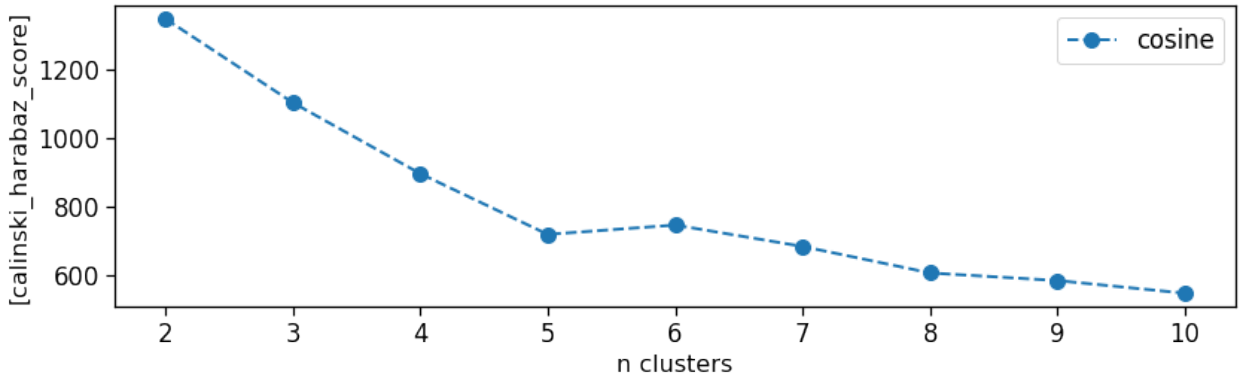
[71]:
```
signal_to_use = "residual"
metric = "cosine"  # The metric to be used in computing the distance matrix

results_clusters = mk.tools.computeClusters(residual_activities,
                                                    signal_to_use,
                                                    metric=metric,
                                                    nClusters=range(2,11))
```

(continues on next page)

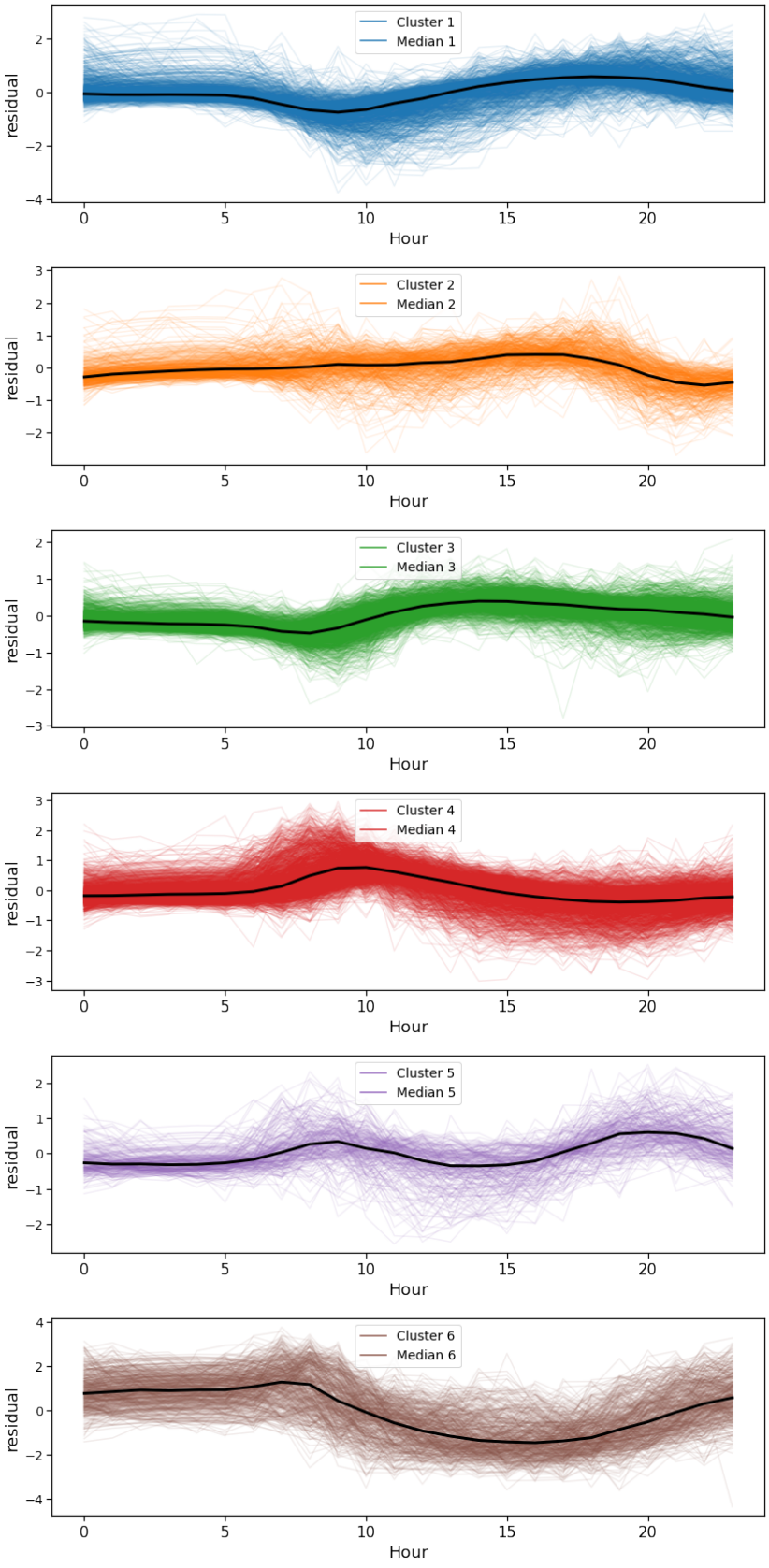
(continued from previous page)

```
# Visualize score
ax_score = mk.tools.checkScore(results_clusters)
plt.savefig(os.path.join(OUT_DIR_FIG, 'landuse_score.pdf'))
```

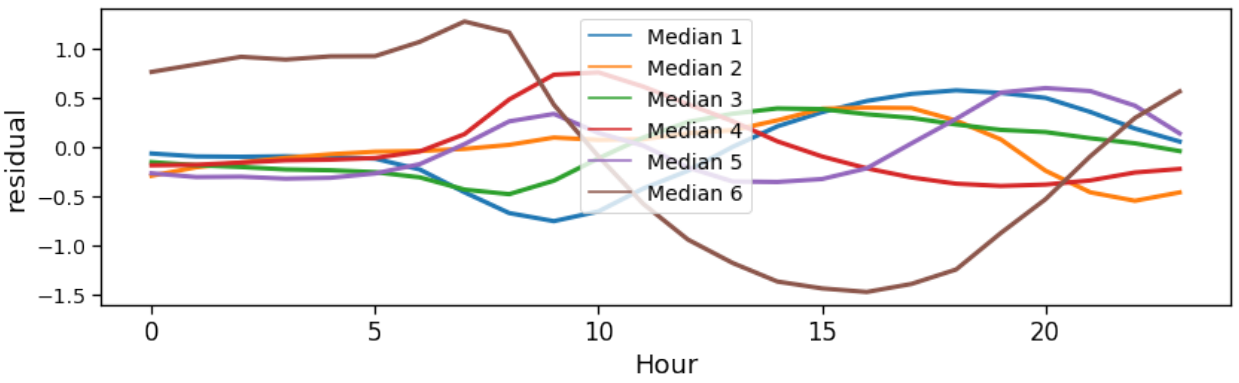
```
Done n clusters = 02
Done n clusters = 03
Done n clusters = 04
Done n clusters = 05
Done n clusters = 06
Done n clusters = 07
Done n clusters = 08
Done n clusters = 09
Done n clusters = 10
```



[72]:
```
nClusters = 6
ax = mk.tools.visualizeClustersProfiles(results_clusters,
                                        nClusts=nClusters,
                                        showMean=False,
                                        showMedian=True,
                                        showCurves=True)
plt.savefig(os.path.join(OUT_DIR_FIG, 'landuse_curves_all.pdf'))
```


Cluster 1
Median 1
residual
Hour
Cluster 2
Median 2
residual
Hour
Cluster 3
Median 3
residual
Hour
Cluster 4
Median 4
residual
Hour
Cluster 5
Median 5
residual
Hour
Cluster 6
Median 6
residual
Hour

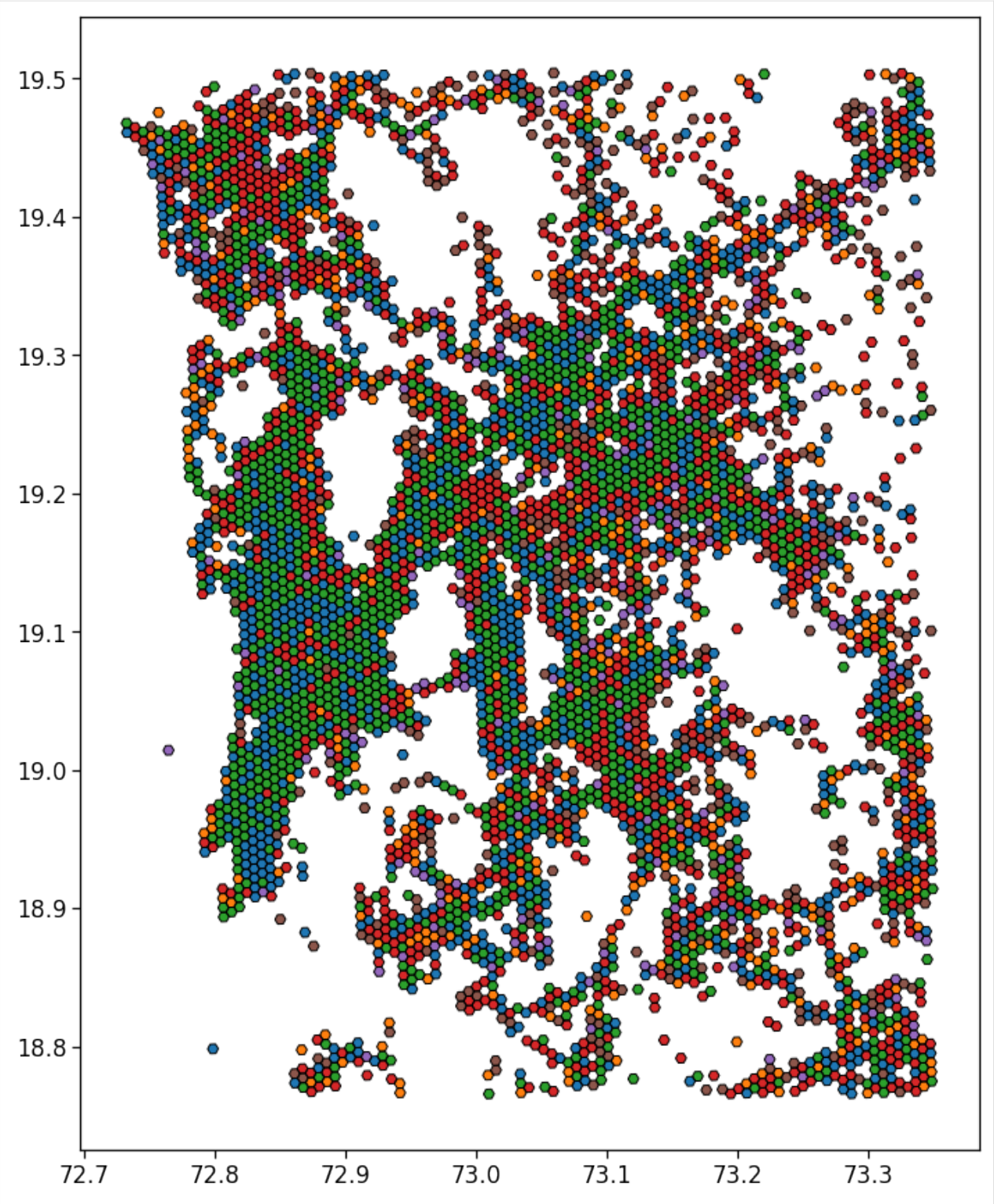
[73]:
```
ax = mk.tools.visualizeClustersProfiles(results_clusters,
                                        nClusts=nClusters,
                                        showMean=False,
                                        showMedian=True,
                                        showCurves=False,
                                        together=True,
                                       )
plt.savefig(os.path.join(OUT_DIR_FIG, 'landuse_curves_median.pdf'))
```



Given the residual activities found, we categorize the landuse by looking at when an area is more active than the overall average. A peak during night-time indicates a residential area, whereas a peak during office hours is an hint for a commercial/workplace oriented area.

[74]:
```
rename_clusters = {
    1: 'Workplace/Leisure',
    2: 'Commercial/Workplace',
    3: 'Workplace',
    4: 'Schools/Commuting',
    5: 'Commuting/Leisure',
    6: 'Residential',
    7: '',
    -1: 'Unclassified',
}
```

[75]:
```
gdf_aoi_grid_landuse, ax = mk.tools.plotClustersMap(gdf_aoi_grid, results_clusters,␣
↪mappings, nClusts=nClusters);
plt.savefig(os.path.join(OUT_DIR_FIG, 'landuse_map.pdf'))
```



[76]:
```
gdf_aoi_grid_landuse['scope'] = gdf_aoi_grid_landuse['cluster'].replace(rename_clusters)
```

[77]:
```
gdf_aoi_grid_landuse.to_file(out_grid_cells_file, driver='GPKG')
```

[78]:
```
df_hw_locs_pd.head(0)
```

[78]:
```
Empty DataFrame
Columns: [tot_pings, home_loc_ID, lat_home, lng_home, home_pings, work_loc_ID, lat_work,␣
↪lng_work, work_pings]
Index: []
```

6.6.6 Map the indicators

In the following, we start mapping the computed indicators and check their functioning with distance from CBD.

Users density

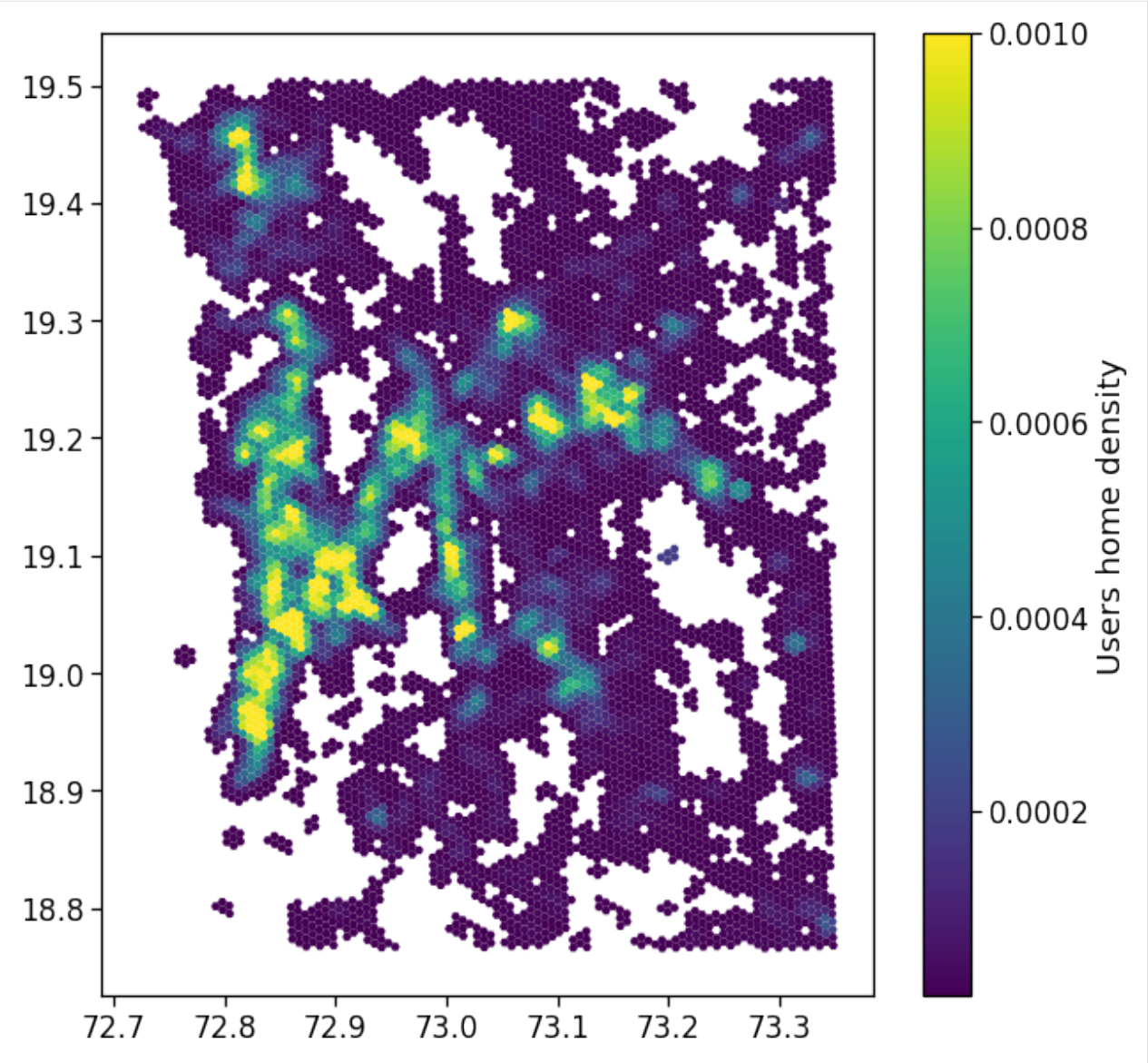
We plot the mapping of where most of the home location are found.

We show the density by dividing the number of houses per area by their total count.

From here on, we will simply join the indicators computed in the previous section with the GeoDataFrame containing the areas to visualize the spatial mapping of the observables.

[79]:
```
gdf_aoi_grid_landuse = pd.merge(gdf_aoi_grid_landuse,
                                users_buffered_per_area[['nUsers']].reset_index(),
                                on=zidColName,
                                how='left')
```

[80]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
gdf_aoi_grid_landuse.plot('nUsers', vmin=1e-5, vmax=1e-3,
                          legend=True, ax=ax,
                          legend_kwds={'label': r'Users home density'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'user_home_density_map.pdf'))
```



Radius of Gyration

We compute the ROG w.r.t. home and day/week medoid/mean position.

We also compute the average radius of gyration per grid cell mapping all the users with the home in that cell.

We simply merge the radius of gyration table with the user's home location and aggregate over the `tile_ID`.

We also normalize the user count to hide the dataset statistics.

[81]:
```
cleaned_users_rog = rogs_user_pd.query('n_pings > 10').copy(deep=True)
cleaned_user_stats_table_df = user_stats_table_df.join(cleaned_users_rog[[]], how='inner
↪')
cleaned_user_stats_table_df = cleaned_user_stats_table_df.query('home_work_straight_dist␣
↪> .75')
print('Selected', cleaned_users_rog.shape[0], 'out of', rogs_user_pd.shape[0], 'users␣
```

(continues on next page)

(continued from previous page)

```
↪for rog.')
print('Selected', cleaned_user_stats_table_df.shape[0], 'out of', user_stats_table_df.
↪shape[0], 'users for stats.')
```

```
Selected 160891 out of 187710 users for rog.
Selected 69890 out of 91790 users for stats.
```

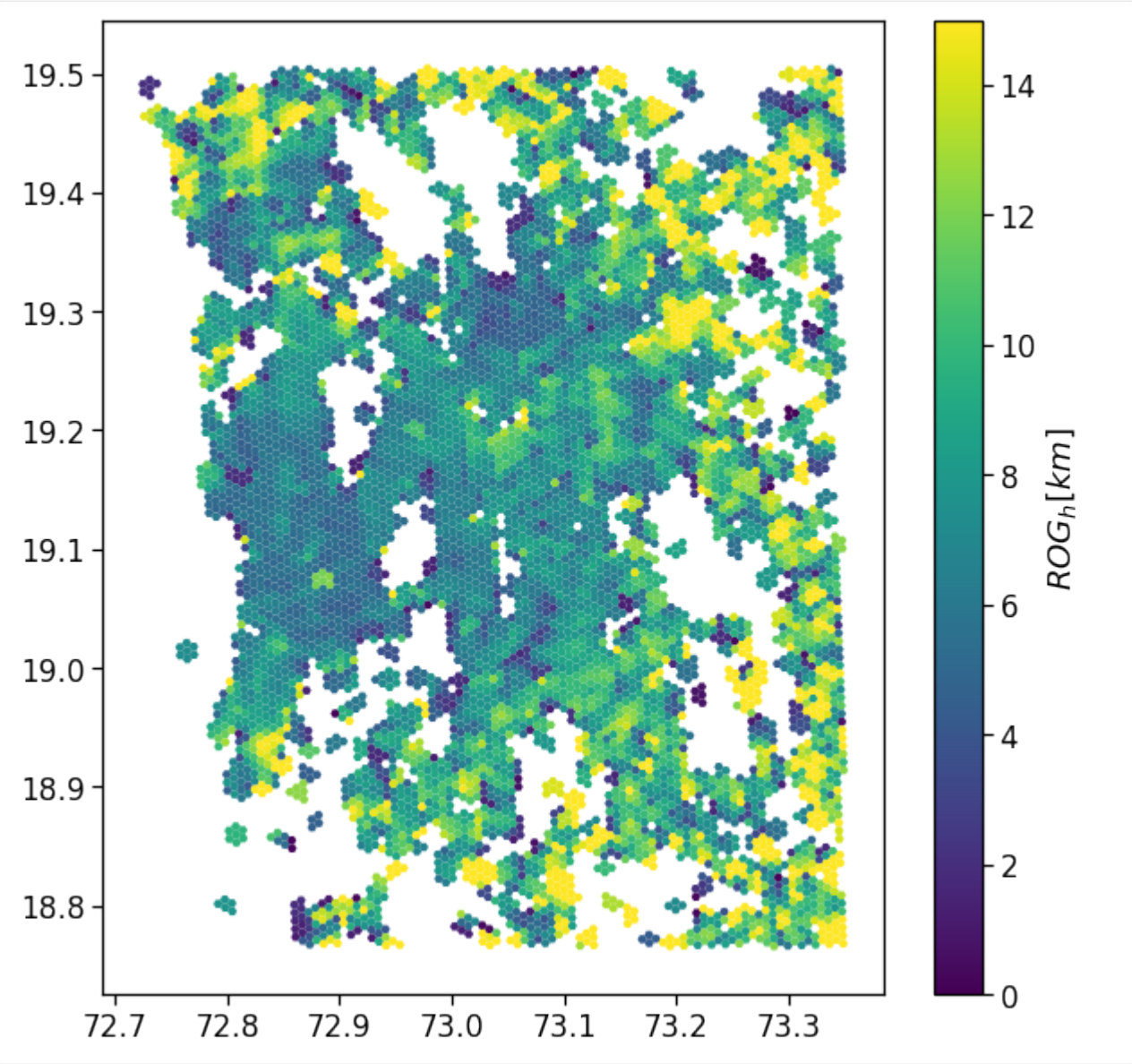
[82]:
```
perAreaTotalROG = mk.stats.userBasedBufferedStat(cleaned_users_rog,
                                                 users_buffered_per_area,
                                                 stat_col='rog_total')

perAreaHomeROG = mk.stats.userBasedBufferedStat(cleaned_users_rog,
                                                users_buffered_per_area,
                                                stat_col='rog_home')
```

[83]:
```
perAreaTotalROG.head(0)
```

[83]:
```
Empty DataFrame
Columns: [rog_total_min, rog_total_max, rog_total_mean, rog_total_std, rog_total_count]
Index: []
```

[84]:
```
for c in list(perAreaTotalROG.columns) + list(perAreaHomeROG.columns):
    if c in gdf_aoi_grid_landuse:
        del gdf_aoi_grid_landuse[c]

gdf_aoi_grid_landuse = pd.merge(gdf_aoi_grid_landuse,
                                perAreaTotalROG.reset_index(),
                                on=zidColName,
                                how='left')

gdf_aoi_grid_landuse = pd.merge(gdf_aoi_grid_landuse,
                                perAreaHomeROG.reset_index(),
                                on=zidColName,
                                how='left')
```

[85]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
gdf_aoi_grid_landuse.plot('rog_home_mean', vmin=.0, vmax=15.,
                          legend=True, ax=ax,
                          legend_kwds={'label': r'$ROG_{h} [km]$'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'user_rog_home_map.pdf'))
```



[86]:
```
# Save the result
gdf_aoi_grid_landuse.to_file(out_grid_cells_file, driver='GPKG')
```

Compute the distance w.r.t. the city center

To inspect the functioning of each observables w.r.t. the CBD, we compute the distance of each grid cell with respect to the city center in the `cbd_dist` column.

Also, to smooth the plotting, we divide the distances in bins of 2.5 km to group the observables and reduce the noise.

[87]:
```
gdf_aoi_grid_landuse['cbd_dist'] = haversine_pairwise(
    np.hstack((
        gdf_aoi_grid_landuse.geometry.centroid.y.values.reshape(-1,1),
        gdf_aoi_grid_landuse.geometry.centroid.x.values.reshape(-1,1))
```

(continues on next page)

(continued from previous page)

```
    ),
    center_of_city,
)

cleaned_user_stats_table_df['cbd_dist'] = haversine_pairwise(
    np.hstack((
        cleaned_user_stats_table_df['lat_home'].values.reshape(-1,1),
        cleaned_user_stats_table_df['lng_home'].values.reshape(-1,1)
    )),
    center_of_city,
)

user_stats_table_df['cbd_dist'] = haversine_pairwise(
    np.hstack((
        user_stats_table_df['lat_home'].values.reshape(-1,1),
        user_stats_table_df['lng_home'].values.reshape(-1,1)
    )),
    center_of_city,
)

dist_bin = 2.5
gdf_aoi_grid_landuse['cbd_dist_bin'] = np.floor(gdf_aoi_grid_landuse['cbd_dist'] / dist_
↪bin)*dist_bin
cleaned_user_stats_table_df['cbd_dist_bin']  = np.floor(cleaned_user_stats_table_df['cbd_
↪dist'] / dist_bin)*dist_bin
cleaned_user_stats_table_df['closest_cbd_dist_bin']  = np.floor(cleaned_user_stats_table_
↪df['closest_cbd_dist'] / dist_bin)*dist_bin
user_stats_table_df['closest_cbd_dist_bin']  = np.floor(user_stats_table_df['closest_cbd_
↪dist'] / dist_bin)*dist_bin
```

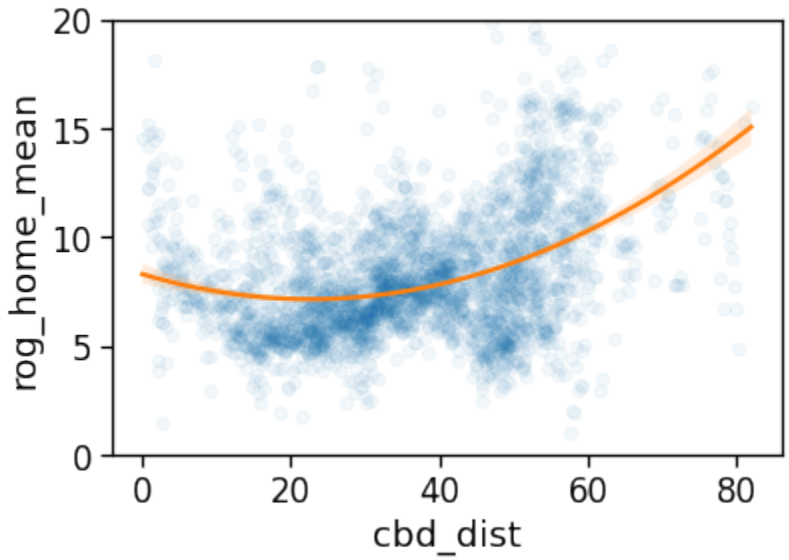
```
/tmp/ipykernel_1452718/688516821.py:3: UserWarning: Geometry is in a geographic CRS.␣
↪Results from 'centroid' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project␣
↪geometries to a projected CRS before this operation.

  gdf_aoi_grid_landuse.geometry.centroid.y.values.reshape(-1,1),
/tmp/ipykernel_1452718/688516821.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.

  gdf_aoi_grid_landuse.geometry.centroid.x.values.reshape(-1,1))
```

We also check the correlation between the ROG and the distance from the CBD.

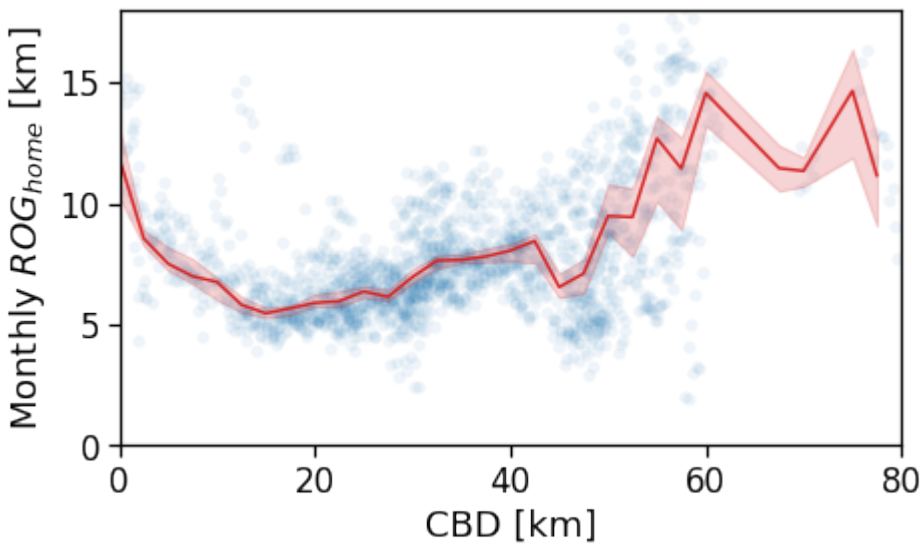
[88]:
```
# Linear interpolation of rog vs distance...
tmp_data = gdf_aoi_grid_landuse[[
    'rog_home_mean',
    'cbd_dist'
]].dropna()

model = sm.GLS(tmp_data['rog_home_mean'],
               tmp_data['cbd_dist'])
res = model.fit()
res.summary()
```

[88]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            GLS Regression Results
==============================================================================
Dep. Variable:          rog_home_mean   R-squared (uncentered):                   0.728
Model:                            GLS   Adj. R-squared (uncentered):              0.728
Method:                 Least Squares   F-statistic:                          1.951e+04
Date:                Mon, 16 May 2022   Prob (F-statistic):                        0.00
Time:                        15:52:26   Log-Likelihood:                         -22435.
No. Observations:                7289   AIC:                                  4.487e+04
Df Residuals:                    7288   BIC:                                  4.488e+04
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
cbd_dist       0.1912      0.001    139.669      0.000       0.189       0.194
==============================================================================
Omnibus:                     2651.633   Durbin-Watson:                   0.853
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            22667.379
Skew:                           1.503   Prob(JB):                         0.00
Kurtosis:                      11.099   Cond. No.                         1.00
==============================================================================

Notes:
[1] R² is computed without centering (uncentered) since the model does not contain a␣
↪constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
"""
```

[89]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='rog_home_mean',
    data=gdf_aoi_grid_landuse.query('rog_home_count>15'),
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.05},
    order=2,
)
plt.ylim(0,20)
plt.savefig(os.path.join(OUT_DIR_FIG, 'rogHome_vs_cbd_scatter.pdf'))
```



[90]:
```
tmp_data = gdf_aoi_grid_landuse.query('rog_home_count > 50')\
                                .copy(deep=True)
fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='rog_home_mean',
            xlabel='CBD [km]',
            ylabel=r'Monthly $ROG_{home}$ [km]',
            xlim=[0,80],
            ylim=[0,18],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'rogHome_vs_cbd_scatterLine.pdf'))
```

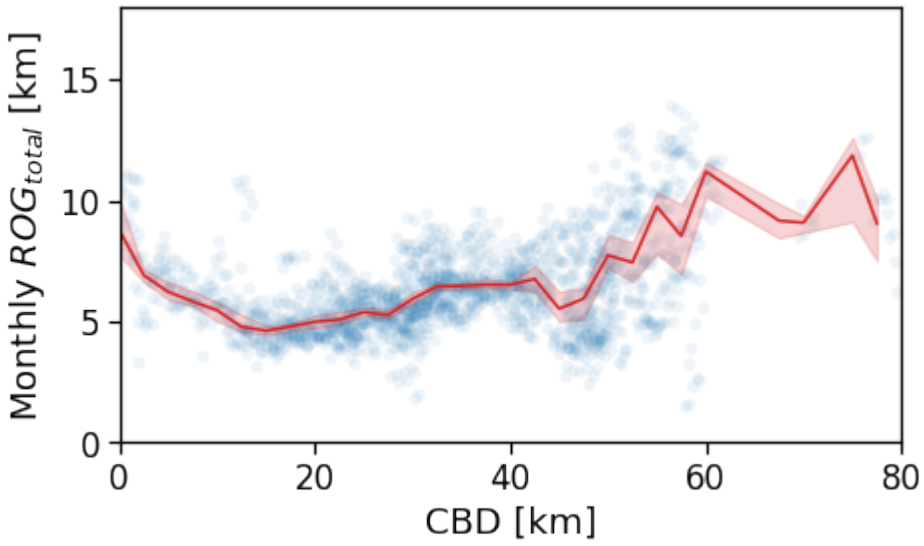


[91]:
```
tmp_data = gdf_aoi_grid_landuse.query('rog_home_count > 50')\
```

(continues on next page)

(continued from previous page)

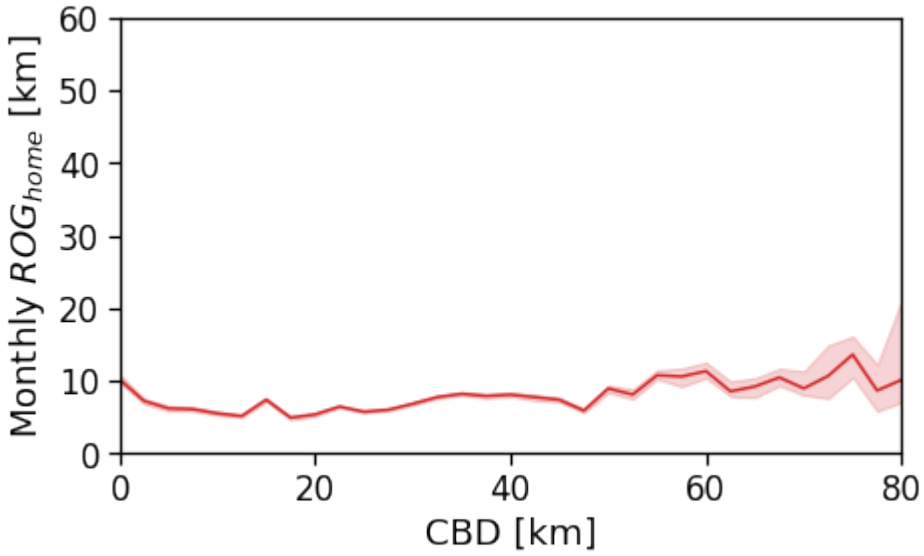
```
                                    .copy(deep=True)
fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='rog_total_mean',
            xlabel='CBD [km]',
            ylabel=r'Monthly $ROG_{total}$ [km]',
            xlim=[0,80],
            ylim=[0,18],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'rogTotal_vs_cbd_scatterLine.pdf'))
```



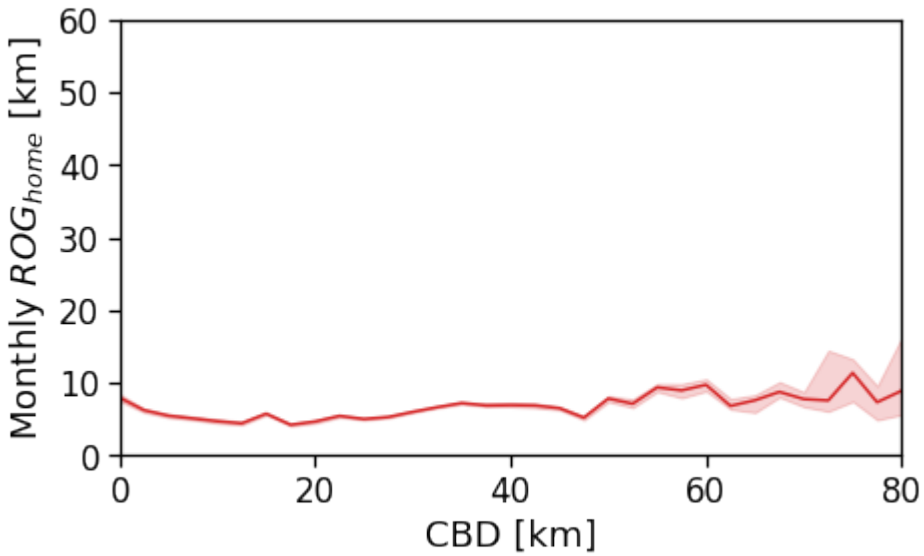
Now we show the per-user plots: for each user we use his home's distance from the CBD (and later from the BD closest to his workplace).

We see that in a policentric city as Mumbai there is no clear dependence of the ROG_{home} when we consider a single CBD.

[92]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='rog_home',
            xlabel='CBD [km]',
            ylabel=r'Monthly $ROG_{home}$ [km]',
            xlim=[0,80],
            ylim=[0,60],
            scatterkws={'alpha': .0},
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'rogHome_vs_cbd_scatterLine_users_cbd.pdf'))
```



[93]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='rog_total',
            xlabel='CBD [km]',
            ylabel=r'Monthly $ROG_{home}$ [km]',
            xlim=[0,80],
            ylim=[0,60],
            scatterkws={'alpha': .0},
    )
# plt.savefig(os.path.join(OUT_DIR_FIG, 'rogHome_vs_cbd_scatterLine_users_cbd.pdf'))
```



However, when we switch to the user's home distance from the reference BD the underlying spatial organization of the city emerges.

Here we see that the ROG (and later the same will be shown for the daily ROG and TTD) increases as we get further from the reference BD, meaning that the distance that a user needs to travel increases as he lives far from the hub of economic and work opportunities that a BD offers.

[94]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='rog_home',
            xlabel='BD closest to workplace [km]',
            ylabel=r'Monthly $ROG_{home}$ [km]',
            xlim=[0,80],
            ylim=[0,60],
            scatterkws={'alpha': .0},
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'rogHome_vs_cbd_scatterLine_users_cbd_closest.pdf
↪'))
```

Daily ROG

We compute the per-user-day average ROG.

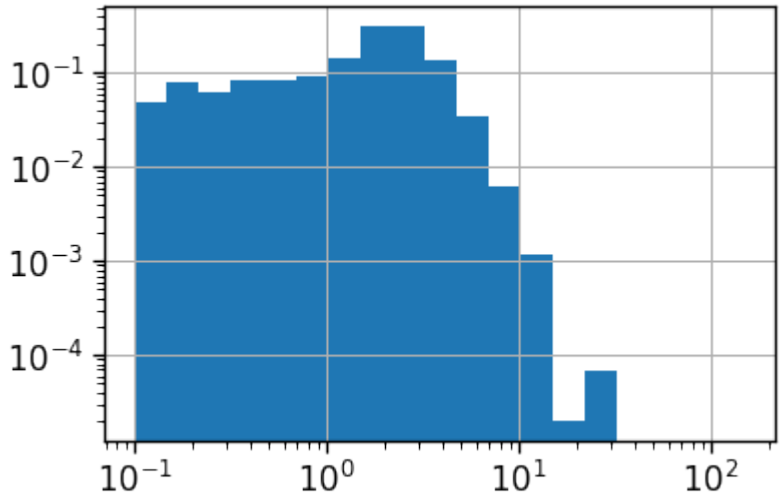
[95]:
```
for c in perAreaDailyROG.columns:
    if c in gdf_aoi_grid_landuse:
        del gdf_aoi_grid_landuse[c]

gdf_aoi_grid_landuse = pd.merge(gdf_aoi_grid_landuse,
                                perAreaDailyROG.reset_index(),
                                on=zidColName,
```

(continues on next page)

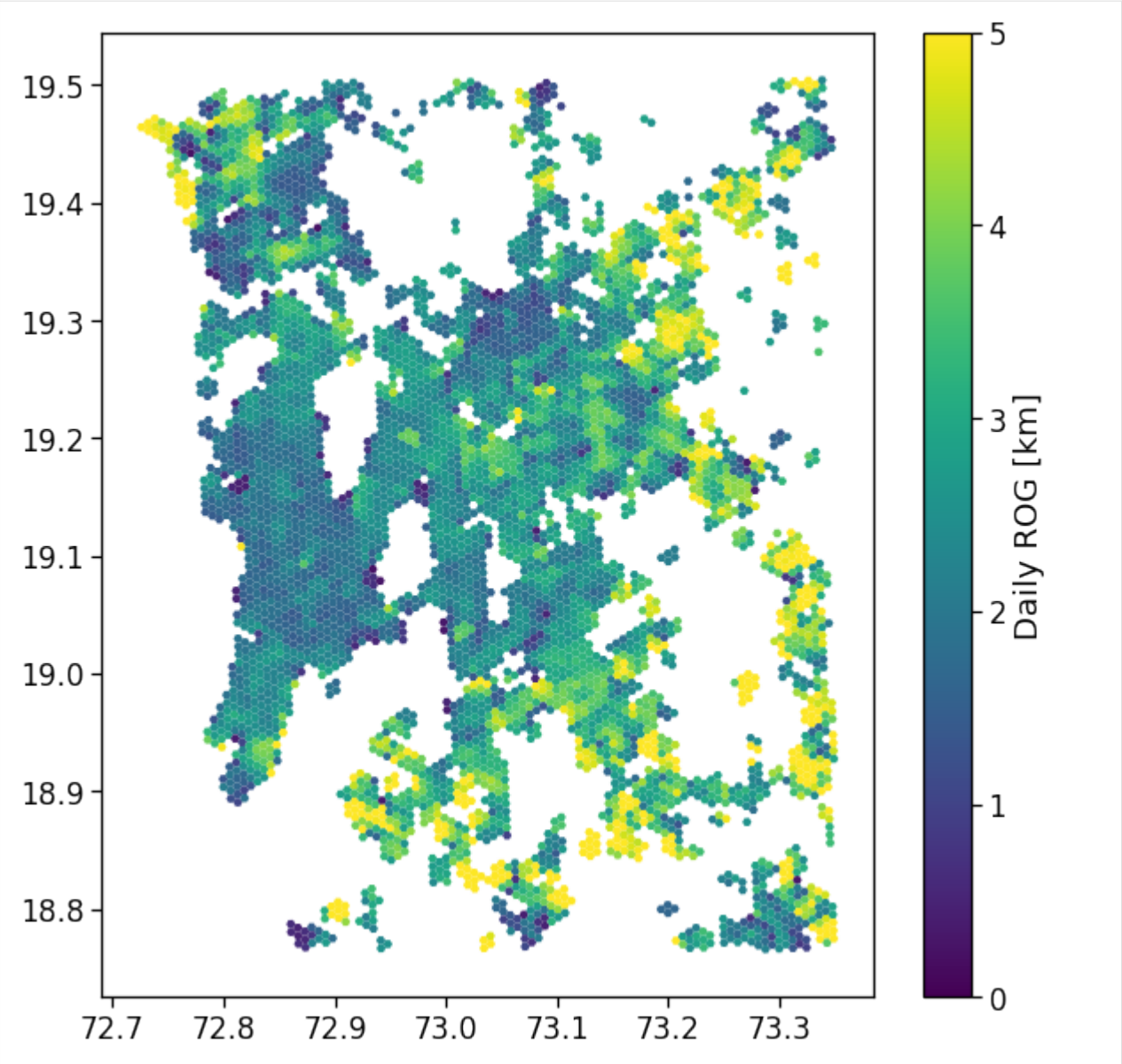
(continued from previous page)

```
                                    how='left')

for c in perAreaDailyROG.columns:
    gdf_aoi_grid_landuse[c].fillna(.0, inplace=True)
```

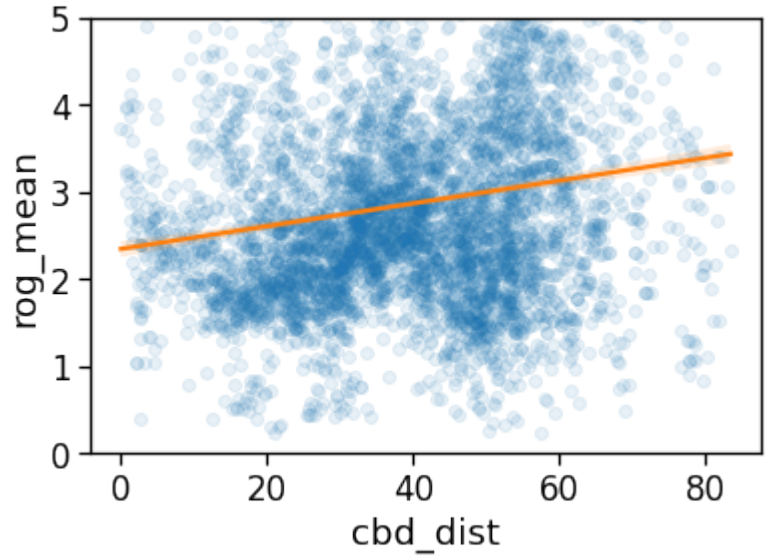
[96]:
```
gdf_aoi_grid_landuse['rog_mean'].hist(bins=np.geomspace(.1,150,20),
                                      density=True)
plt.loglog();
```



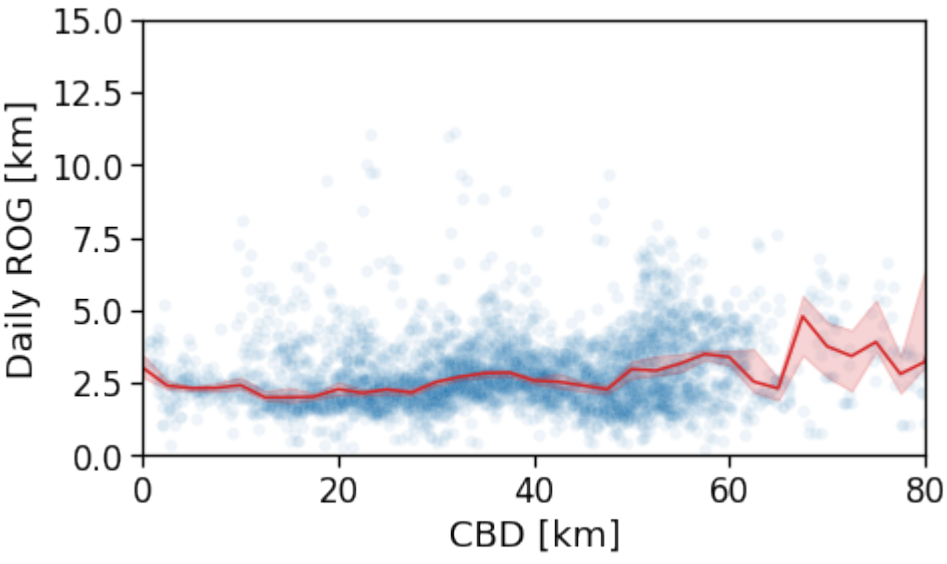
[97]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
gdf_aoi_grid_landuse.query('rog_mean>0 & rog_count > 100').plot('rog_mean',
                                                                vmin=.0, vmax=5.,
                           legend=True, ax=ax,
                           legend_kwds={'label': r'Daily ROG [km]'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'daily_rog_map.pdf'))
```



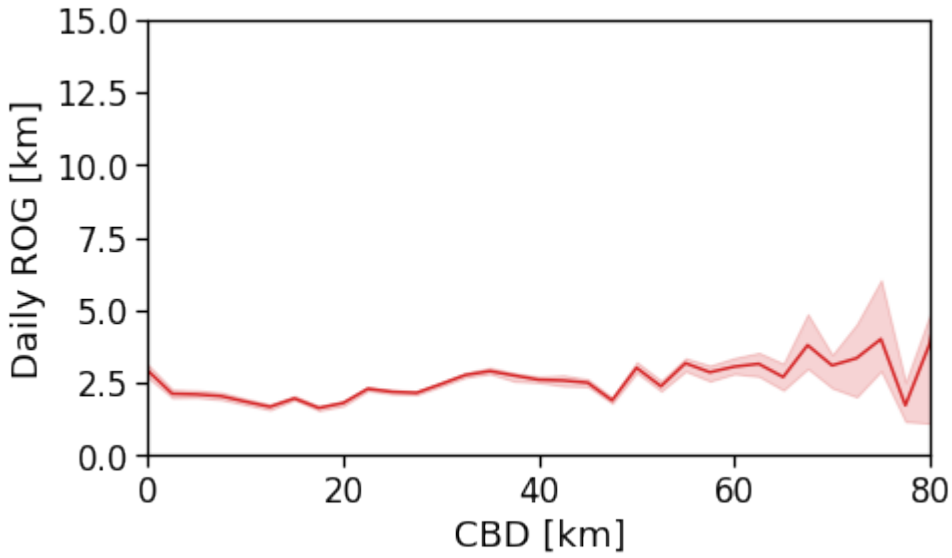
[98]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='rog_mean',
    data=gdf_aoi_grid_landuse.query('rog_mean>0 & rog_count > 100'),
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.1},
    order=1,
)
plt.ylim(0,5)
plt.savefig(os.path.join(OUT_DIR_FIG, 'daily_rog_vs_cbd.pdf'))
```



[99]:
```
tmp_data = gdf_aoi_grid_landuse.query('rog_mean>0 & rog_count > 200')\
                                .copy(deep=True)
fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='rog_mean',
            xlabel='CBD [km]',
            ylabel='Daily ROG [km]',
            xlim=[0,80],
            ylim=[0,15],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'daily_rog_vs_cbd_scatterLine.pdf'))
```



[100]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='rog',
            xlabel='CBD [km]',
            ylabel='Daily ROG [km]',
            xlim=[0,80],
            ylim=[0,15],
            doScatter=False,
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'daily_rog_vs_cbd_scatterLine_user_cbd.pdf'))
```



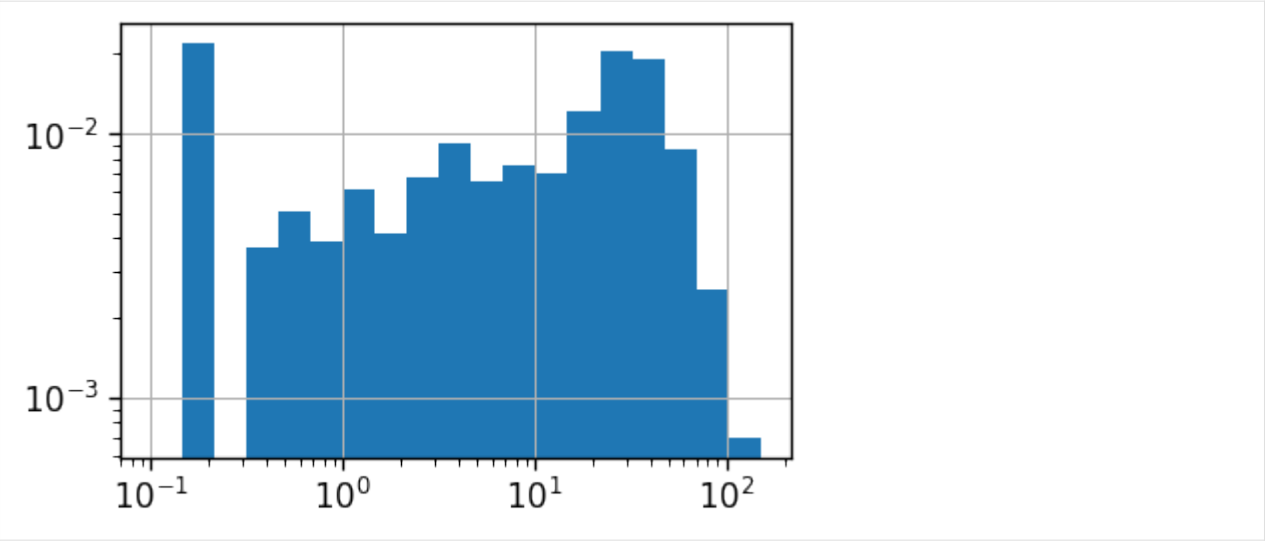
[101]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='rog',
            xlabel='BD closest to workplace [km]',
            ylabel='Daily ROG [km]',
            xlim=[0,80],
            ylim=[0,15],
            doScatter=False,
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'daily_rog_vs_cbd_scatterLine_user_cbd_closest.pdf
↪'))
```

Daily total traveled distance

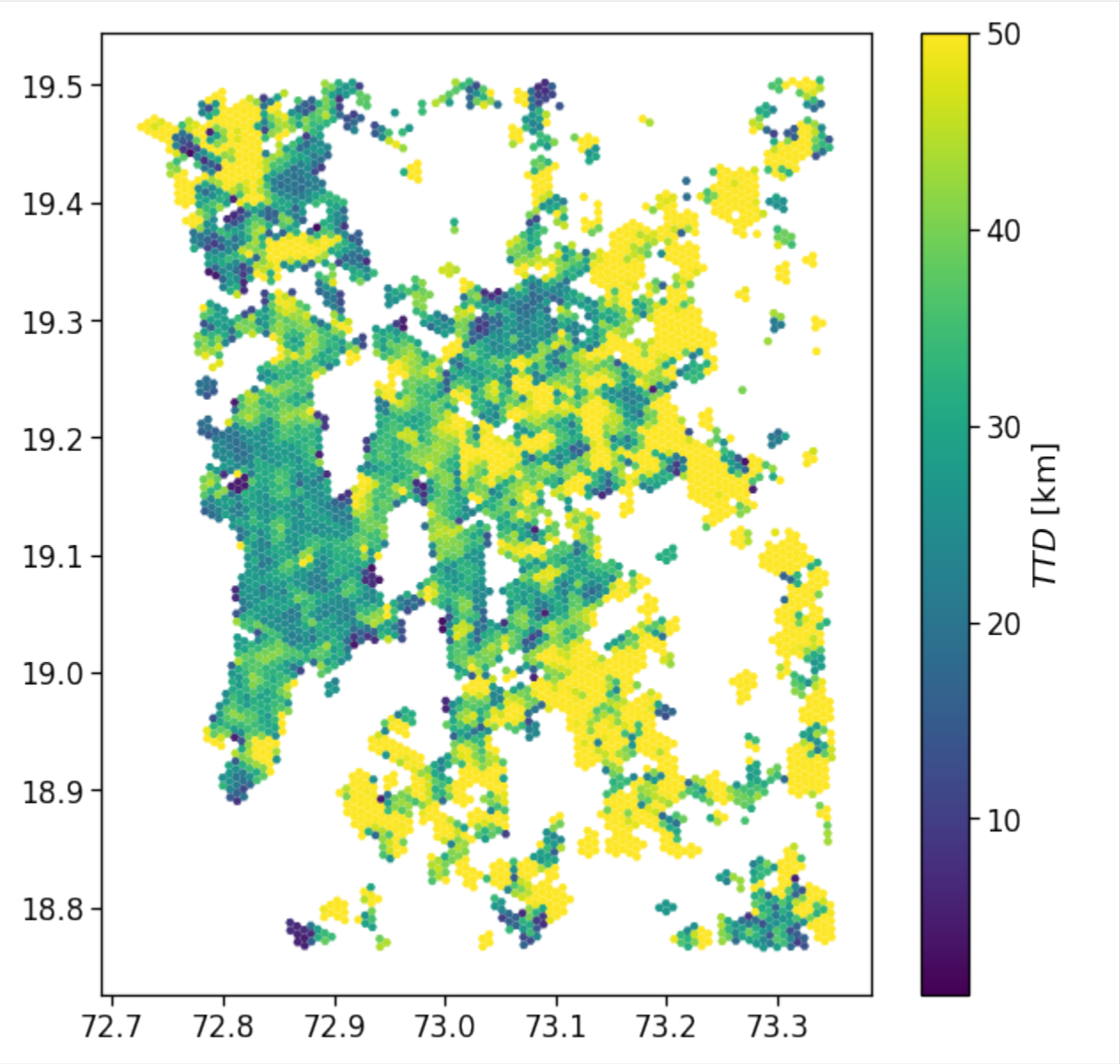
We compute the per-user-day total travel distance.

[102]:
```
for c in perAreaDailyTTD.columns:
    if c in gdf_aoi_grid_landuse:
        del gdf_aoi_grid_landuse[c]

gdf_aoi_grid_landuse = pd.merge(gdf_aoi_grid_landuse,
                                perAreaDailyTTD.reset_index(),
                                on=zidColName,
                                how='left')

for c in perAreaDailyTTD.columns:
    gdf_aoi_grid_landuse[c].fillna(.0, inplace=True)
```

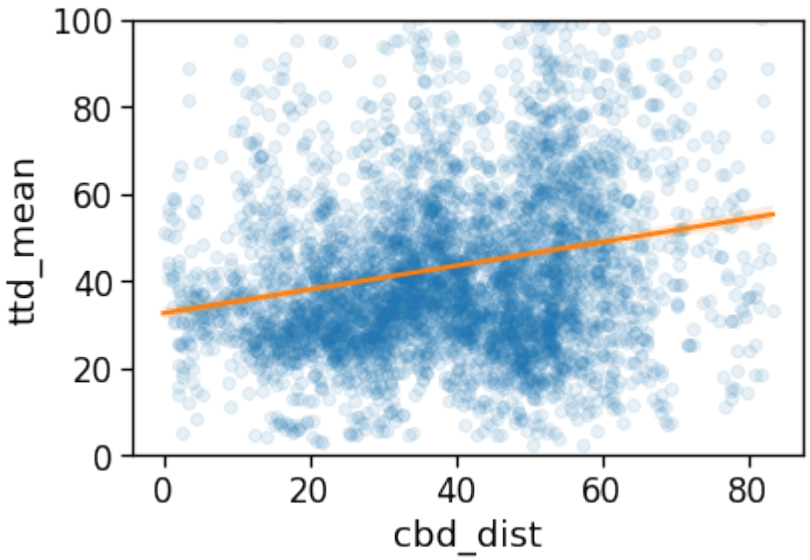
[103]:
```
gdf_aoi_grid_landuse['ttd_mean'].hist(bins=np.geomspace(.1,150,20),
                                      density=True)
plt.loglog();
```



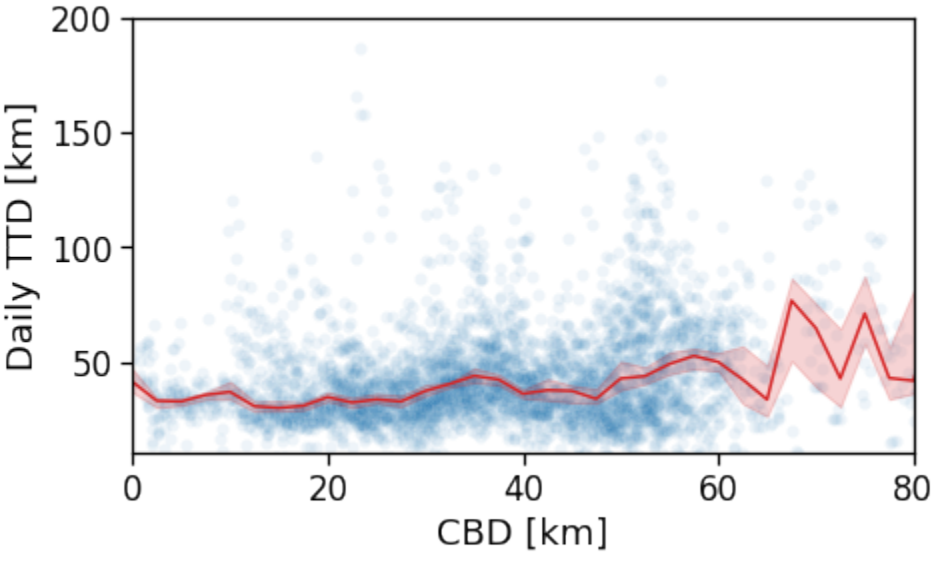
[104]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
gdf_aoi_grid_landuse.query('ttd_mean>0 & ttd_count > 100').plot('ttd_mean', vmin=1,␣
↪vmax=50,
                            legend=True, ax=ax,
                            legend_kwds={'label': r'$TTD$ [km]'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'ttd_map.pdf'))
```



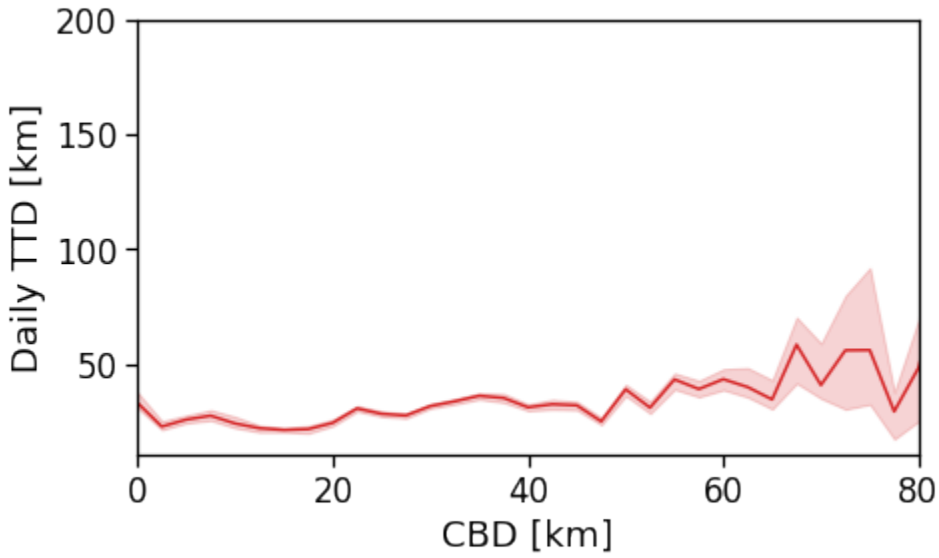
[105]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='ttd_mean',
    data=gdf_aoi_grid_landuse.query('ttd_mean>0 & ttd_count > 100'),
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.1},
    order=1,
)
plt.ylim(0,100)
plt.savefig(os.path.join(OUT_DIR_FIG, 'ttd_vs_cbd.pdf'))
```



[106]:
```
tmp_data = gdf_aoi_grid_landuse.query('ttd_mean>0 & ttd_count > 200')\
                                .copy(deep=True)
fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='ttd_mean',
            xlabel='CBD [km]',
            ylabel='Daily TTD [km]',
            xlim=[0,80],
            ylim=[10,200],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'ttd_vs_cbd_scatterLine.pdf'))
```



[107]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='ttd',
            xlabel='CBD [km]',
            ylabel='Daily TTD [km]',
            xlim=[0,80],
            ylim=[10,200],
            doScatter=False,
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'ttd_vs_cbd_scatterLine_user_cbd.pdf'))
```



[108]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='ttd',
            xlabel='BD closest to workplace [km]',
            ylabel='Daily TTD [km]',
            xlim=[0,80],
            ylim=[10,200],
            doScatter=False,
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'ttd_vs_cbd_scatterLine_user_cbd_closest.pdf'))
```

[109]:
```
gdf_aoi_grid_landuse.to_file(out_grid_cells_file, driver='GPKG')
```

Number of houses/offices

We: - project the user home/work locations to the local projection given by `local_EPSG`; - buffer their location by `homeWorkBufferMeters` meters to simulate a kernel density (with flat kernel over a circle); - count the number of offices and homes found in each cell and compute their absolute ratio;

Area vocation: ratio of of houses/offices

We simply compute the per-area a ratio R_a between the density of homes and offices.

To obtain an homogeneous indicator we compute the log_2 of this ratio, i.e.,

$$R_a = \log_2 \left(\frac{H_a}{W_a} \right),$$

where H_a, W_a are the number of homes and workplaces found in area a, respectively.

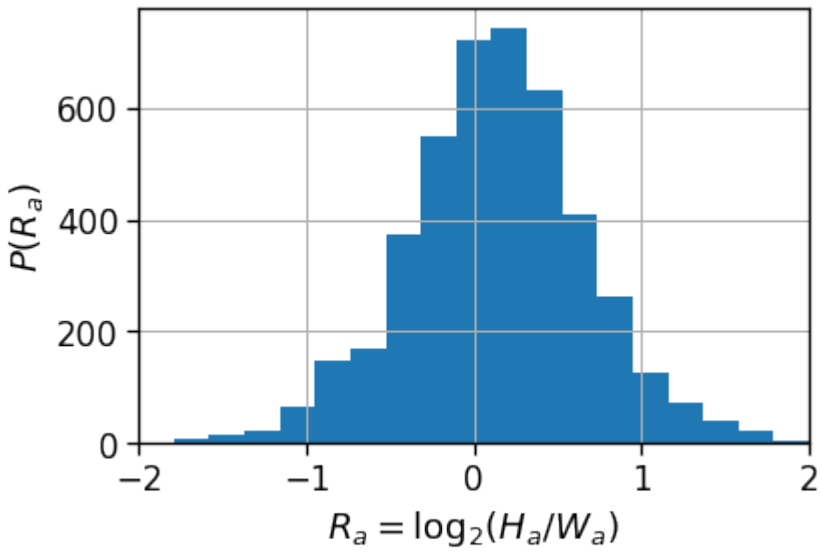
A negative value of R_a signals a predominance of workplaces (an area devoted to commercial and business activities), whereas a positive value indicates a more residental-based area. Values $R_a \sim 0$ highlight a balanced mixing of the two.

NOTE >When determining the home location of a user, please consider that some data providers, like Cuebiq, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy. > >This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern. > >However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

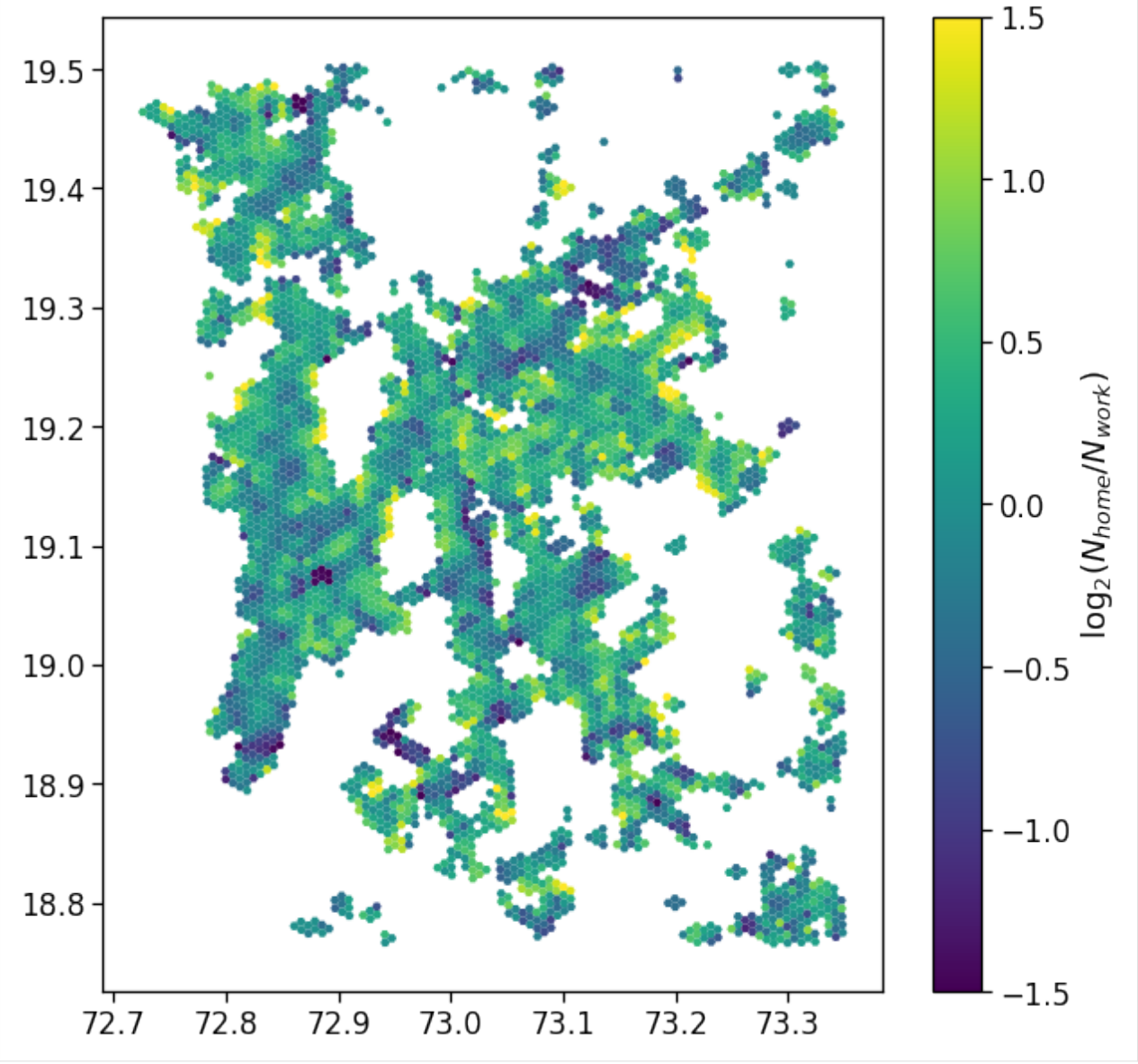
[110]:
```
gdf_aoi_grid_landuse['n_homes'] = gdf_aoi_grid_landuse[zidColName].apply(lambda z: home_
↪counts[z]
                                                                          if␣
↪z in home_counts else 0)
gdf_aoi_grid_landuse['n_works'] = gdf_aoi_grid_landuse[zidColName].apply(lambda z: work_
↪counts[z]
                                                                          if␣
↪z in work_counts else 0)

gdf_aoi_grid_landuse['home_work_ratio'] = np.log2(gdf_aoi_grid_landuse['n_homes'].
↪clip(lower=1)
                                                  / gdf_aoi_grid_landuse['n_works'].
↪clip(lower=1))

gdf_aoi_grid_landuse.loc[
    (gdf_aoi_grid_landuse['n_homes'] < min_pings_home_work)
    | (gdf_aoi_grid_landuse['n_works'] < min_pings_home_work), 'home_work_ratio'] = None
gdf_aoi_grid_landuse.head(0)
```

[110]:
```
Empty GeoDataFrame
Columns: [id, left, top, right, bottom, tile_ID, geometry, cluster, scope, nUsers, rog_
↪total_min, rog_total_max, rog_total_mean, rog_total_std, rog_total_count, rog_home_min,
↪ rog_home_max, rog_home_mean, rog_home_std, rog_home_count, cbd_dist, cbd_dist_bin,␣
↪rog_min, rog_max, rog_mean, rog_std, rog_count, ttd_min, ttd_max, ttd_mean, ttd_std,␣
↪ttd_count, n_homes, n_works, home_work_ratio]
Index: []

[0 rows x 35 columns]
```

[111]:
```
gdf_aoi_grid_landuse['home_work_ratio'].hist(bins=np.linspace(-2,2, 20))
plt.xlabel(r'$R_a = \log_2\left(H_a / W_a\right)$')
plt.ylabel(r'$P(R_a)$')
plt.xlim(-2,2)
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_ratio_hist.pdf'))
```



[112]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
gdf_aoi_grid_landuse.plot('home_work_ratio', vmin=-1.5, vmax=1.5,
                          legend=True, ax=ax,
                          legend_kwds={'label': r'$\log_{2}(N_{home}/N_{work})$'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_ratio_map.pdf'))
```

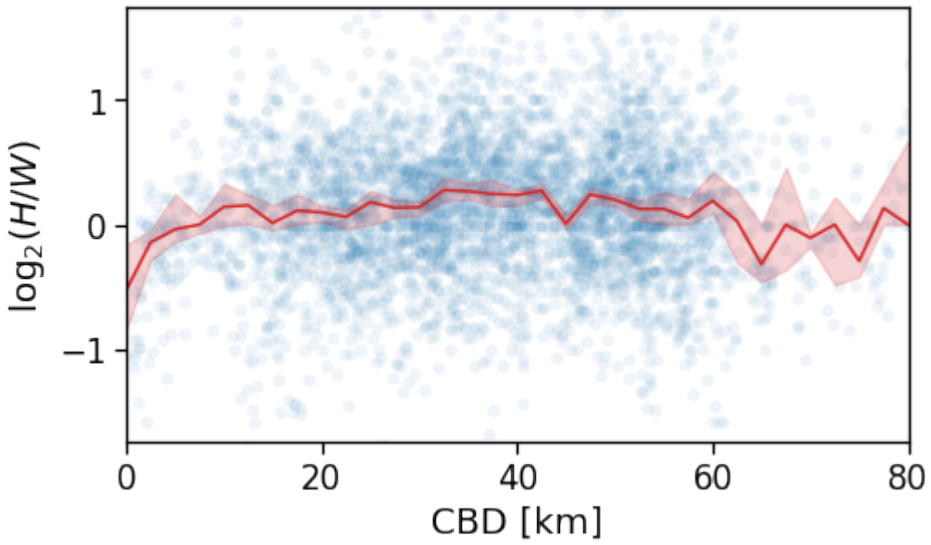


[113]:
```
# Check the indicator dependence on the distance from main CBD
tmp_data = gdf_aoi_grid_landuse

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='home_work_ratio',
            xlabel='CBD [km]',
```

(continues on next page)

(continued from previous page)

```
            ylabel=r'$\log_2(H/W)$',
            xlim=[0,80],
            ylim=[-1.75,1.75],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_ratio_scatterLine.pdf'))
```



[114]:
```
gdf_aoi_grid_landuse.to_file(out_grid_cells_file, driver='GPKG')
```

Length and duration of commute

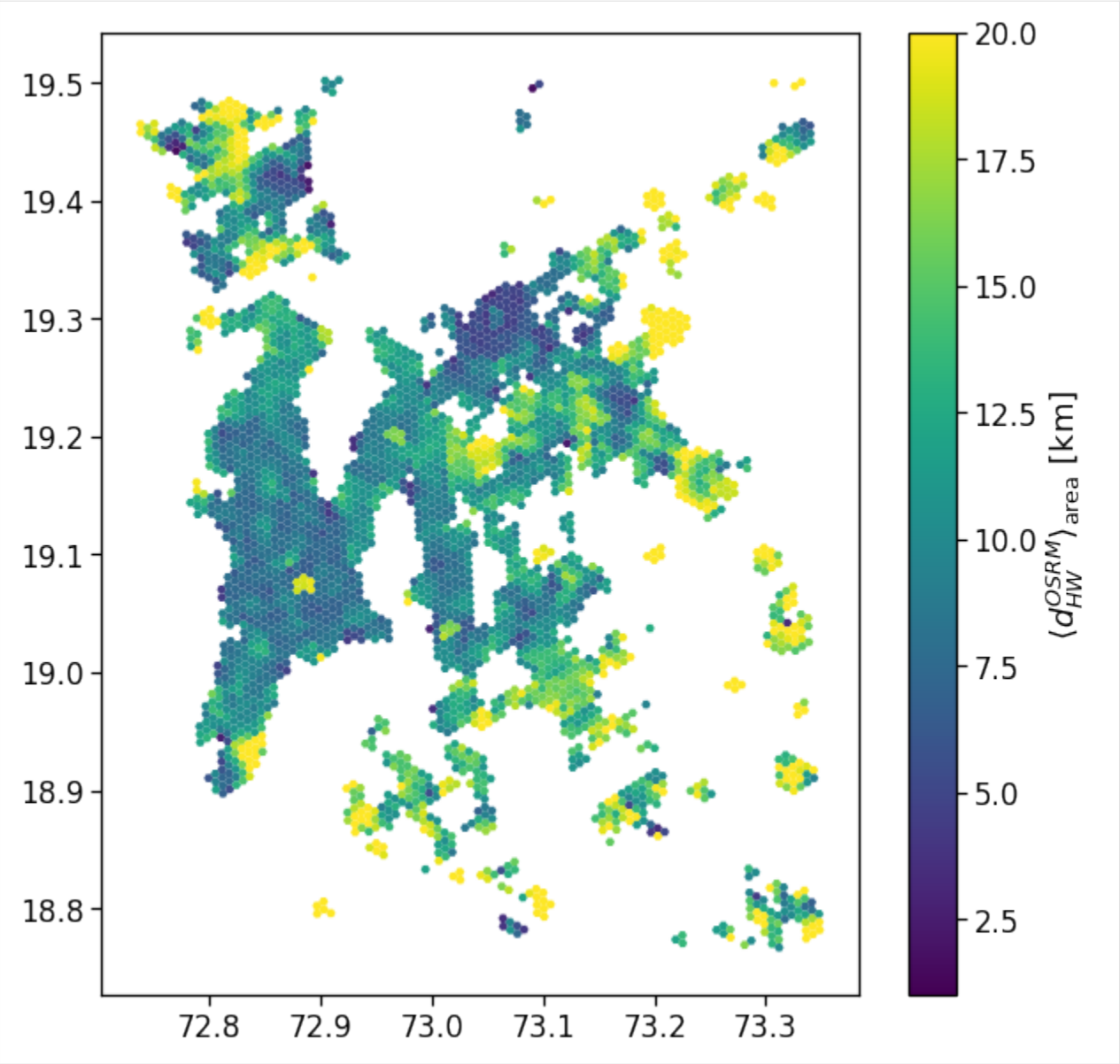
[116]:
```
gdf_aoi_grid_landuse_trip_stats = pd.merge(gdf_aoi_grid_landuse,
                                time_trips_stats_df[[c for c in time_trips_stats_df.
↪columns if c != uidColName]].reset_index(),
                                on=zidColName, how='outer')

for c in time_trips_stats_df.columns:
    if c != uidColName:
        gdf_aoi_grid_landuse_trip_stats[c] = gdf_aoi_grid_landuse_trip_stats[c].
↪astype(float)

gdf_aoi_grid_landuse_trip_stats.to_file(out_grid_cells_file, driver='GPKG')
```

Avg. commute real OSRM distance

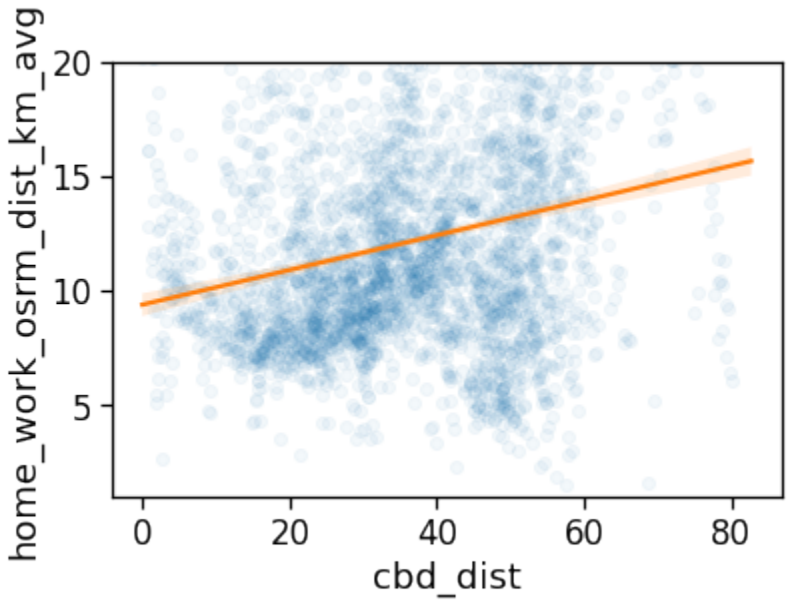
We have to transform the output of the OSRM backend from seconds/meters to hours/km to compare them to the measures produced by `mobilkit`.

[117]:
```
gdf_aoi_grid_landuse_trip_stats['home_work_osrm_dist_km_avg'] = gdf_aoi_grid_landuse_
↪trip_stats['home_work_osrm_dist_avg'] / 1000.
gdf_aoi_grid_landuse_trip_stats['home_work_osrm_time_h_avg']  = gdf_aoi_grid_landuse_
↪trip_stats['home_work_osrm_time_avg'] / 3600.

user_stats_table_df['home_work_osrm_dist_km'] = user_stats_table_df['home_work_osrm_dist
↪'] / 1000.
user_stats_table_df['home_work_osrm_time_h']  = user_stats_table_df['home_work_osrm_time
↪'] / 3600.

cleaned_user_stats_table_df['home_work_osrm_dist_km'] = cleaned_user_stats_table_df[
↪'home_work_osrm_dist'] / 1000.
cleaned_user_stats_table_df['home_work_osrm_time_h']  = cleaned_user_stats_table_df[
↪'home_work_osrm_time'] / 3600.

tmp_df_stat = gdf_aoi_grid_landuse_trip_stats.query('n_homes > 15 | n_works > 15')
```

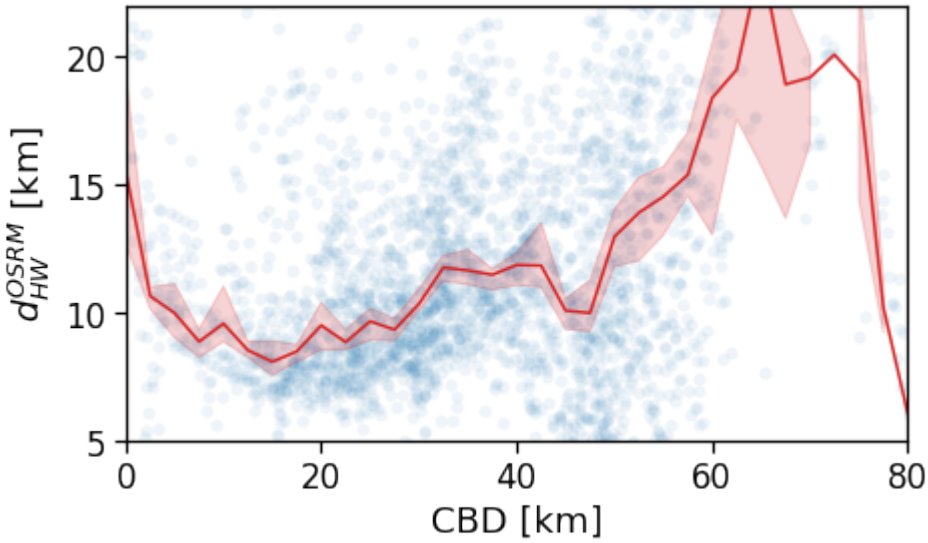
[118]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
tmp_df_stat.plot('home_work_osrm_dist_km_avg',
                                    vmin=1., vmax=20,
                                    ax=ax, legend=True,
                                    legend_kwds={'label': r'$\langle d^{OSRM}_{HW}\
↪rangle_{\rm area}$ [km]'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_osrm_map.pdf'))
```



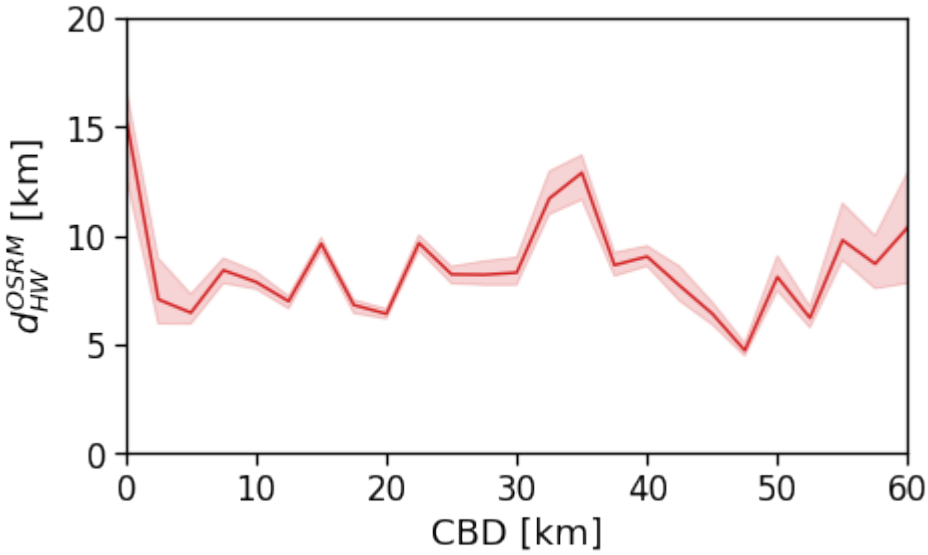
[119]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='home_work_osrm_dist_km_avg',
    data=tmp_df_stat,
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.05},
    order=1,
)
plt.ylim(1.,20)
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_osrm_scatter.pdf'), bbox_inches=
↪'tight')
```



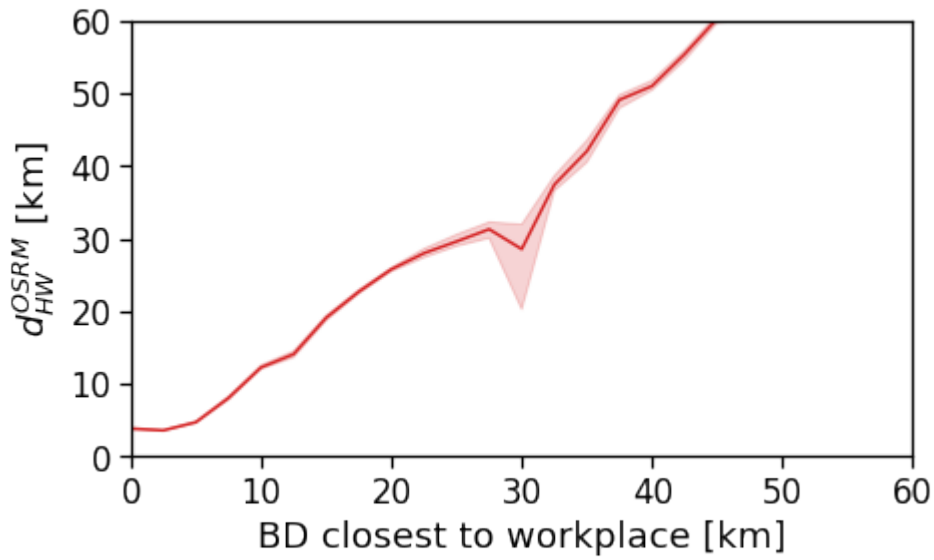
[120]:
```
tmp_data = gdf_aoi_grid_landuse_trip_stats.query('rog_home_count > 20')\
                                .copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='home_work_osrm_dist_km_avg',
            xlabel='CBD [km]',
            ylabel=r'$d^{OSRM}_{HW}$ [km]',
            xlim=[0,80],
            ylim=[5,22],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_osrm_scatterLine.pdf'), bbox_
↪inches='tight')
```



[121]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='home_work_osrm_dist_km',
            xlabel='CBD [km]',
            ylabel=r'$d^{OSRM}_{HW}$ [km]',
            xlim=[0,60],
            ylim=[0,20],
            scatterkws={'alpha':0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_osrm_scatterLine_user_cbd.pdf'),␣
↪bbox_inches='tight')
```

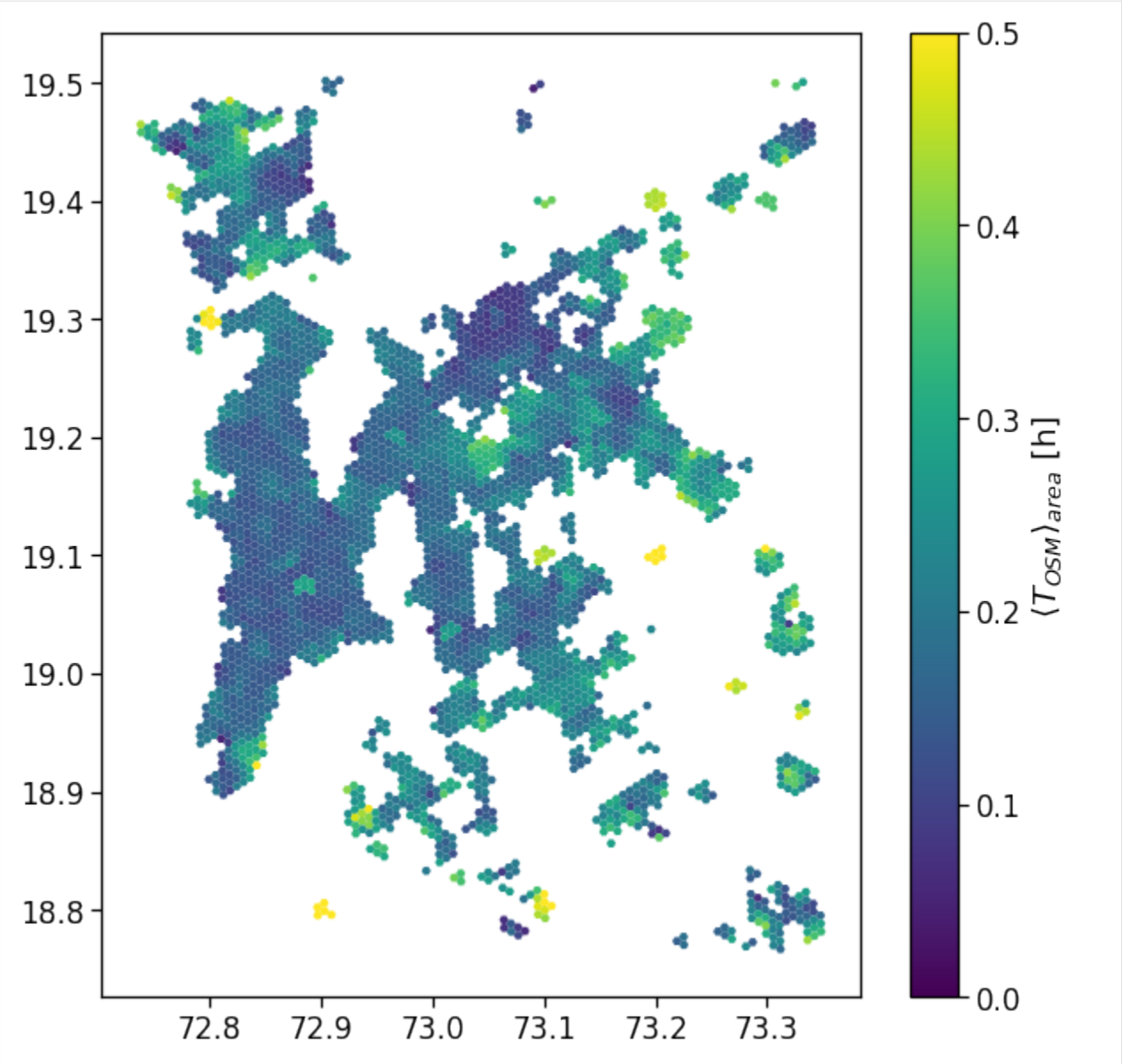


[122]:
```
# Check the indicator dependence on the distance from the user's home to closest BD
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='home_work_osrm_dist_km',
            xlabel='BD closest to workplace [km]',
            ylabel=r'$d^{OSRM}_{HW}$ [km]',
            xlim=[0,60],
            ylim=[0,60],
            scatterkws={'alpha':0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_osrm_scatterLine_user_cbd_closest.
↪pdf'), bbox_inches='tight')
```

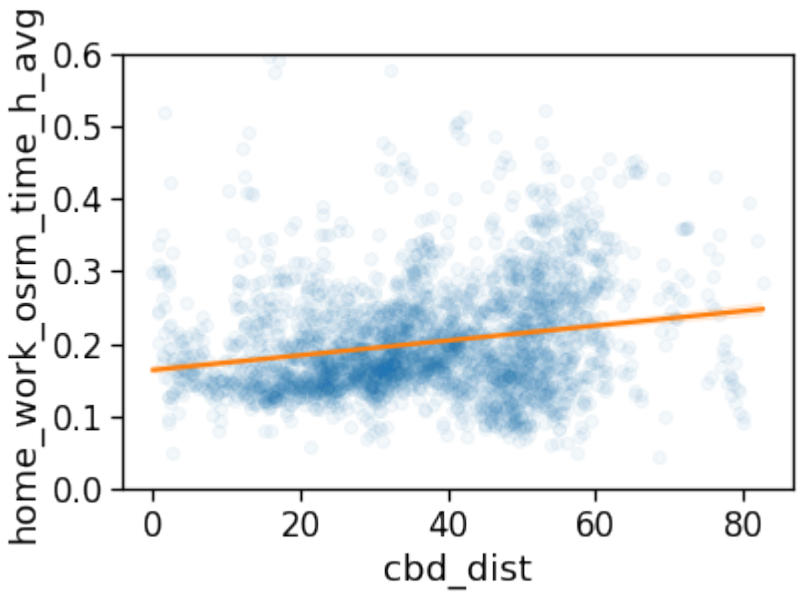


OSRM commute duration

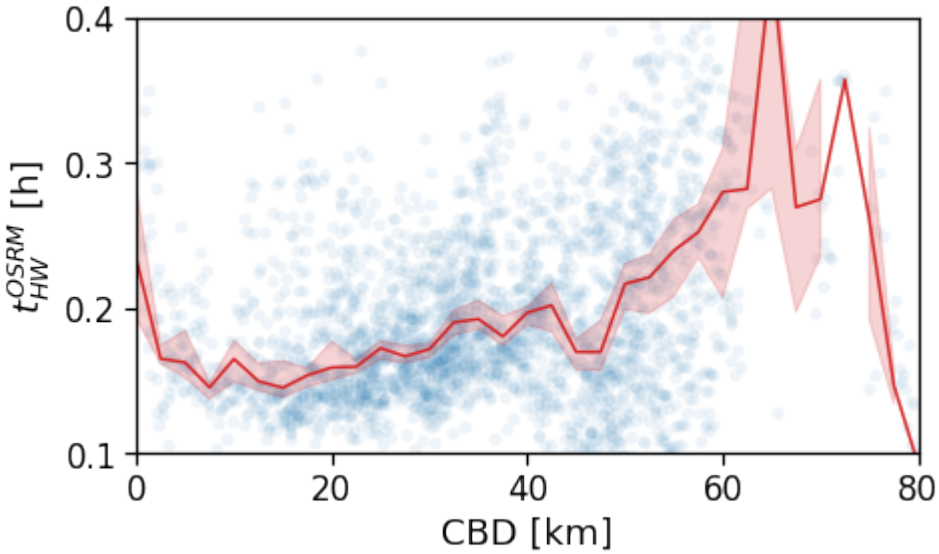
[123]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
tmp_df_stat.plot('home_work_osrm_time_h_avg',
                                    vmin=.0, vmax=.5,
                                    ax=ax, legend=True,
                                    legend_kwds={'label': r'$\langle T_{OSM}\rangle_
↪{area}$ [h]'}
                                   )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_osrm_map.pdf'), bbox_inches='tight
↪')
```



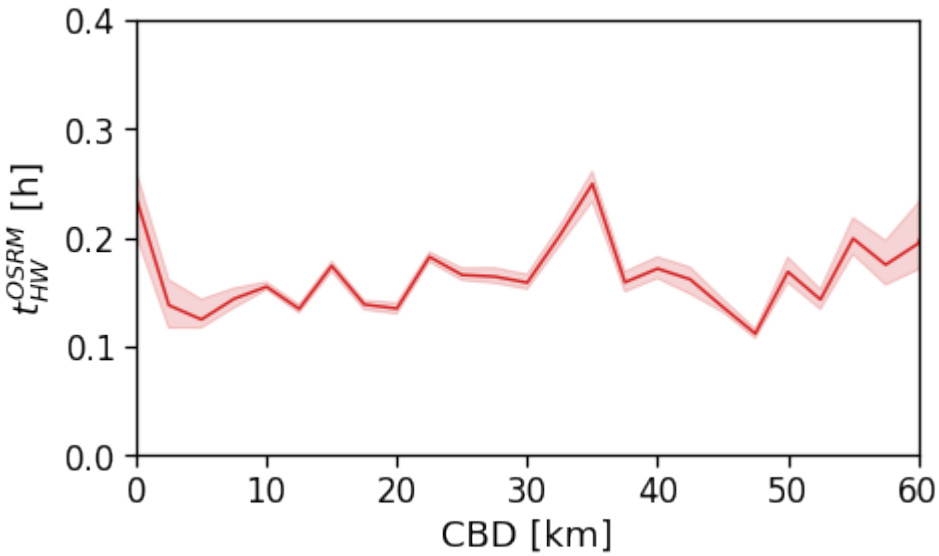
[124]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='home_work_osrm_time_h_avg',
    data=tmp_df_stat,
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.05},
    order=1,
)
plt.ylim(0,.6)
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_osrm_scatter.pdf'), bbox_inches=
↪'tight')
```



[125]:
```
tmp_data = gdf_aoi_grid_landuse_trip_stats.query('rog_home_count > 20')\
                                .copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='home_work_osrm_time_h_avg',
            xlabel='CBD [km]',
            ylabel=r'$t^{OSRM}_{HW}$ [h]',
            xlim=[0,80],
            ylim=[.1,.4],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_osrm_scatterLine.pdf'), bbox_
↪inches='tight')
```



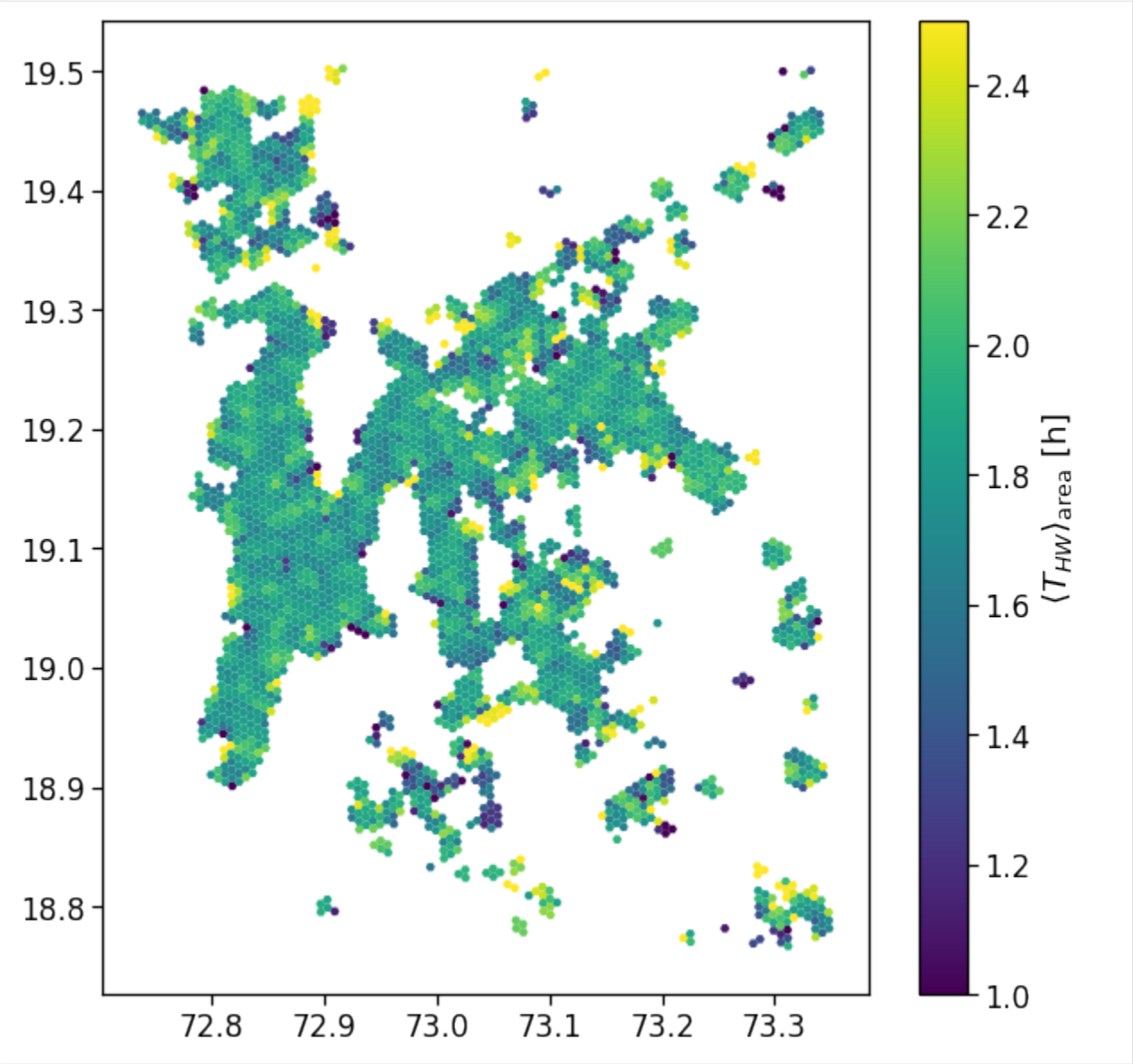
[126]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='home_work_osrm_time_h',
            xlabel='CBD [km]',
            ylabel=r'$t^{OSRM}_{HW}$ [h]',
            xlim=[0,60],
            ylim=[.0,.4],
            scatterkws={'alpha':0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_osrm_scatterLine_user_cbd.pdf'),␣
↪bbox_inches='tight')
```



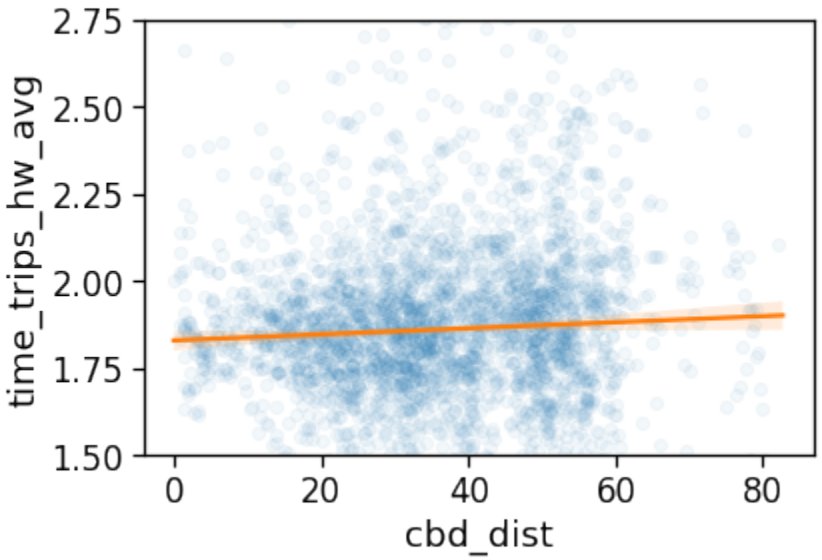
[127]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='home_work_osrm_time_h',
            xlabel='BD closest to workplace [km]',
            ylabel=r'$t^{OSRM}_{HW}$ [h]',
            xlim=[0,60],
            ylim=[.0,1.5],
            scatterkws={'alpha':0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_osrm_scatterLine_user_cbd_closest.
↪pdf'), bbox_inches='tight')
```

Avg. commute real duration

[128]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
tmp_df_stat.plot('time_trips_hw_avg',
                                    vmin=1., vmax=2.5,
                                    ax=ax, legend=True,
                                    legend_kwds={'label': r'$\langle T_{HW}\rangle_{\rm␣
↪area}$ [h]'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_map.pdf'), bbox_inches='tight
↪')
```



[129]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='time_trips_hw_avg',
    data=tmp_df_stat,
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.05},
    order=1,
)
plt.ylim(1.5,2.75)
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_scatter.pdf'), bbox_inches=
↪'tight')
```



[130]:
```
# Linear interpolation of rog vs distance...
x_col = 'cbd_dist'
y_col = 'time_trips_hw_avg'

tmp_data = tmp_df_stat[[x_col, y_col]].dropna()

model = sm.GLS(tmp_data[y_col], tmp_data[x_col])
res = model.fit()
res.summary()
```

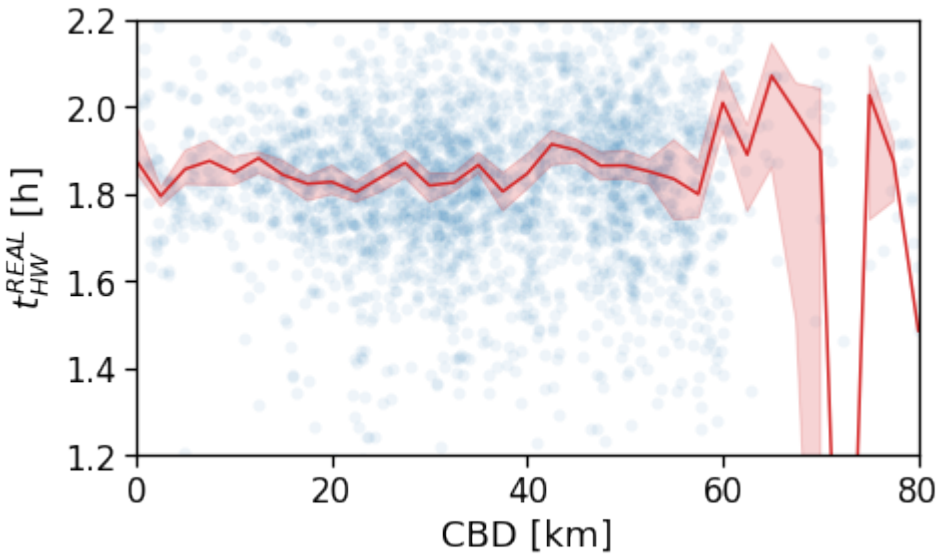
[130]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                                 GLS Regression Results
==============================================================================================
Dep. Variable:      time_trips_hw_avg   R-squared (uncentered):                   0.827
Model:                            GLS   Adj. R-squared (uncentered):              0.827
Method:                 Least Squares   F-statistic:                          1.558e+04
Date:                Mon, 16 May 2022   Prob (F-statistic):                        0.00
Time:                        15:53:33   Log-Likelihood:                         -3828.6
No. Observations:                3256   AIC:                                      7659.
Df Residuals:                    3255   BIC:                                      7665.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
cbd_dist       0.0435      0.000    124.812      0.000       0.043       0.044
==============================================================================
Omnibus:                       23.276   Durbin-Watson:                   0.335
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               30.370
Skew:                          -0.101   Prob(JB):                     2.54e-07
Kurtosis:                       3.427   Cond. No.                         1.00
==============================================================================
```

(continues on next page)

(continued from previous page)

```
Notes:
[1] R² is computed without centering (uncentered) since the model does not contain a␣
↪constant.
[2] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
"""
```

[131]:
```
tmp_data = gdf_aoi_grid_landuse_trip_stats.query('rog_home_count > 20')\
                                .copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='time_trips_hw_avg',
            xlabel='CBD [km]',
            ylabel=r'$t^{REAL}_{HW}$ [h]',
            xlim=[0,80],
            ylim=[1.2,2.2],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_scatterLine.pdf'), bbox_
↪inches='tight')
```

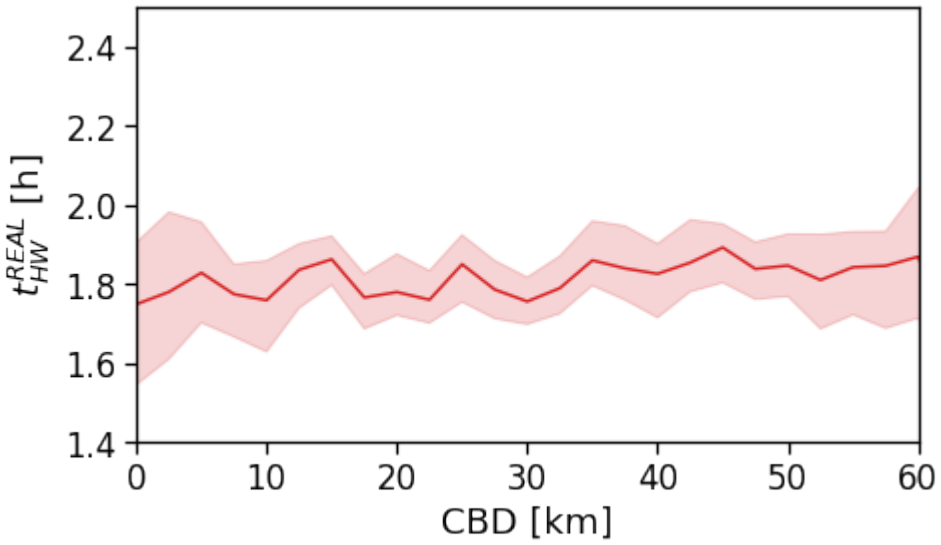


[132]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='time_trips_hw',
            xlabel='CBD [km]',
            ylabel=r'$t^{REAL}_{HW}$ [h]',
            xlim=[0,60],
```

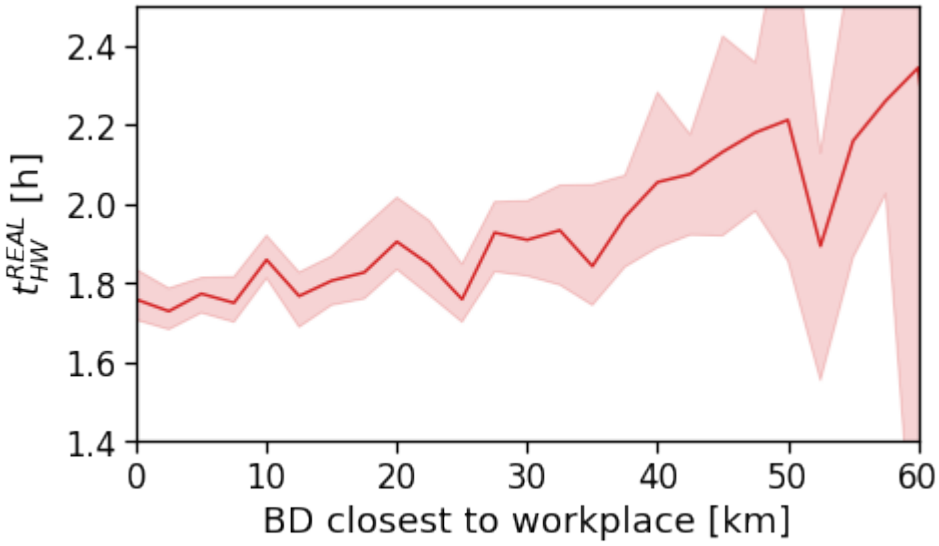
(continues on next page)

(continued from previous page)

```
            ylim=[1.4,2.5],
            scatterkws={'alpha':0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_scatterLine_user_cbd.pdf'),␣
↪bbox_inches='tight')
```

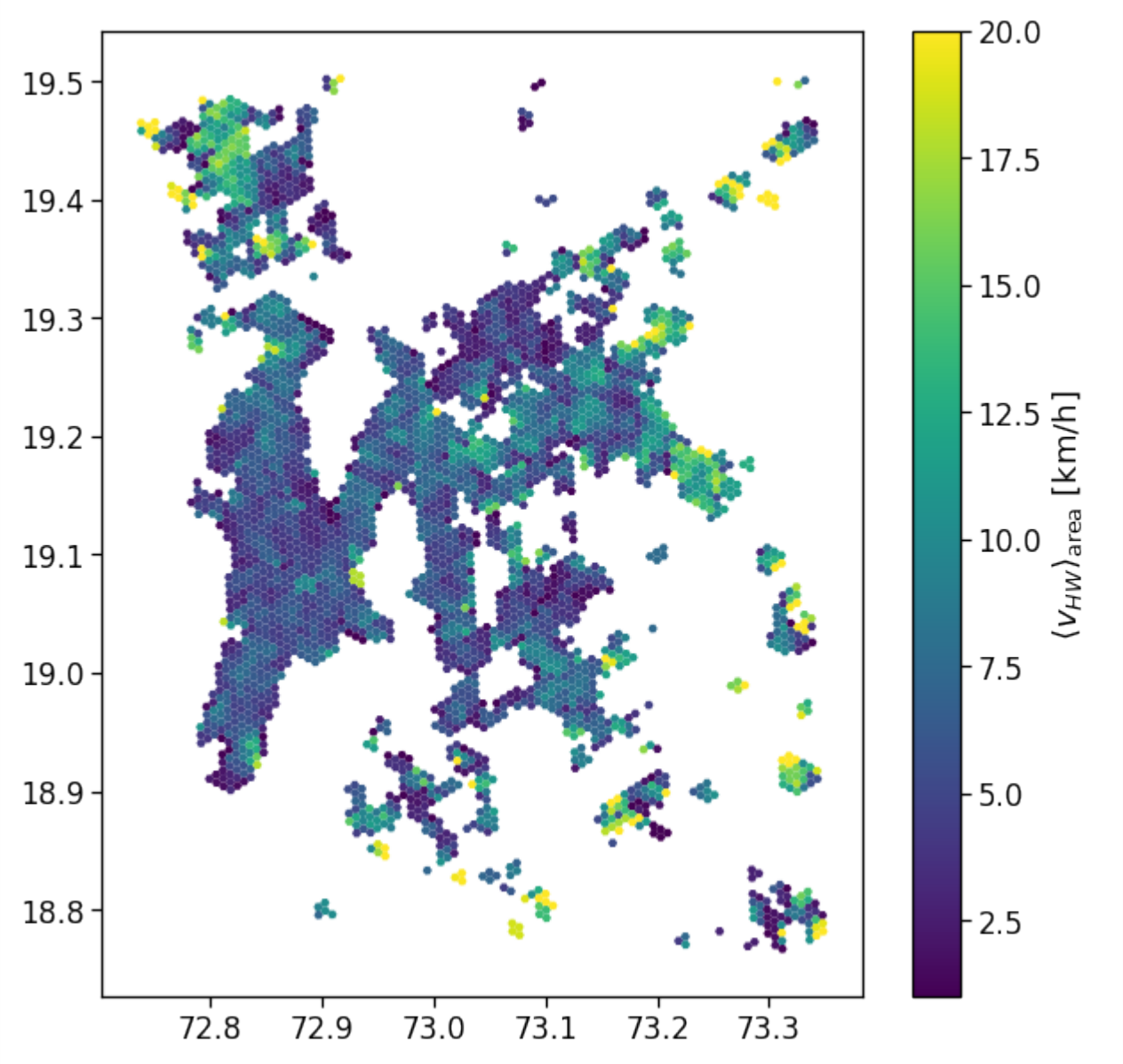


[133]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='time_trips_hw',
            xlabel='BD closest to workplace [km]',
            ylabel=r'$t^{REAL}_{HW}$ [h]',
            xlim=[0,60],
            ylim=[1.4,2.5],
            scatterkws={'alpha':0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_scatterLine_user_cbd_closest.
↪pdf'), bbox_inches='tight')
```



Avg. commute real speed

[134]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
tmp_df_stat.plot('speed_trips_hw_avg',
                                        vmin=1., vmax=20,
                                        ax=ax, legend=True,
                                        legend_kwds={'label': r'$\langle v_{HW}\rangle_{\rm␣
↪area}$ [km/h]'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_speed_real_map.pdf'), bbox_inches='tight
↪')
```



[135]:
```
tmp_df_stat.speed_trips_hw_avg.describe()
```

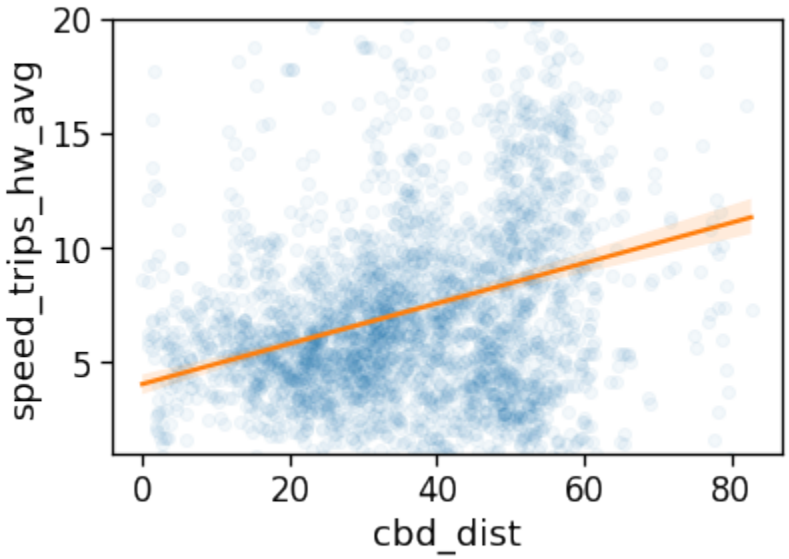
[135]:
```
count    3256.000000
mean        7.236964
std         4.947375
min         0.062201
25%         4.257395
50%         6.173521
75%         8.896125
max        86.212988
Name: speed_trips_hw_avg, dtype: float64
```

[136]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
```

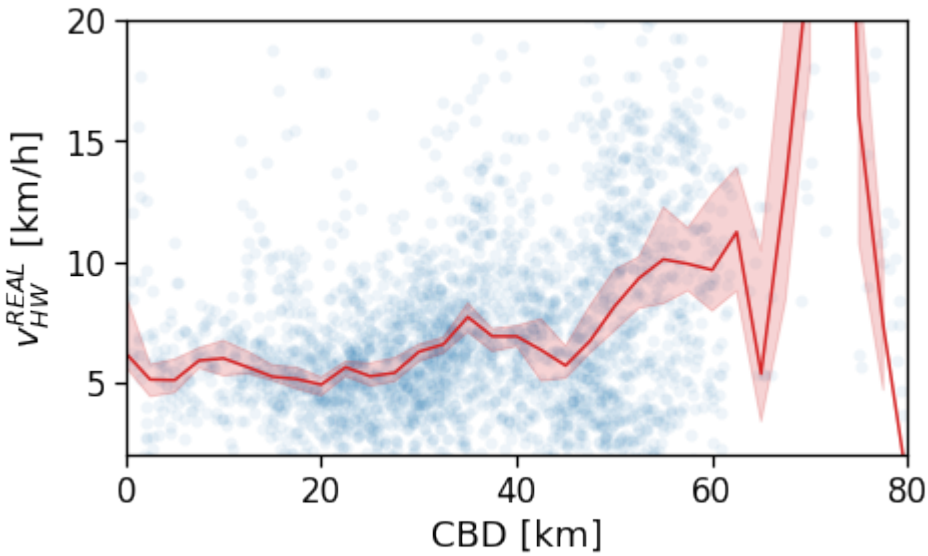
(continues on next page)

(continued from previous page)

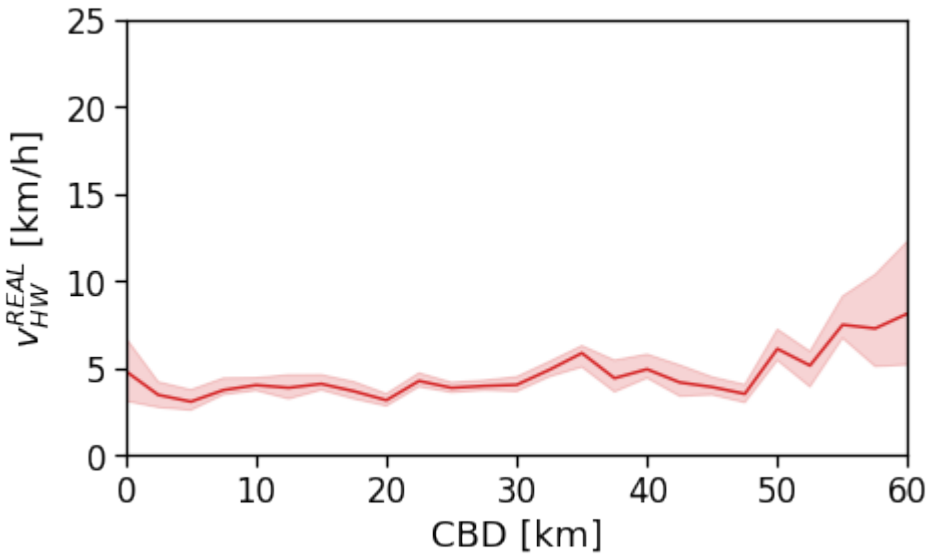
```
    x='cbd_dist',
    y='speed_trips_hw_avg',
    data=tmp_df_stat,
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.05},
    order=1,
)
plt.ylim(1.,20)
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_speed_real_scatter.pdf'), bbox_inches=
↪'tight')
```



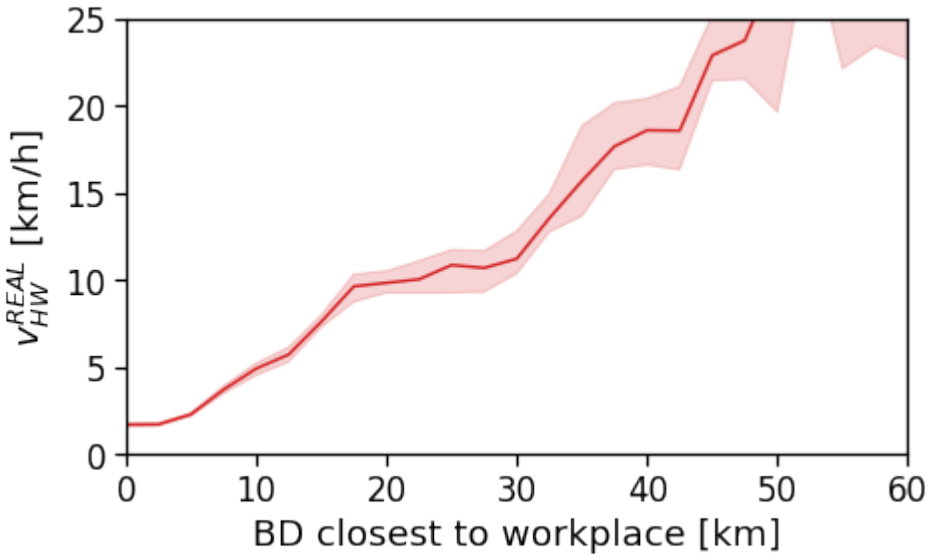
[137]:
```
tmp_data = gdf_aoi_grid_landuse_trip_stats.query('rog_home_count > 20')\
                                .copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='speed_trips_hw_avg',
            xlabel='CBD [km]',
            ylabel=r'$v^{REAL}_{HW}$ [km/h]',
            xlim=[0,80],
            ylim=[2,20],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_speed_real_scatterLine.pdf'), bbox_
↪inches='tight')
```



[138]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='speed_trips_hw',
            xlabel='CBD [km]',
            ylabel=r'$v^{REAL}_{HW}$ [km/h]',
            xlim=[0,60],
            ylim=[0,25],
            scatterkws={'alpha': 0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_speed_real_scatterLine_user_cbd.pdf'),␣
↪bbox_inches='tight')
```



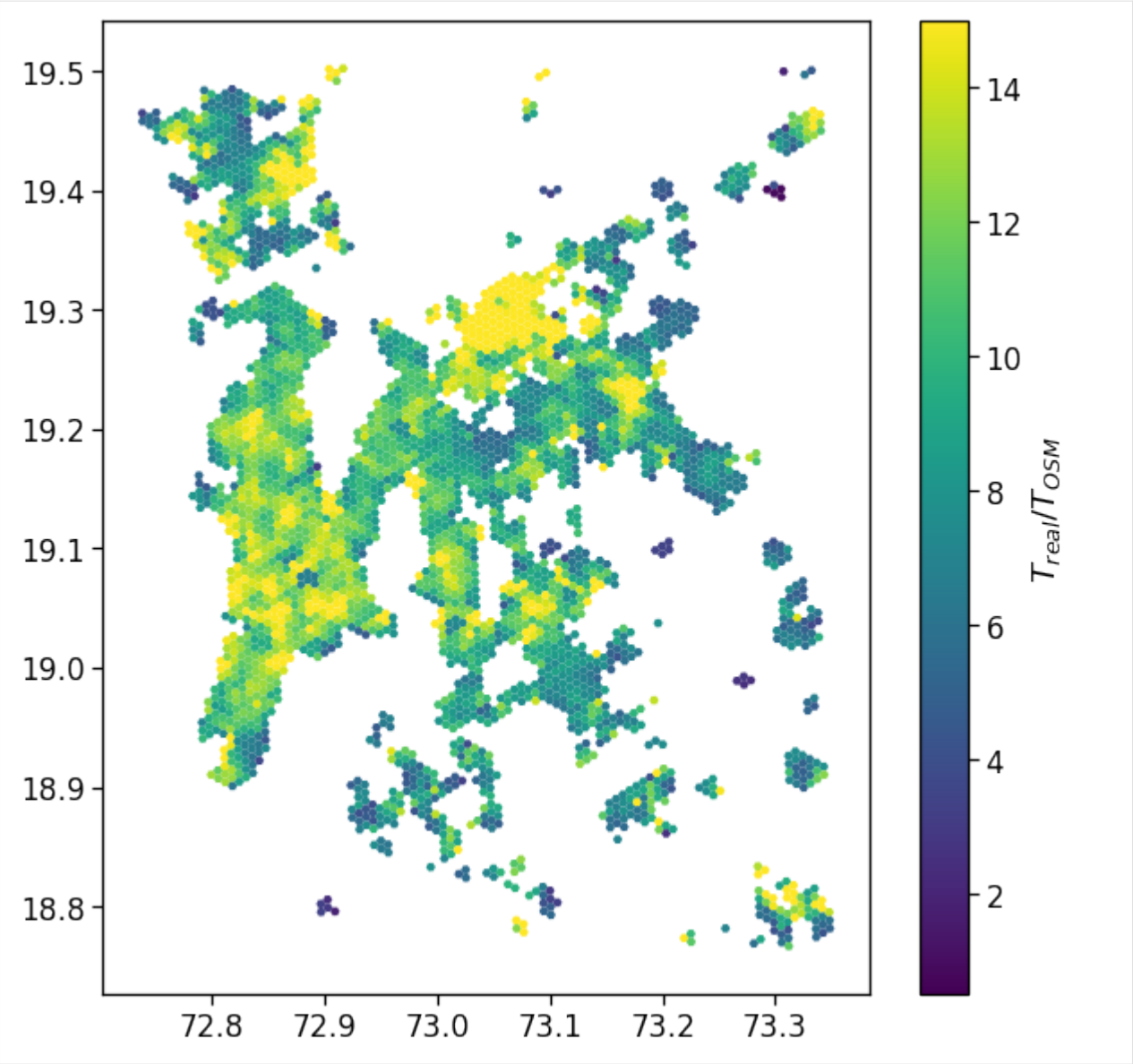
[139]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='speed_trips_hw',
            xlabel='BD closest to workplace [km]',
            ylabel=r'$v^{REAL}_{HW}$ [km/h]',
            xlim=[0,60],
            ylim=[0,25],
            scatterkws={'alpha': 0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_speed_real_scatterLine_user_cbd_closest.
↪pdf'), bbox_inches='tight')
```



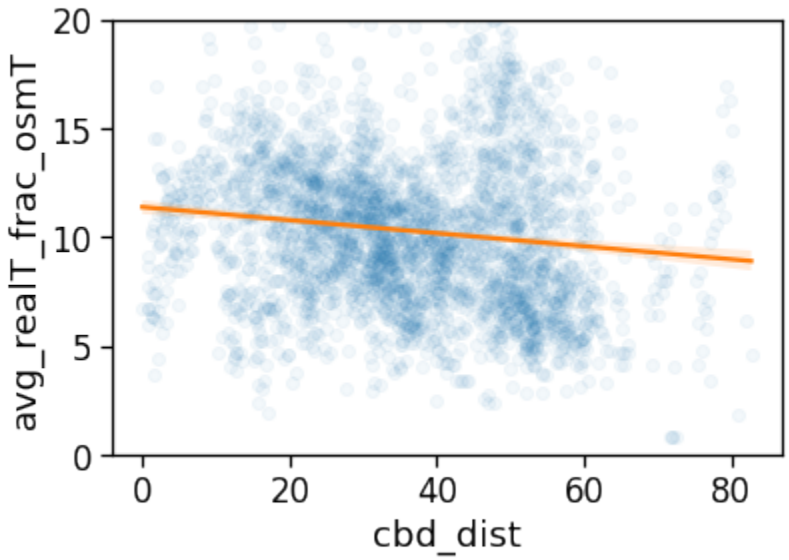
Real commuting duration / traffic-free OSRM driving duration

This is an indicator of congestions and/or lack of street infrastructure performance, as a ratio >> 1 indicates longer commuting times.

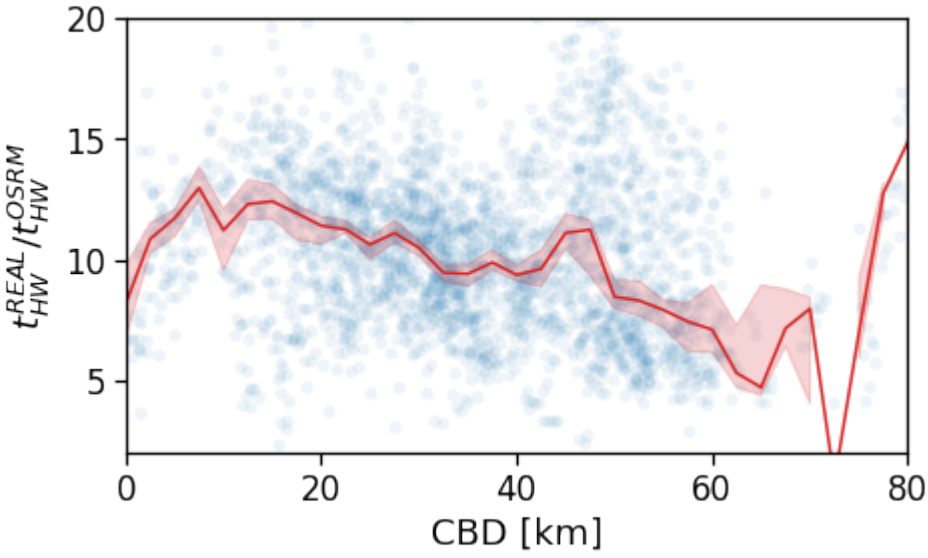
[140]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
tmp_df_stat.plot('avg_realT_frac_osmT',
                                    vmin=.5, vmax=15,
                                    ax=ax, legend=True,
                                    legend_kwds={'label': r'$T_{real}/T_{OSM}$'}
                                   )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_frac_time_OSRM_map.pdf'),␣
↪bbox_inches='tight')
```



[141]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='avg_realT_frac_osmT',
    data=tmp_df_stat,
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.05},
    order=1,
)
plt.ylim(0,20)
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_frac_time_OSRM_scatter.pdf'),␣
↪bbox_inches='tight')
```

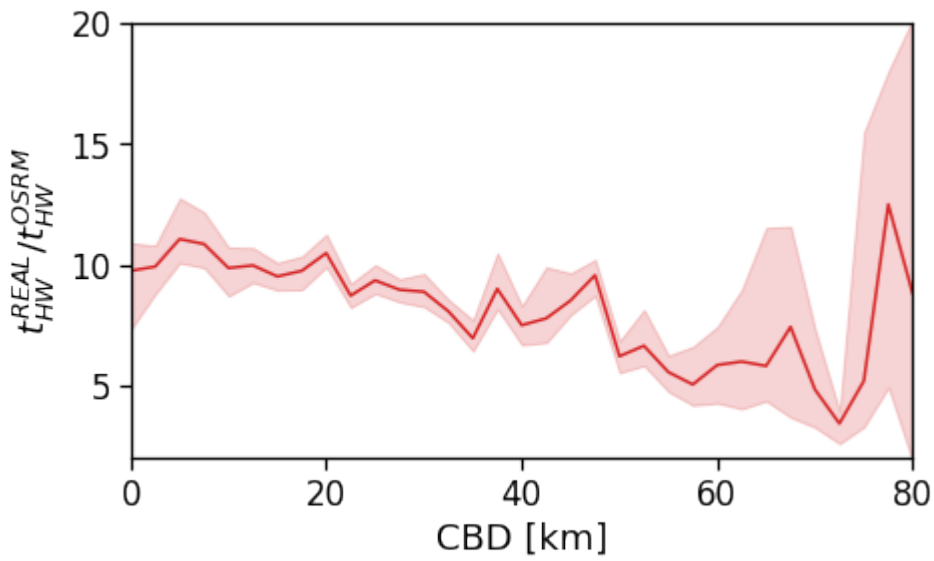


[142]:
```
tmp_data = gdf_aoi_grid_landuse_trip_stats.query('rog_home_count > 20')\
                                .copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='avg_realT_frac_osmT',
            xlabel='CBD [km]',
            ylabel=r'$t^{REAL}_{HW} / t^{OSRM}_{HW}$',
            xlim=[0,80],
            ylim=[2,20],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_frac_time_OSRM_scatterLine.pdf
↪'), bbox_inches='tight')
```



As for the speed of commuting, we observe no clear signal in the congestion rate when we consider the single CBD.

[143]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='avg_realT_frac_osmT',
            xlabel='CBD [km]',
            ylabel=r'$t^{REAL}_{HW} / t^{OSRM}_{HW}$',
            xlim=[0,80],
            ylim=[2,20],
            scatterkws={'alpha': .0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_frac_time_OSRM_scatterLine_
↪user_cbd.pdf'), bbox_inches='tight')
```



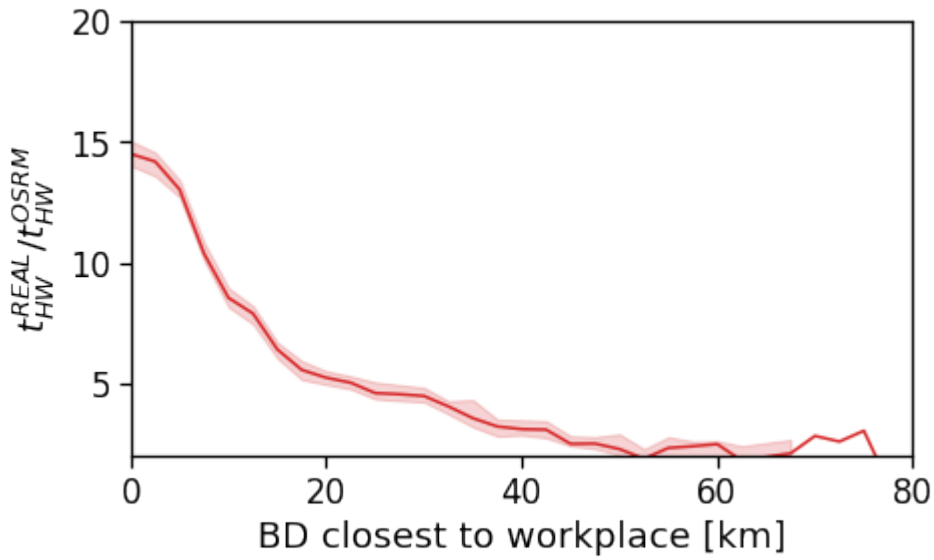
However, once the BD closest to user's workplace is consideres, a clear pattern emerges: BD are always congested resulting in slower commuting.

[144]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='avg_realT_frac_osmT',
            xlabel='BD closest to workplace [km]',
            ylabel=r'$t^{REAL}_{HW} / t^{OSRM}_{HW}$',
            xlim=[0,80],
            ylim=[2,20],
            scatterkws={'alpha': .0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_real_frac_time_OSRM_scatterLine_
```

(continues on next page)

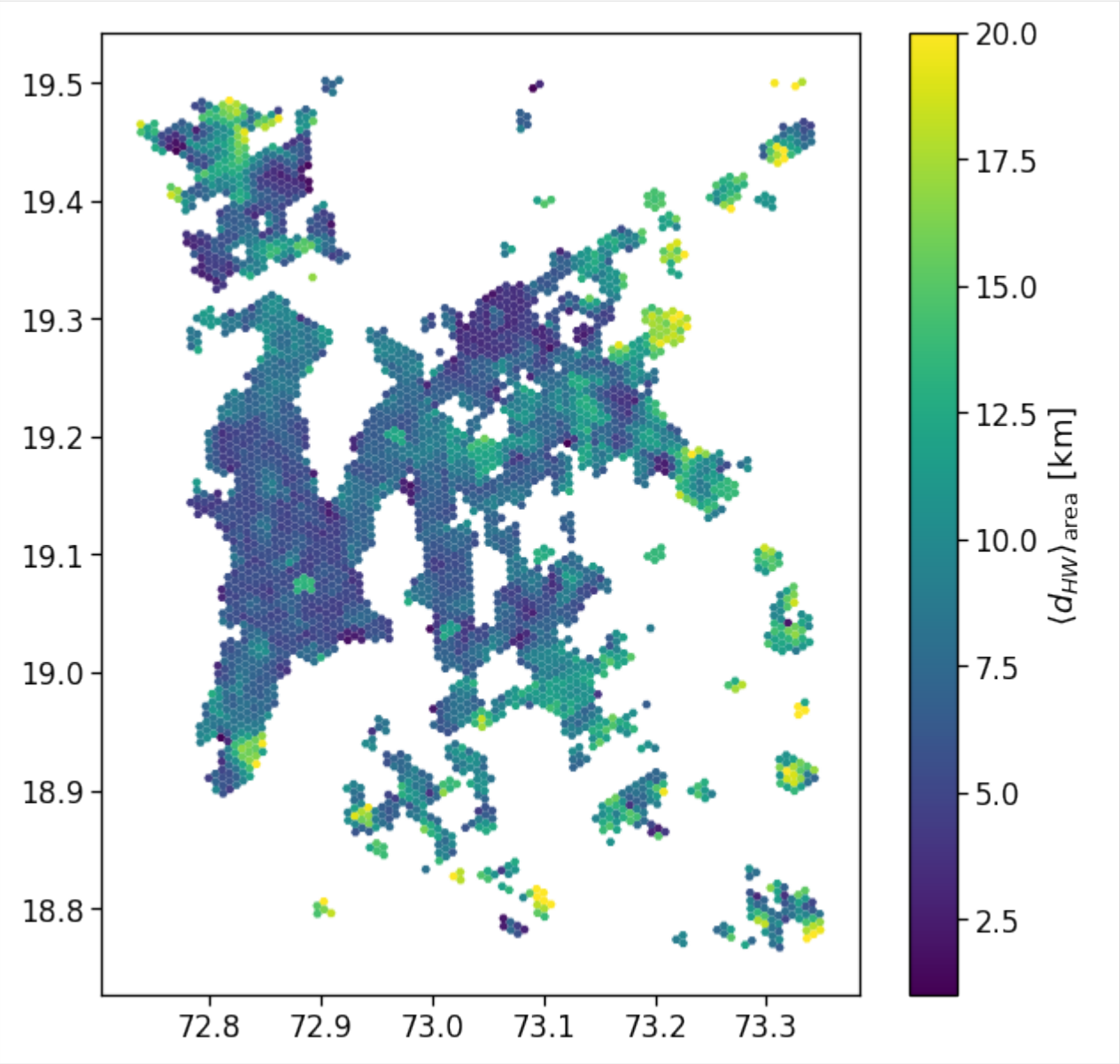
(continued from previous page)

```
↪user_cbd_closest.pdf'), bbox_inches='tight')
```

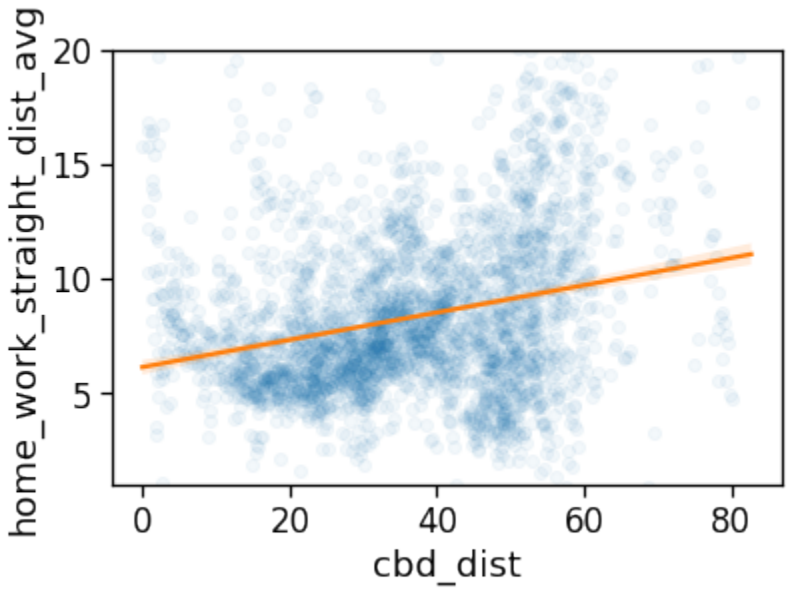


Avg. commute straight distance

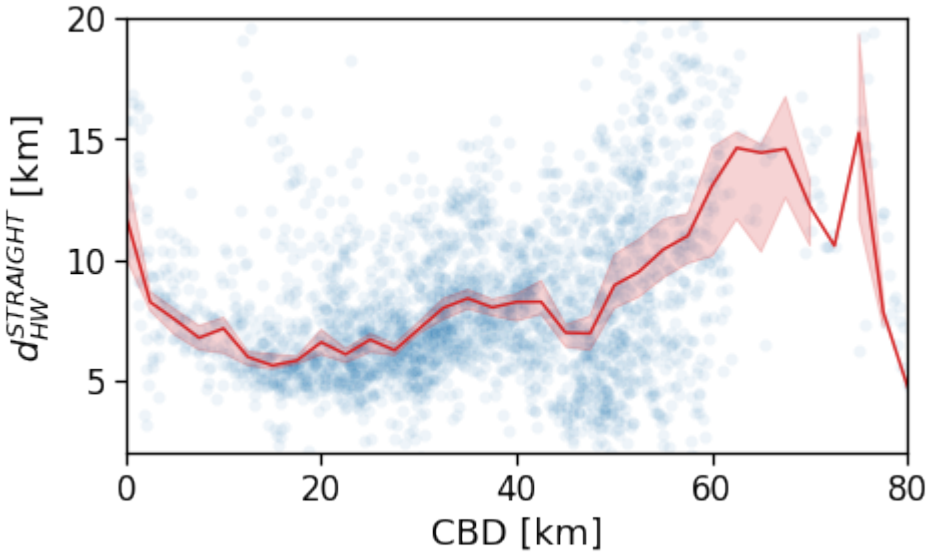
[145]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
tmp_df_stat.plot('home_work_straight_dist_avg',
                                        vmin=1., vmax=20,
                                        ax=ax, legend=True,
                                        legend_kwds={'label': r'$\langle d_{HW}\rangle_{\rm␣
↪area}$ [km]'})
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_straight_dist_map.pdf'), bbox_
↪inches='tight')
```



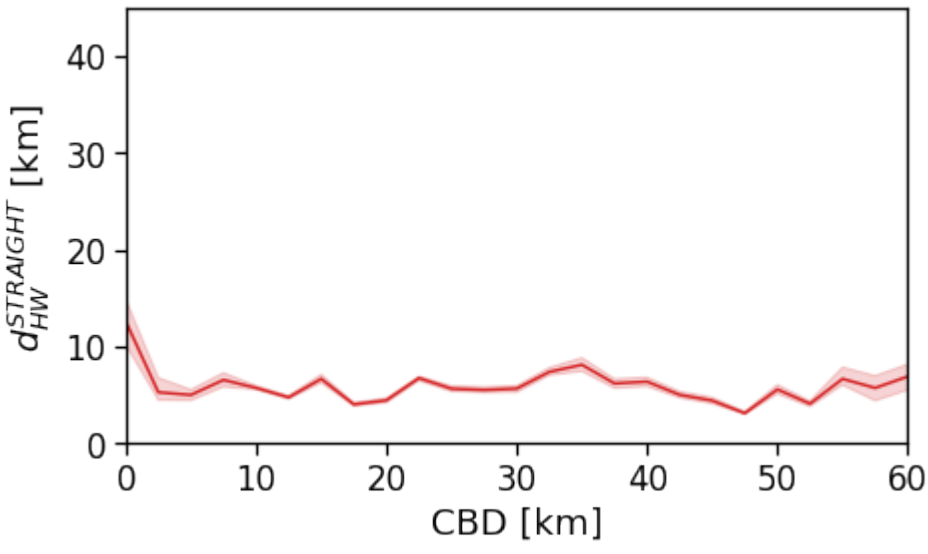
[146]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='home_work_straight_dist_avg',
    data=tmp_df_stat,
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.05},
    order=1,
)
plt.ylim(1.,20)
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_straight_dist_scatter.pdf'), bbox_
↪inches='tight')
```



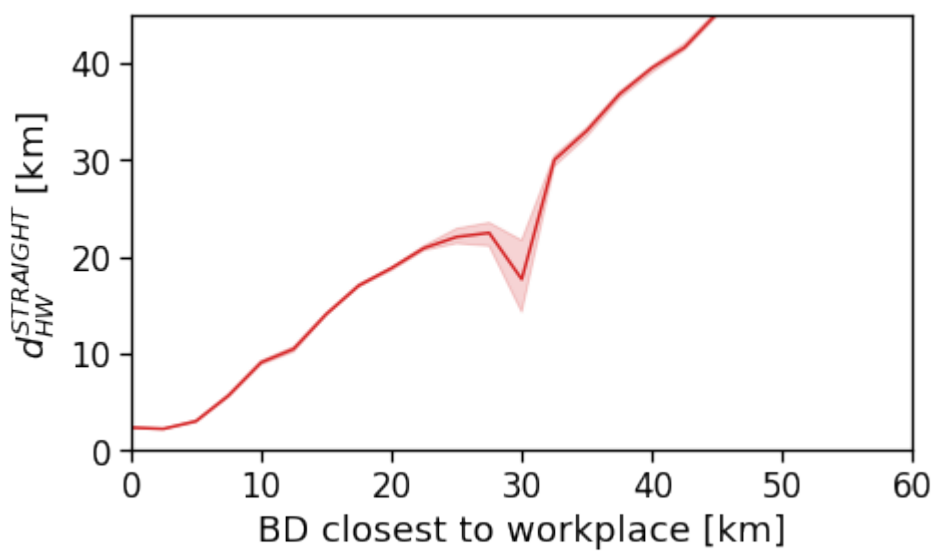
[147]:
```
tmp_data = gdf_aoi_grid_landuse_trip_stats.query('rog_home_count > 20')\
                                .copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='home_work_straight_dist_avg',
            xlabel='CBD [km]',
            ylabel=r'$d^{STRAIGHT}_{HW}$ [km]',
            xlim=[0,80],
            ylim=[2,20],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_straight_dist_scatterLine.pdf'),␣
↪bbox_inches='tight')
```



[148]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='home_work_straight_dist',
            xlabel='CBD [km]',
            ylabel=r'$d^{STRAIGHT}_{HW}$ [km]',
            xlim=[0,60],
            ylim=[0,45],
            scatterkws={'alpha':0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_straight_dist_scatterLine_user_cbd.
↪pdf'), bbox_inches='tight')
```

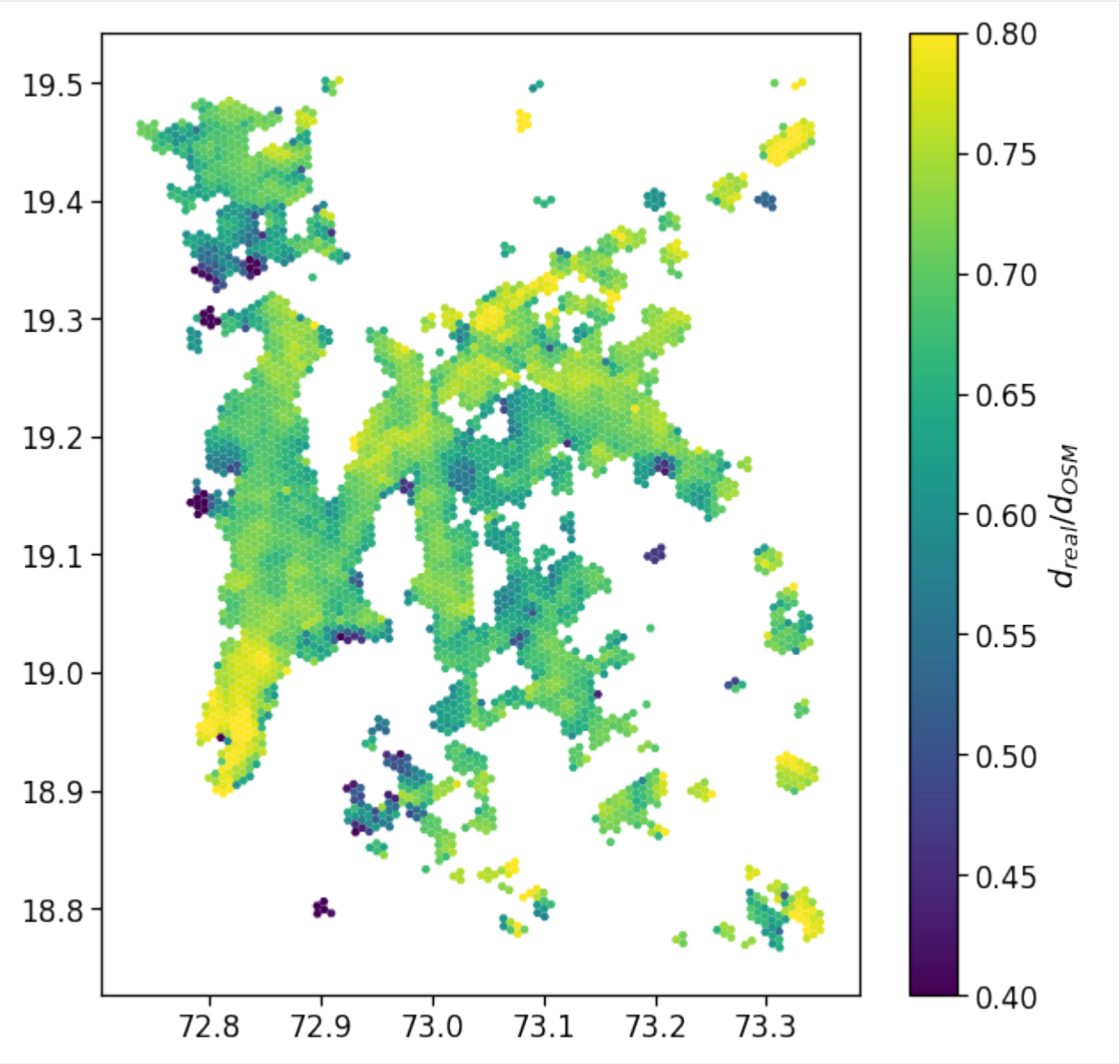


[149]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='home_work_straight_dist',
            xlabel='BD closest to workplace [km]',
            ylabel=r'$d^{STRAIGHT}_{HW}$ [km]',
            xlim=[0,60],
            ylim=[0,45],
            scatterkws={'alpha':0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_time_straight_dist_scatterLine_user_cbd_
↪closest.pdf'), bbox_inches='tight')
```

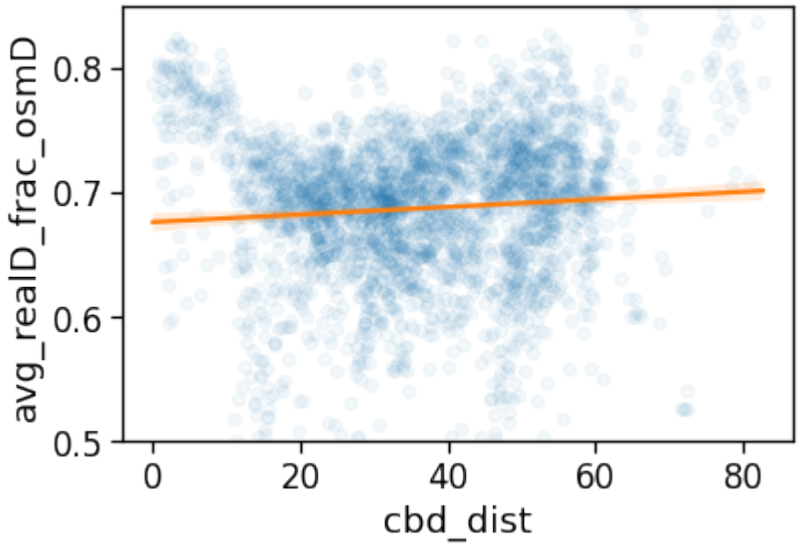


Real distance / straight distance

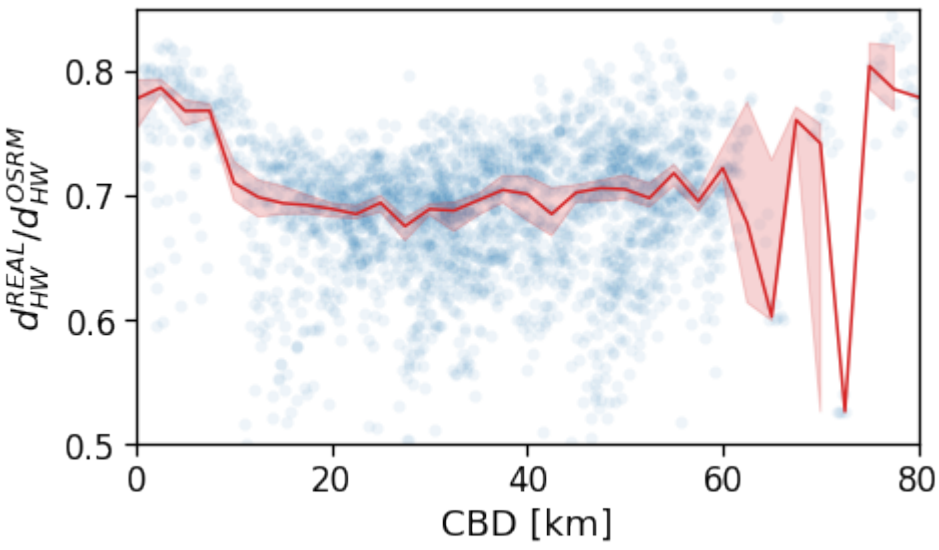
[150]:
```
fig, ax = plt.subplots(1, 1, figsize=(10,10))
tmp_df_stat.plot('avg_realD_frac_osmD',
                                    vmin=.4, vmax=.8,
                                    ax=ax, legend=True,
                                    legend_kwds={'label': r'$d_{real}/d_{OSM}$'}
                                   )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_real_frac_dist_OSRM_map.pdf'),␣
↪bbox_inches='tight')
```



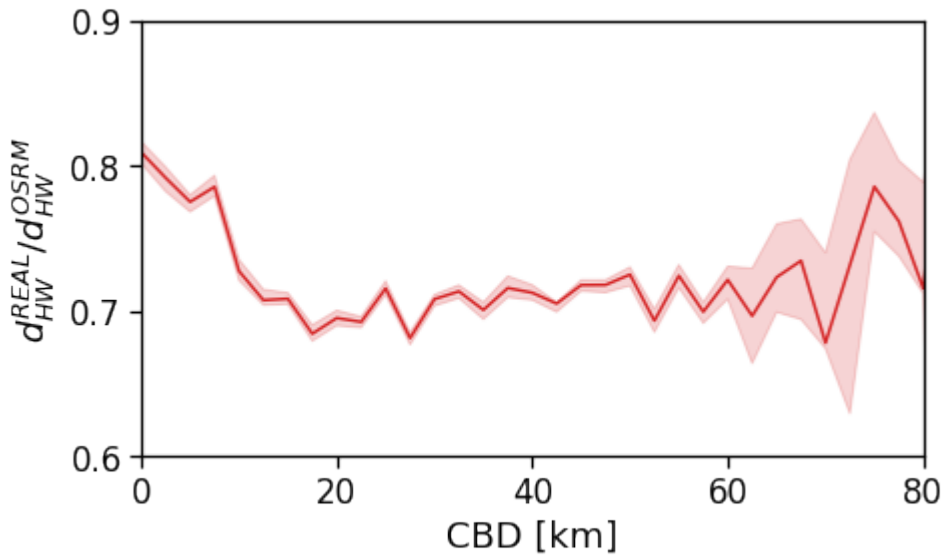
[151]:
```
# The same analysis with a 2-degree interpolating line...
sns.regplot(
    x='cbd_dist',
    y='avg_realD_frac_osmD',
    data=tmp_df_stat,
    line_kws={'color': 'C1'},
    scatter_kws={'alpha':.05},
    order=1,
)
plt.ylim(.5,.85)
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_real_frac_dist_OSRM_scatter.pdf'),␣
↪bbox_inches='tight')
```



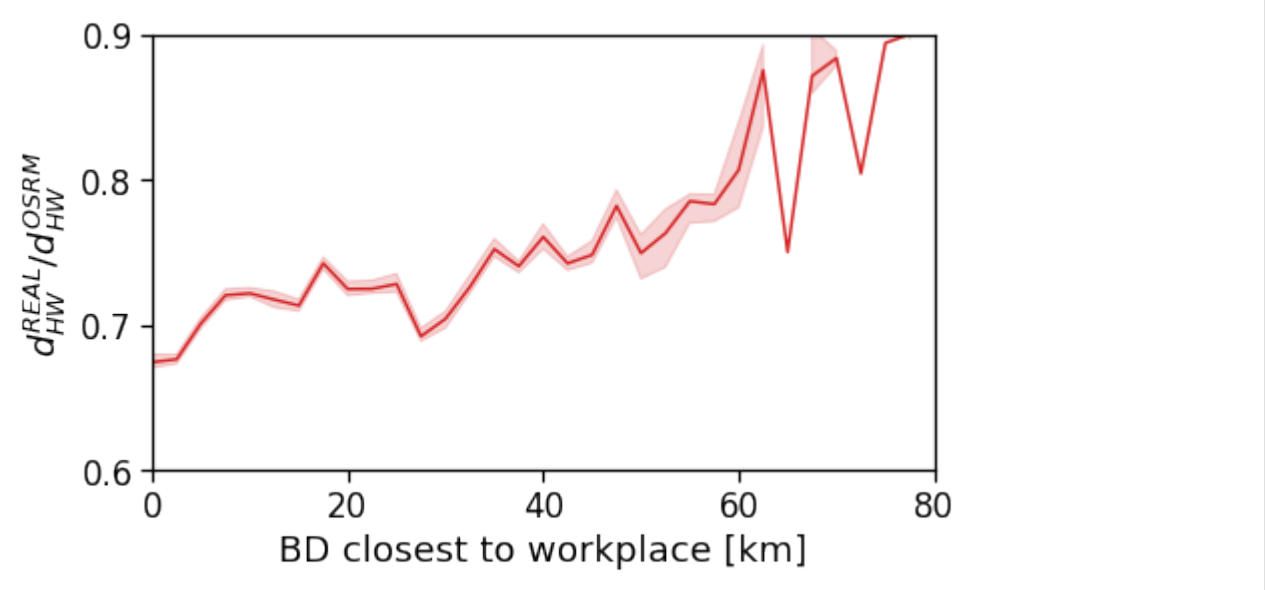
[152]:
```
tmp_data = gdf_aoi_grid_landuse_trip_stats.query('rog_home_count > 20')\
                                .copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='avg_realD_frac_osmD',
            xlabel='CBD [km]',
            ylabel=r'$d^{REAL}_{HW} / d^{OSRM}_{HW}$',
            xlim=[0,80],
            ylim=[.5,.85],
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_real_frac_dist_OSRM_scatterLine.pdf
↪'), bbox_inches='tight')
```



[153]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='cbd_dist',
            x_line='cbd_dist_bin',
            y='avg_realD_frac_osmD',
            xlabel='CBD [km]',
            ylabel=r'$d^{REAL}_{HW} / d^{OSRM}_{HW}$',
            xlim=[0,80],
            ylim=[.6,.9],
            scatterkws={'alpha': 0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_real_frac_dist_OSRM_scatterLine_
↪user_cbd.pdf'), bbox_inches='tight')
```



[154]:
```
tmp_data = cleaned_user_stats_table_df.copy(deep=True)

fig, ax = compareLinePlot(
            data=tmp_data,
            x_scatter='closest_cbd_dist',
            x_line='closest_cbd_dist_bin',
            y='avg_realD_frac_osmD',
            xlabel='BD closest to workplace [km]',
            ylabel=r'$d^{REAL}_{HW} / d^{OSRM}_{HW}$',
            xlim=[0,80],
            ylim=[.6,.9],
            scatterkws={'alpha': 0}
    )
plt.savefig(os.path.join(OUT_DIR_FIG, 'home_work_dist_real_frac_dist_OSRM_scatterLine_
↪user_cbd_closest.pdf'), bbox_inches='tight')
```



[155]:
```
pd.to_pickle(user_stats_table_df,
             os.path.join(OUT_DIR,  'user_stats_table_df.pkl'))
pd.to_pickle(cleaned_user_stats_table_df,
             os.path.join(OUT_DIR,  'cleaned_user_stats_table_df.pkl'))
```

6.6.7 Supplementary info

Alternative way for home and work location

By looking at the single pings.

[156]:
```
dd_dayNight_pings = mk.stats.userHomeWork(exploded_stops_df,
                                          homeHours=homeHours,
                                          workHours=workHours)
df_hw_locs = mk.stats.userHomeWorkLocation(dd_dayNight_pings, force_different=True)
df_hw_locs
```

[156]:
```
Dask DataFrame Structure:
                tot_pings home_tile_ID lat_home lng_home home_pings work_tile_ID lat_
↪work lng_work work_pings
npartitions=200
                    int64        int64  float64  float64      int64        int64 ␣
↪float64  float64      int64
                      ...          ...      ...      ...        ...          ...     ␣
↪...      ...        ...
...                   ...          ...      ...      ...        ...          ...     ␣
↪...      ...        ...
                      ...          ...      ...      ...        ...          ...     ␣
↪...      ...        ...
                      ...          ...      ...      ...        ...          ...     ␣
↪...      ...        ...
Dask Name: determine_home_work_user, 23885 tasks
```

[ ]:

6.7 Urban spatial structure: cities comparison

In this notebook we show the comparison of all the analyzed cities.

Specifically we start by loading the users statistics and figures on stops and pois and we later focus on spatial structure.

Note on data

The data used in this notebook have been provided by Quadrant within the **Resilient Urban Planning Analysis Using Smartphone Location Data** project of the The World Bank / Global Facility for Disaster Reduction and Recovery (GFDRR) - contract number 7204724.

Quadrant (An Appen Company) is a global leader in mobile location data, POI data, and corresponding compliance services. Quadrant provides anonymised location data and location-based business solutions that are fit for purpose, authentic, easy to use, and simple to organise. We offer data for almost all countries in the world, with hundreds of millions of unique devices and tens of billions of events per month, allowing our clients to perform location analyses, derive location-based intelligence, and make well-informed business decisions. Our data is gathered directly from first party opt-in mobile devices through a server-to-server integration with trusted publisher partners, delivering genuine and reliable raw GPS data unlike other location data sources relying heavily on Bidstream. Our consent management platform, QCMP, ensures that our data is compliant with applicable consent and opt-out provisions of data privacy laws governing the collection and use of location data.

[1]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import geopandas as gpd
import matplotlib.pyplot as plt
from datetime import datetime
import sys
import os
from shutil import rmtree
from dask.distributed import Client
from dask import dataframe as dd
import dask
import seaborn as sns
import statsmodels.api as sm
import pytz
import psutil
import multiprocessing
sns.set_context('notebook', font_scale=1.5)

dask.__version__
```

[1]:
```
'2022.04.1'
```

We also import `mobilkit` as `mk` to gain access to all the librarie's capabilities. We import some constant names of columns from the `dask_schemas` submodule as they will be useful later on.

[2]:
```
import mobilkit as mk
from mobilkit.spatial import haversine_pairwise
from mobilkit.dask_schemas import (latColName, lonColName,
```

(continues on next page)

(continued from previous page)

```
                                    uidColName, zidColName,
                                    accColName, utcColName,
                                    dttColName, durColName,
                                    locColName, ldtColName,
                                    medLatColName, medLonColName)
from mobilkit.viz import compareLinePlot
```

6.7.1 Configuration

We start defining the cities to be loaded, where the data are and some auxiliary information for each city.

[3]:
```
ROOT_IN_FOLDER = '/mnt/data/resilience_analyses'
OUT_FOLDER = '/mnt/data/resilience_analyses/comparison'

DATA_CITIES = {}
```

[4]:
```
DATA_CITIES['mumbai'] = {
    'name': 'Mumbai',
    'population': 25632800,
    'aoi_radius_km': 60,
    'to_do': True,
}

DATA_CITIES['ankara'] = {
    'name': 'Ankara',
    'population': 5841050,
    'aoi_radius_km': 30,
    'to_do': True,
}

DATA_CITIES['riyadh'] = {
    'name': 'Riyadh',
    'population': 6623460,
    'aoi_radius_km': 40,
    'to_do': True,
}

DATA_CITIES['rio_de_janeiro'] = {
    'name': 'Rio de Janeiro',
    'population': 12545800,
    'aoi_radius_km': 50,
    'to_do': True,
}

DATA_CITIES['doha'] = {
    'name': 'Doha',
    'population': 2409370,
    'aoi_radius_km': 35,
    'to_do': True,
}
```

(continues on next page)

(continued from previous page)

```
DATA_CITIES['dhaka'] = {
    'name': 'Dhaka',
    'population': 32033100,
    'aoi_radius_km': 40,
    'to_do': False,
}

DATA_CITIES['bogota'] = {
    'name': 'Bogota',
    'population': 10056200,
    'aoi_radius_km': 30,
    'to_do': False,
}

DATA_CITIES['buenos_aires'] = {
    'name': 'Buenos Aires',
    'population': 16877600,
    'aoi_radius_km': 50,
    'to_do': False,
}

DATA_CITIES['ho_chi_minh'] = {
    'name': 'Ho Chi Minh',
    'population': 16431400,
    'aoi_radius_km': 45,
    'to_do': False,
}

DATA_CITIES['johannesburg'] = {
    'name': 'Johannesburg',
    'population': 13609300,
    'aoi_radius_km': 45,
    'to_do': False,
}
```

[5]:
```
os.makedirs(OUT_FOLDER, exist_ok=True)
```

[ ]:
```
```

6.7.2 Create *Dask* client

Here we create and connect to a dask client. We just specify where we want Dask to store tmp files (the `tmp_dask_dir`) and the memory limit per worker.

[6]:
```
tmp_dask_dir = '/mnt/dask_workplace/'
dask.config.set({'temporary_directory': tmp_dask_dir})
os.environ["DASK_TEMPORARY_DIRECTORY"] = tmp_dask_dir
```

[7]:
```
n_proc = int(multiprocessing.cpu_count() / 2) - 1
mem_per_proc = psutil.virtual_memory().total * 1.3 / 1e9 / n_proc
client = Client(
    memory_limit='%.01fG'%mem_per_proc,
    n_workers=n_proc,
    threads_per_worker=2
               )
client
```

```
2022-05-18 10:23:50,377 - distributed.diskutils - INFO - Found stale lock file and␣
↪directory '/mnt/dask_workplace/dask-worker-space/worker-p24zfk__', purging
```

[7]:
```
<Client: 'tcp://127.0.0.1:39577' processes=23 threads=46, memory=162.80 GiB>
```

6.7.3 Load the needed data

[8]:
```
for city, data in DATA_CITIES.items():
    print(data['name'])
    path_users_stats = os.path.join(ROOT_IN_FOLDER, city, 'users_stats.pkl')
    data['users_stats_df'] = pd.read_pickle(path_users_stats)



    path_users_stop_locs = os.path.join(ROOT_IN_FOLDER, city, 'users_stop_locs_df')
    users_stop_locs_df = dd.read_parquet(path_users_stop_locs)
    n_locs = users_stop_locs_df.shape[0].compute()
    data['number_of_locations'] = n_locs

    path_user_locs_stats_hw_separated = os.path.join(ROOT_IN_FOLDER, city,
                                                     'user_locs_stats_hw_separated.pkl')
    data['user_locs_stats_hw_separated'] = pd.read_pickle(path_user_locs_stats_hw_
↪separated)

    path_df_hw_locs_pd = os.path.join(ROOT_IN_FOLDER, city,
                                      'df_hw_locs_pd.pkl')
    data['df_hw_locs_pd'] = pd.read_pickle(path_df_hw_locs_pd)


    data['user_stats_table_df'] = pd.read_pickle(os.path.join(ROOT_IN_FOLDER, city,
                                                      'user_stats_table_df.pkl'))
    data['cleaned_user_stats_table_df'] = pd.read_pickle(os.path.join(ROOT_IN_FOLDER,␣
↪city,
                                                                      'cleaned_user_stats_table_
↪df.pkl'))

    data['out_df_hw_locs'] = pd.read_pickle(os.path.join(ROOT_IN_FOLDER, city, 'out_df_
↪hw_locs.pkl'))
    data['df_cnt_hw_locs'] = pd.read_pickle(os.path.join(ROOT_IN_FOLDER, city, 'out_df_
↪cnt_hw_locs.pkl'))


    data['gdf'] = gpd.read_file(os.path.join(ROOT_IN_FOLDER, city,
```

(continues on next page)

(continued from previous page)

```
                                                'gdf_landuse_rog.gpkg'))
```

```
Mumbai
Ankara
Riyadh
Rio de Janeiro
Doha
Dhaka
Bogota
Buenos Aires
Ho Chi Minh
Johannesburg
```

[9]:
```
for city, data in  DATA_CITIES.items():
    tmp_stats = data['users_stats_df']
    tot_pings = tmp_stats['pings'].sum()
    avg_pings_per_user_per_day = np.mean([v for pp in tmp_stats['pingsPerDay'].values␣
↪for v in pp])
    n_days = (tmp_stats['max_day'].max() - tmp_stats['min_day'].min()).days
    avg_pings_per_day = tot_pings / n_days
    data['avg_pings_per_day'] = avg_pings_per_day
    data['avg_pings_per_user_per_day'] = avg_pings_per_user_per_day

    print(city, avg_pings_per_day, avg_pings_per_user_per_day)
```

```
mumbai 3346701.6 6.592426995706759
ankara 1020145.9333333333 8.223536963944412
riyadh 2967286.466666667 8.364091005227968
rio_de_janeiro 1758042.0 11.443970860967202
doha 374636.1 5.211704756022236
dhaka 526990.8 4.819173879804474
bogota 2400355.9032258065 11.383109358480972
buenos_aires 5114443.354838709 9.892531696796421
ho_chi_minh 5200092.433333334 15.6421293813728
johannesburg 431136.8 3.4879473430436314
```

Check representativeness with statistical requirements

We sort the cities by the ratio of the observed monhly users over the city's population. We check how this ratio decreases as we increase the minimum number of required pings.

[10]:
```
df_mau = []
for city, data in DATA_CITIES.items():
    tmp_stats = data['users_stats_df']

    for thres in [0,10,20,50,100]:
        tmp_df = tmp_stats.query('pings>=@thres')
        tmp_n = tmp_df.shape[0]
        tmp_p = tmp_df['pings'].sum()

        df_mau.append([data['name'], thres, tmp_n, tmp_p,
                       tmp_n / data['population']*100., data['to_do']])
```

(continues on next page)

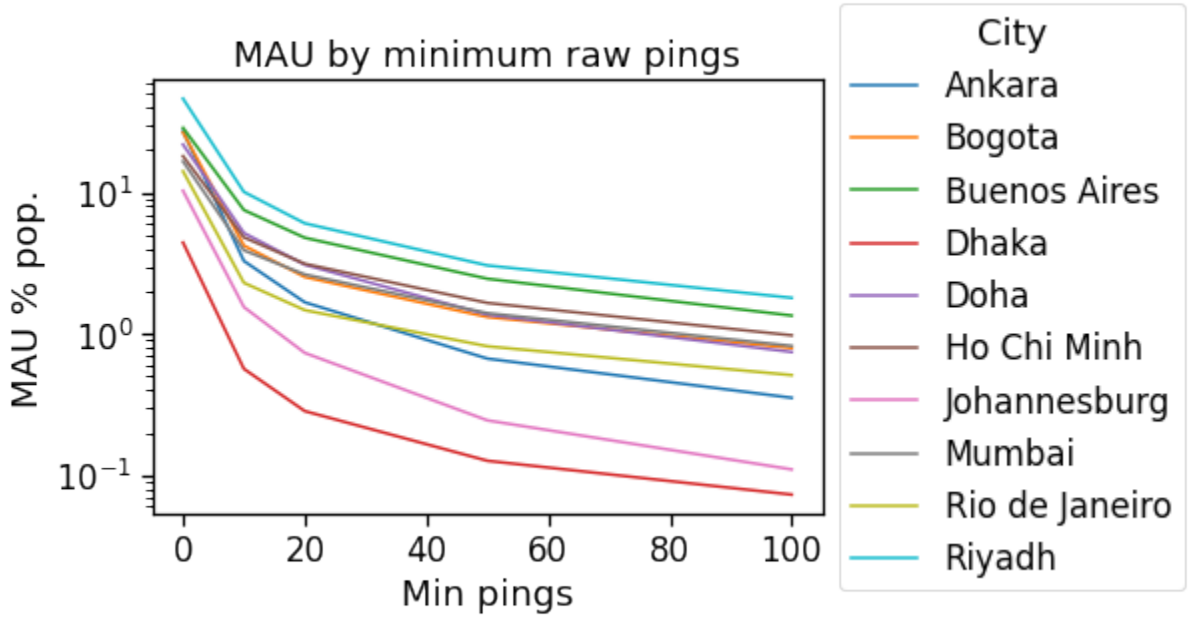
(continued from previous page)

```
df_mau = pd.DataFrame(df_mau, columns=['City', 'Min pings', 'N users', 'N pings', 'MAU %␣
↪pop.', 'Good data'])
df_mau.head(0)
```

[10]:
```
Empty DataFrame
Columns: [City, Min pings, N users, N pings, MAU % pop., Good data]
Index: []
```

[11]:
```
city_order = sorted(list(df_mau['City'].unique()))
```

[12]:
```
g = sns.lineplot(x='Min pings',
                 y='MAU % pop.',
                 hue='City',
                 hue_order=city_order,
                 # style='Good data',
                 # style_order=[True,False],
                 data=df_mau,
            )
plt.semilogy()
plt.title('MAU by minimum raw pings')

# Put the legend out of the figure
g.legend(loc='center left',
         title='City',
         bbox_to_anchor=(1, 0.5))
plt.savefig(os.path.join(OUT_FOLDER, 'mau_users_min_pings.pdf'), bbox_inches='tight')
```



[13]:
```
df_mau[df_mau['Min pings']==10][['City','MAU % pop.']]
```

[13]:
```
              City  MAU % pop.
1           Mumbai    3.903885
6           Ankara    3.279650
```

(continues on next page)

(continued from previous page)

```
11          Riyadh   10.101880
16  Rio de Janeiro    2.299128
21            Doha    5.172929
26           Dhaka    0.566093
31          Bogota    4.214564
36    Buenos Aires    7.519973
41     Ho Chi Minh    4.818226
46    Johannesburg    1.548008
```

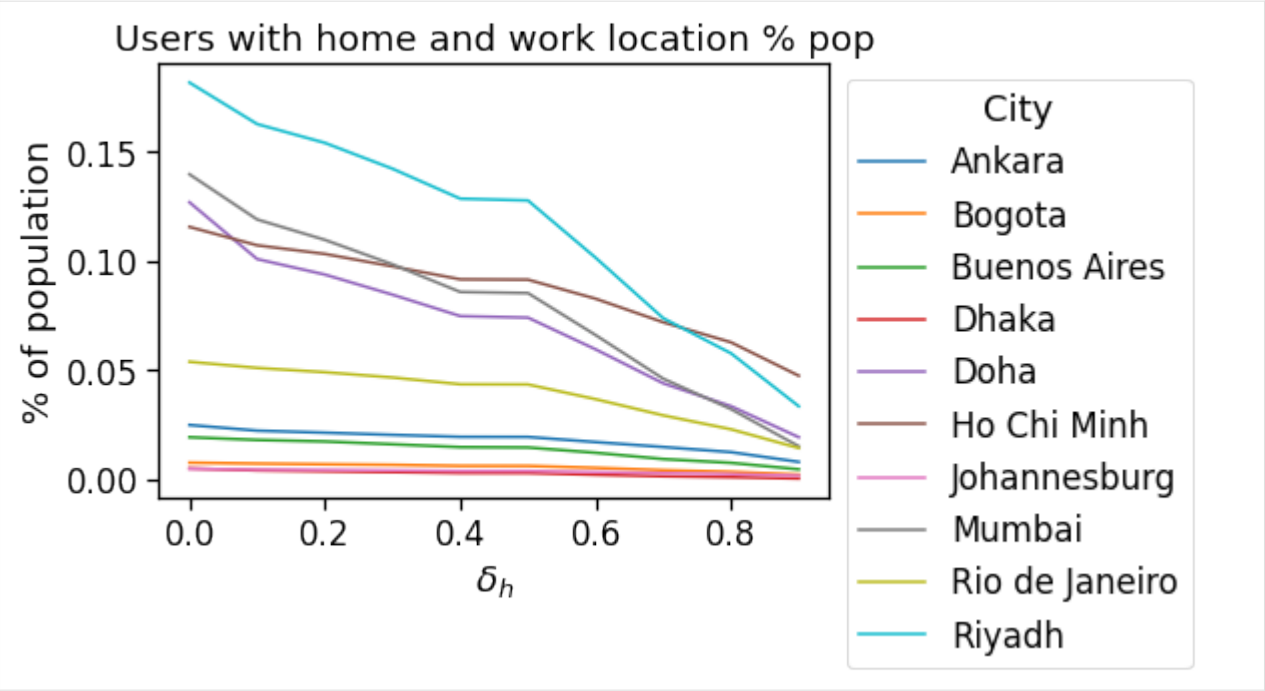
[ ]:

[14]:
```
df_mau_homeWork = None
for city, data in DATA_CITIES.items():
    tmp_stats = data['df_cnt_hw_locs']
    tmp_min_days = tmp_stats['n_days'].min()

    tmp_df = tmp_stats.query('n_days ==%d & n_hours == 10' % max(tmp_min_days, 4)).
↪copy(deep=True)
    tmp_df['City'] = data['name']
    tmp_df['MAU % pop.'] = tmp_df['n_users'] / data['population'] * 100
    tmp_df['MAU with home % pop.'] = tmp_df['with_home_users'] / data['population'] * 100
    tmp_df['MAU with work % pop.'] = tmp_df['with_work_users'] / data['population'] * 100

    df_mau_homeWork = pd.concat((tmp_df, df_mau_homeWork),
                                sort=True, ignore_index=True)
df_mau_homeWork.head(0)
```

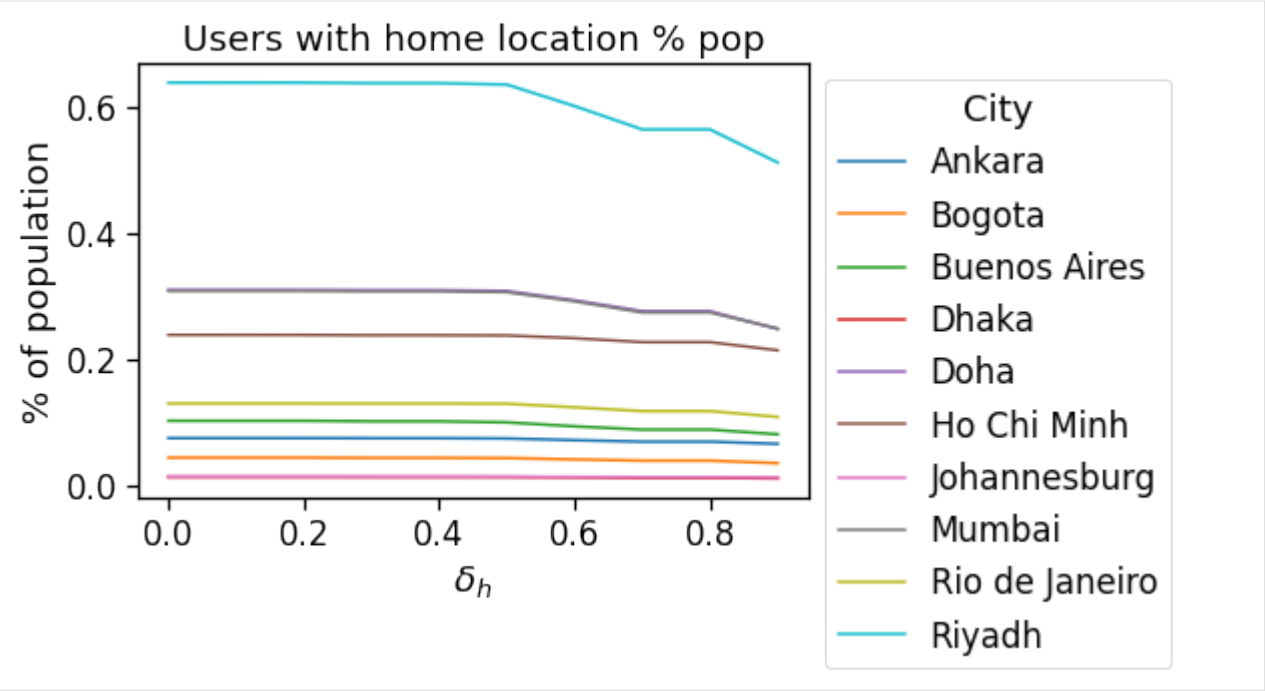
[14]:
```
Empty DataFrame
Columns: [City, MAU % pop., MAU with home % pop., MAU with work % pop., delta_count,␣
↪delta_duration, home_work_same_area_users, home_work_same_area_users_frac, n_days, n_
↪hours, n_users, tot_duration, with_home_users, with_work_users]
Index: []
```

[ ]:

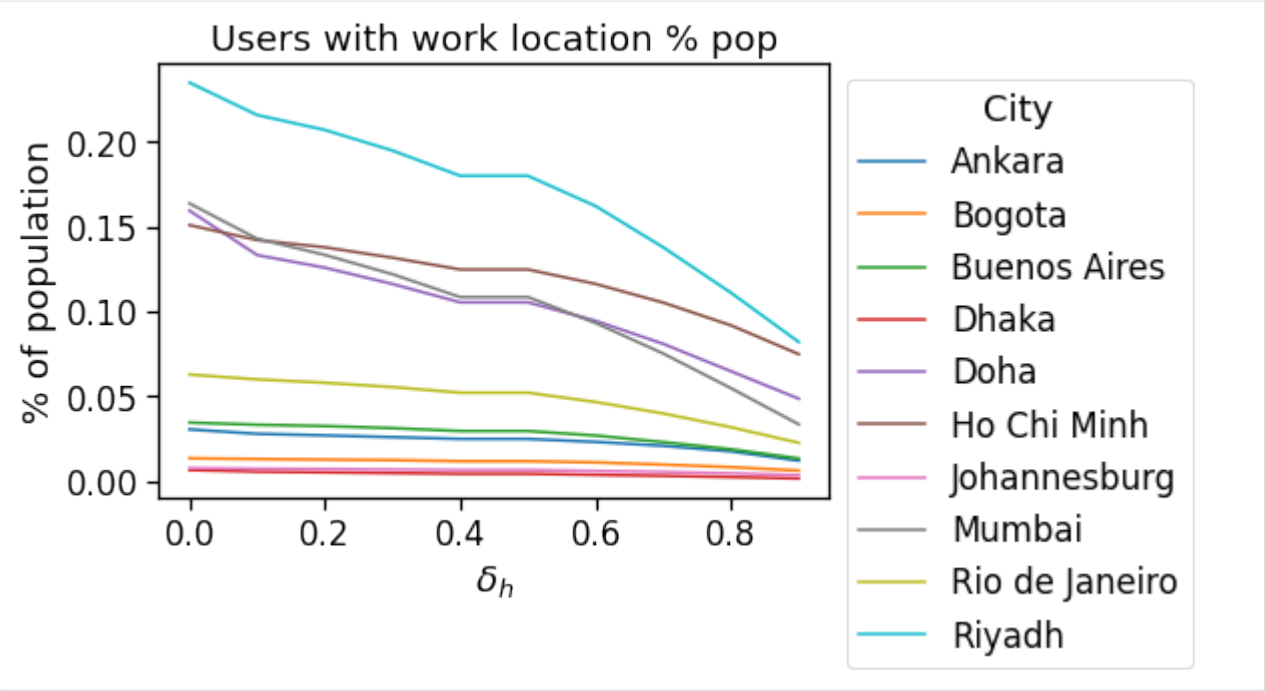
[15]:
```
fig, ax = plt.subplots(1,1,figsize=(6,4))
sns.lineplot(x='delta_duration', y='MAU % pop.',
             data=df_mau_homeWork,
             hue='City',
             hue_order=city_order,
            )
# plt.semilogy()
plt.legend(loc=2, bbox_to_anchor=[1,1], title='City',)
plt.xlabel(r"$\delta_h$")
plt.ylabel("% of population")
plt.title('Users with home and work location % pop')
fig.savefig(os.path.join(OUT_FOLDER, 'mau_users_with_home_work.pdf'), bbox_inches='tight
↪')
```



[16]:
```
fig, ax = plt.subplots(1,1,figsize=(6,4))
sns.lineplot(x='delta_duration', y='MAU with home % pop.',
             data=df_mau_homeWork,
             hue='City',
             hue_order=city_order,
            )
# plt.semilogy()
plt.legend(loc=2, bbox_to_anchor=[1,1], title='City',)
plt.xlabel(r"$\delta_h$")
plt.ylabel("% of population")
plt.title('Users with home location % pop')
fig.savefig(os.path.join(OUT_FOLDER, 'mau_users_with_home.pdf'), bbox_inches='tight')
```



[17]:
```
fig, ax = plt.subplots(1,1,figsize=(6,4))
sns.lineplot(x='delta_duration', y='MAU with work % pop.',
             data=df_mau_homeWork,
             hue='City',
             hue_order=city_order,
            )
# plt.semilogy()
plt.legend(loc=2, bbox_to_anchor=[1,1], title='City',)
plt.xlabel(r"$\delta_h$")
plt.ylabel("% of population")
plt.title('Users with work location % pop')
fig.savefig(os.path.join(OUT_FOLDER, 'mau_users_with_work.pdf'), bbox_inches='tight')
```



6.7.4 Spatial structure

We test the universality of the spatial structure of the cities by comparing the functioning of the different indicators in the different context.

[1]:
```
min_count_per_area = 10
```

[19]:
```
df_spatial_areas = None
for city, data in DATA_CITIES.items():
    if not data['to_do']: continue

    tmp_df = data['gdf'].query('rog_home_count >= @min_count_per_area').copy(deep=True)
    tmp_df['City'] = data['name']
    tmp_df['cbd_dist_norm'] = tmp_df['cbd_dist'] / data['aoi_radius_km']
    tmp_df['cbd_dist_bin_norm'] = np.round(tmp_df['cbd_dist_norm'], decimals=1)

    df_spatial_areas = pd.concat((tmp_df, df_spatial_areas),
                                 sort=True, ignore_index=True)
df_spatial_areas.head(0)
```

[19]:
```
Empty GeoDataFrame
Columns: [City, avg_realD_frac_osmD, avg_realT_frac_osmT, bottom, cbd_dist, cbd_dist_bin,
↪ cbd_dist_bin_norm, cbd_dist_norm, cluster, geometry, home_work_osrm_dist_avg, home_
↪work_osrm_dist_max, home_work_osrm_dist_min, home_work_osrm_dist_std, home_work_osrm_
↪time_avg, home_work_osrm_time_max, home_work_osrm_time_min, home_work_osrm_time_std,␣
↪home_work_ratio, home_work_straight_dist_avg, home_work_straight_dist_max, home_work_
↪straight_dist_min, home_work_straight_dist_std, id, left, nUsers, n_homes, n_works,␣
↪right, rog_count, rog_home_count, rog_home_max, rog_home_mean, rog_home_min, rog_home_
↪std, rog_max, rog_mean, rog_min, rog_std, rog_total_count, rog_total_max, rog_total_
↪mean, rog_total_min, rog_total_std, scope, speed_trips_hw_avg, speed_trips_hw_max,␣
```

(continues on next page)

(continued from previous page)

```
↪speed_trips_hw_min, speed_trips_hw_std, speed_trips_wh_avg, speed_trips_wh_max, speed_
↪trips_wh_min, speed_trips_wh_std, tile_ID, time_trips_hw_avg, time_trips_hw_max, time_
↪trips_hw_min, time_trips_hw_std, time_trips_wh_avg, time_trips_wh_max, time_trips_wh_
↪min, time_trips_wh_std, top, ttd_count, ttd_max, ttd_mean, ttd_min, ttd_std, work_home_
↪osrm_dist_avg, work_home_osrm_dist_max, work_home_osrm_dist_min, work_home_osrm_dist_
↪std, work_home_osrm_time_avg, work_home_osrm_time_max, work_home_osrm_time_min, work_
↪home_osrm_time_std]
Index: []

[0 rows x 76 columns]
```

[20]:
```
df_spatial_users = None
for city, data in DATA_CITIES.items():
    if not data['to_do']: continue

    tmp_df = data['cleaned_user_stats_table_df'].copy(deep=True)
    tmp_df['City'] = data['name']

    tmp_df['aoi_radius_km'] = data['aoi_radius_km']

    tmp_df['cbd_dist_norm'] = tmp_df['cbd_dist'] / tmp_df['aoi_radius_km']
    tmp_df['cbd_dist_bin_norm'] = np.round(tmp_df['cbd_dist_norm'], decimals=1)

    tmp_df['closest_cbd_dist_norm'] = tmp_df['closest_cbd_dist'] / tmp_df['aoi_radius_km
↪']
    tmp_df['closest_cbd_dist_bin_norm'] = np.round(tmp_df['closest_cbd_dist_norm'],␣
↪decimals=1)

    df_spatial_users = pd.concat((tmp_df, df_spatial_users),
                                 sort=True, ignore_index=True)
df_spatial_users.head(0)
```

[20]:
```
Empty DataFrame
Columns: [City, aoi_radius_km, avg, avg_realD_frac_osmD, avg_realT_frac_osmT, cbd_dist,␣
↪cbd_dist_bin, cbd_dist_bin_norm, cbd_dist_norm, closest_cbd_dist, closest_cbd_dist_bin,
↪ closest_cbd_dist_bin_norm, closest_cbd_dist_norm, closest_cbd_idx, com_home_lat, com_
↪home_lng, com_total_lat, com_total_lng, daysActive, daysSpanned, home_loc_ID, home_
↪pings, home_work_osrm_dist, home_work_osrm_dist_km, home_work_osrm_time, home_work_
↪osrm_time_h, home_work_straight_dist, lat_home, lat_work, lng_home, lng_work, max_day,␣
↪min_day, n_pings, pings, pingsPerDay, rog, rog_home, rog_total, speed_trips_hw, speed_
↪trips_wh, time_trips_hw, time_trips_wh, tot_pings, ttd, work_home_osrm_dist, work_home_
↪osrm_time, work_loc_ID, work_pings]
Index: []

[0 rows x 49 columns]
```

[21]:
```
city_todo_order = sorted(list(df_spatial_areas['City'].unique()))
```

[22]:
```
def plotLines(
    x, y, data,
    hue='City',
    hue_order=city_todo_order,
```

(continues on next page)

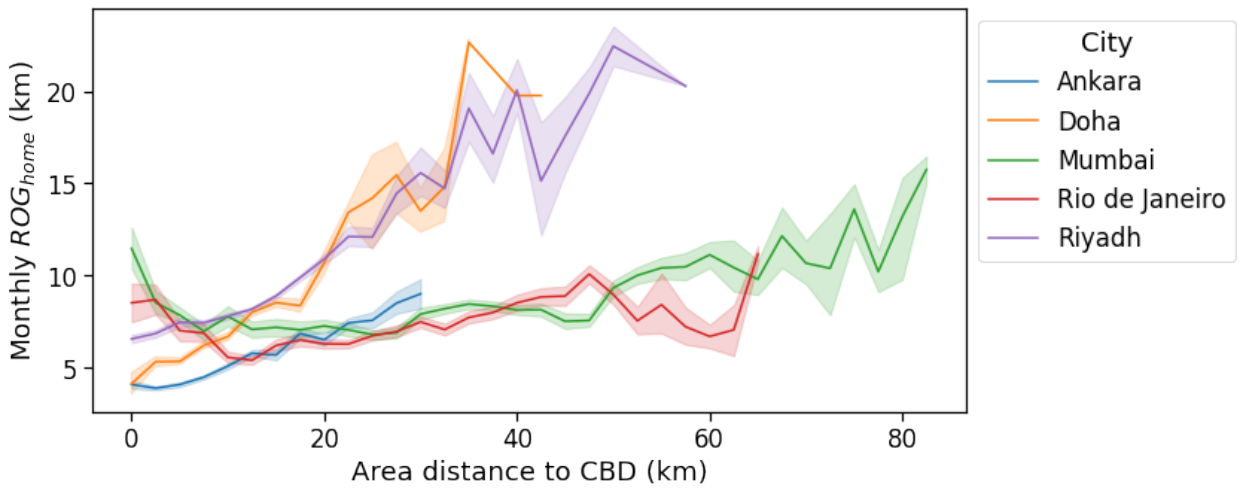
(continued from previous page)

```
    figsize=(12,5),
    xlabel='ylabel',
    ylabel='xlabel',
    title=None,
    xlim=None,
    ylim=None,
    kwargs_leg={},
    tight=True,
    **kwargs,
):
    fig, ax = plt.subplots(1,1,figsize=figsize)
    g = sns.lineplot(data=data,
                 x=x,
                 y=y,
                 hue=hue,
                 hue_order=hue_order,
                ax=ax,
                 **kwargs
    )
    g.legend(loc=2, bbox_to_anchor=[1,1], title=hue, **kwargs_leg)

    ax.set_title(title)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)

    if xlim:
        ax.set_xlim(xlim)
    if ylim:
        ax.set_ylim(ylim)
    if tight:
        fig.tight_layout()
    return fig, ax
```

Radius of Gyration (ROG) vs. distance to Central Business District

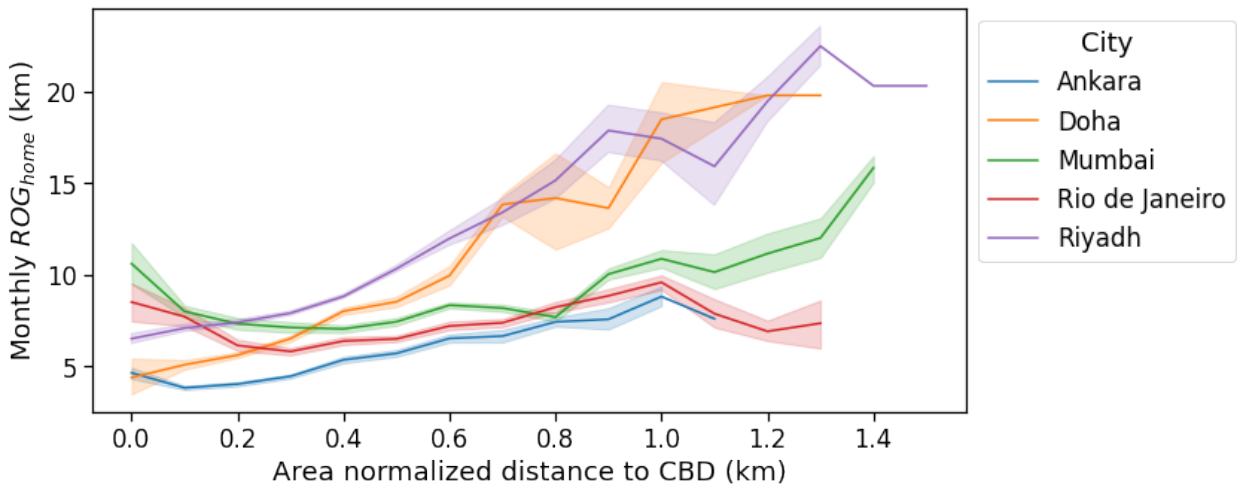
In this first plot we observe that Doha and Riyadh display a strong monocentric pattern, whereas the other cities feature a non-monotonic increase of the ROG (centered on the users' homes) as we move away from the CBD.

[23]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
    y='rog_home_mean',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel=r'Monthly $ROG_{home}$ (km)',
    # title=r'Per area $ROG_{home}$ vs. distance to CBD'
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_home_month_area_cbd.pdf'), bbox_inches='tight')
```



The same holds if we normalize the distance from the CBD by the city's radius.

[24]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin_norm',
    y='rog_home_mean',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area normalized distance to CBD (km)',
    ylabel=r'Monthly $ROG_{home}$ (km)',
    # title=r'Per area $ROG_{home}$ vs. normalized distance to CBD'
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_home_month_area_cbd_norm.pdf'), bbox_inches=
↪'tight')
```



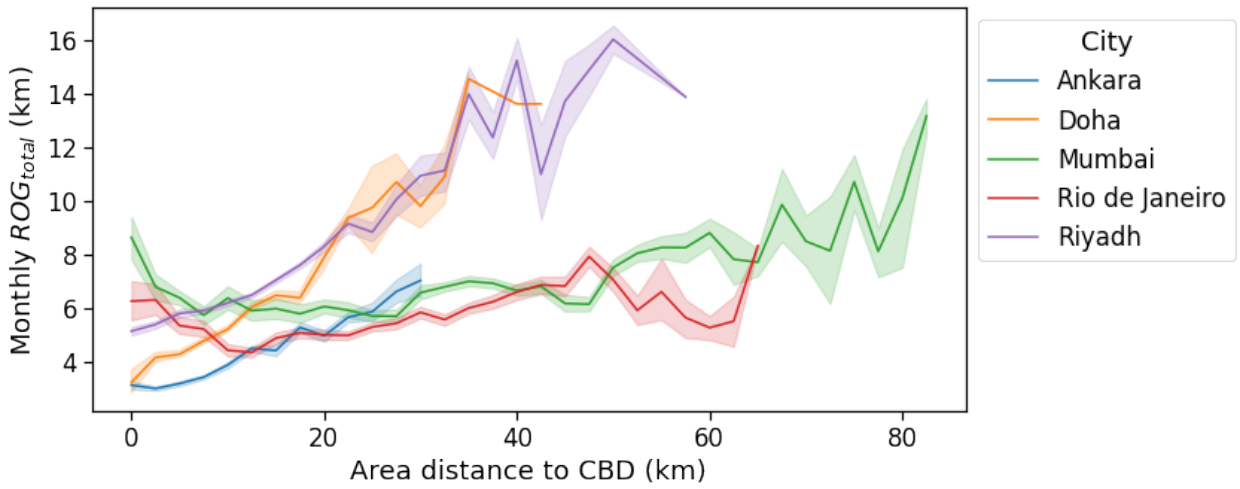
The same holds if we measure the ROG with respect to the user's baricenter.

[25]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
```

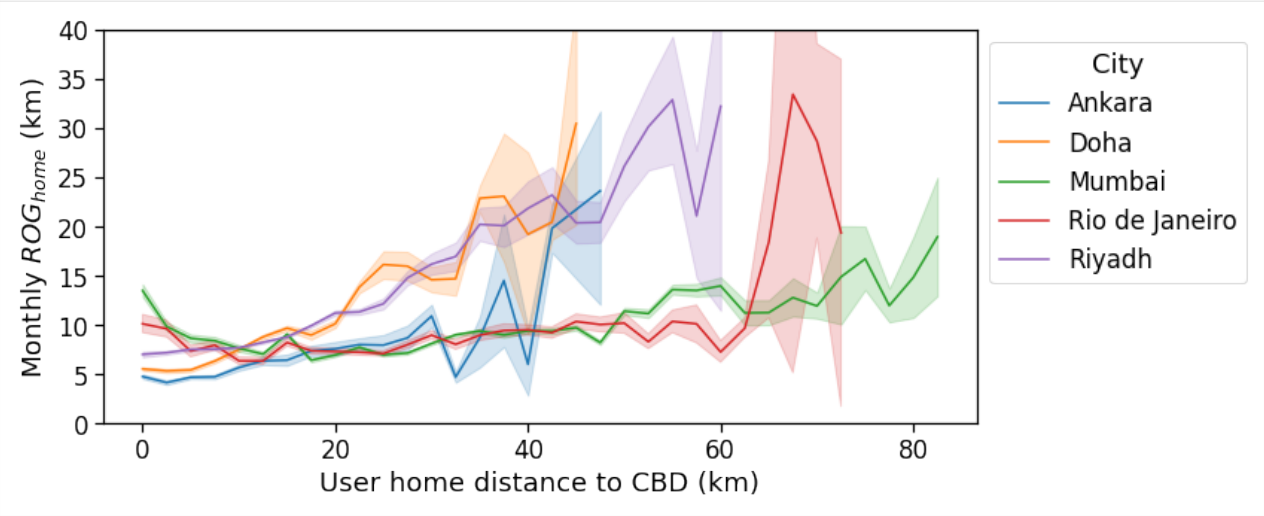
(continues on next page)

(continued from previous page)

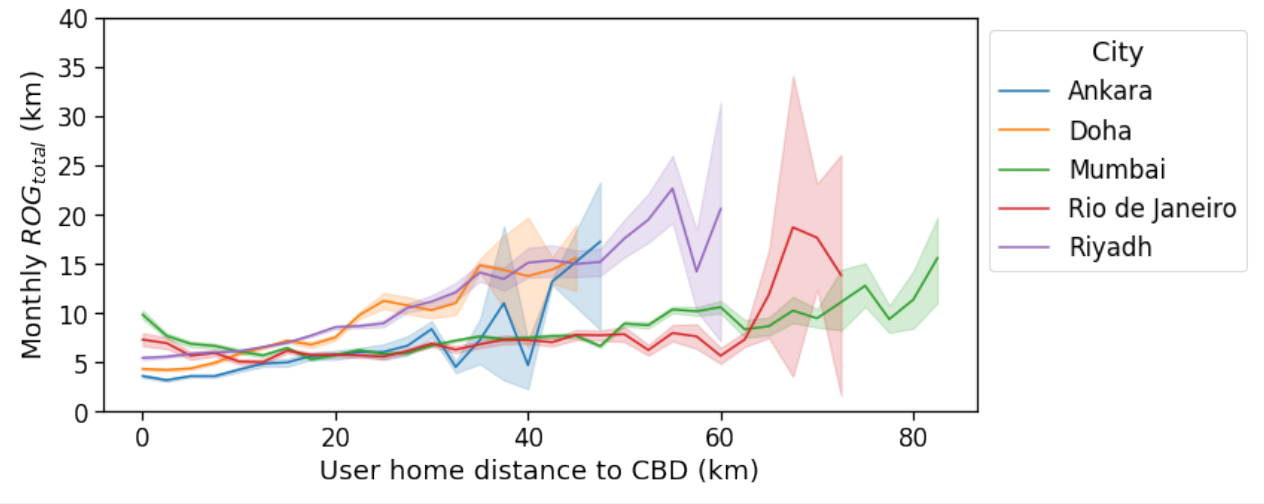
```
    y='rog_total_mean',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel=r'Monthly $ROG_{total}$ (km)',
    # title=r'Per area $ROG_{total}$ vs. distance to CBD'
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_total_month_area_cbd.pdf'), bbox_inches='tight
↪')
```



[26]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin',
    y='rog_home',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to CBD (km)',
    ylabel=r'Monthly $ROG_{home}$ (km)',
    # title=r'Per user $ROG_{home}$ vs. home distance to CBD',
    ylim=(0,40),
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_home_month_user_cbd.pdf'), bbox_inches='tight')
```



[27]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin',
    y='rog_total',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to CBD (km)',
    ylabel=r'Monthly $ROG_{total}$ (km)',
    # title=r'Per user $ROG_{total}$ vs. home distance to CBD',
    ylim=(0,40),
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_total_month_user_cbd.pdf'), bbox_inches='tight
↪')
```

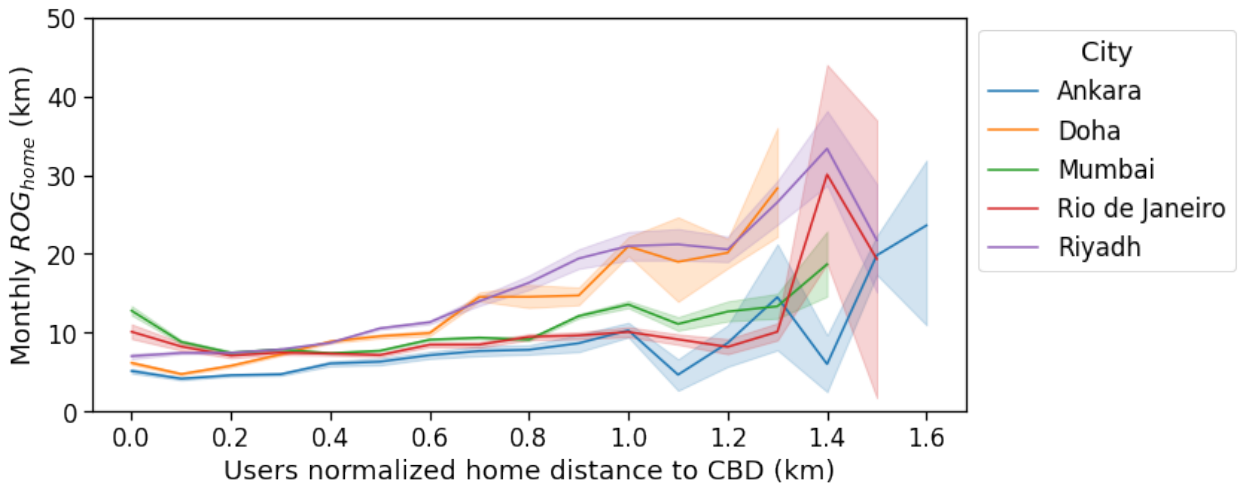


[28]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin_norm',
    y='rog_home',
    hue='City',
```

(continues on next page)

(continued from previous page)

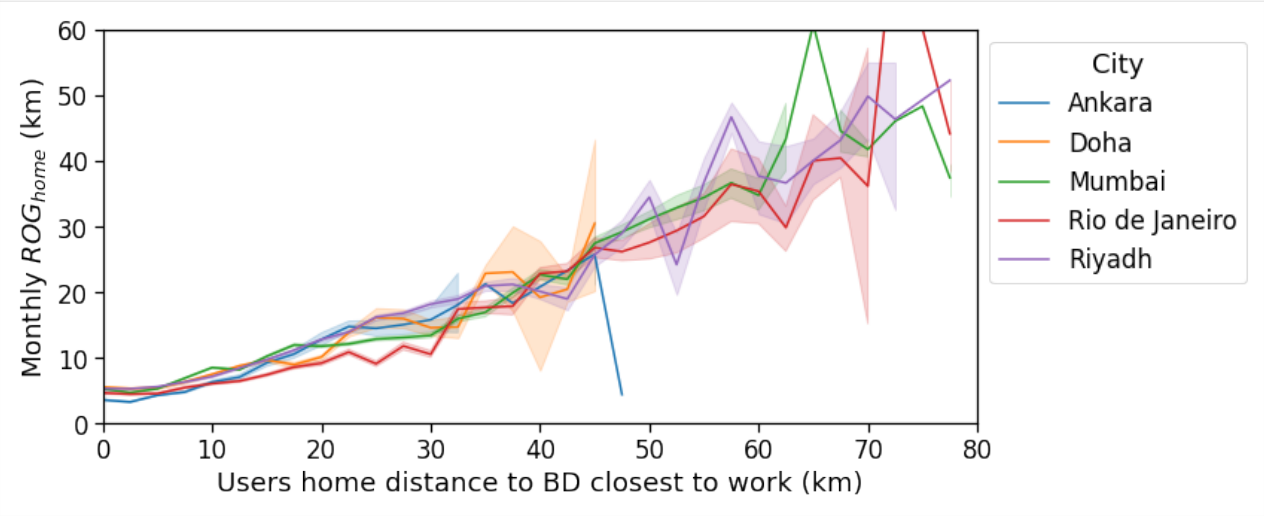
```
    hue_order=city_todo_order,
    xlabel=r'Users normalized home distance to CBD (km)',
    ylabel=r'Monthly $ROG_{home}$ (km)',
    # title=r'Per user $ROG_{home}$ vs. normalized home distance to CBD',
    ylim=(0,50),
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_home_month_user_cbd_norm.pdf'), bbox_inches=
↪'tight')
```



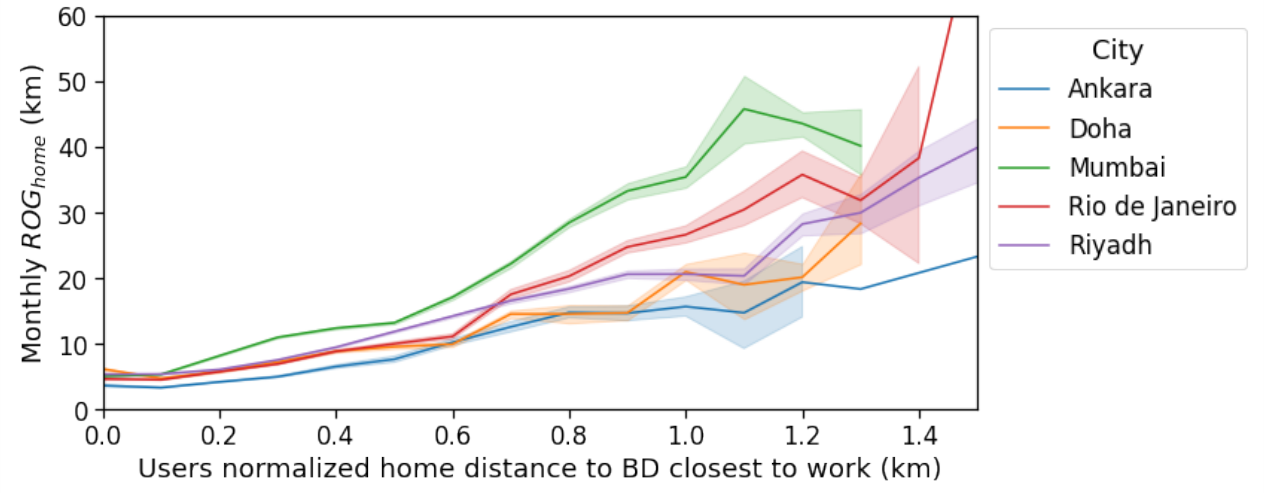
Radius of Gyration (ROG) vs. distance to Business District closest to workplace

If we instead compte the average ROG for users with the home at a given distance from the BD closest to the user's workplace, all the cities collapse on a single curve: the new definition of distance from user-dependent BD instead of the unique CBD cancels out the polycentric structure of the city and all the signals lie on a single, monotonically increasing line.

[29]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin',
    y='rog_home',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Users home distance to BD closest to work (km)',
    ylabel=r'Monthly $ROG_{home}$ (km)',
    # title=r'Per user $ROG_{home}$ vs. home distance to BD closest to workplace',
    xlim=[0,80],
    ylim=(0,60),
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_home_month_user_cbd_closest.pdf'), bbox_inches=
↪'tight')
```

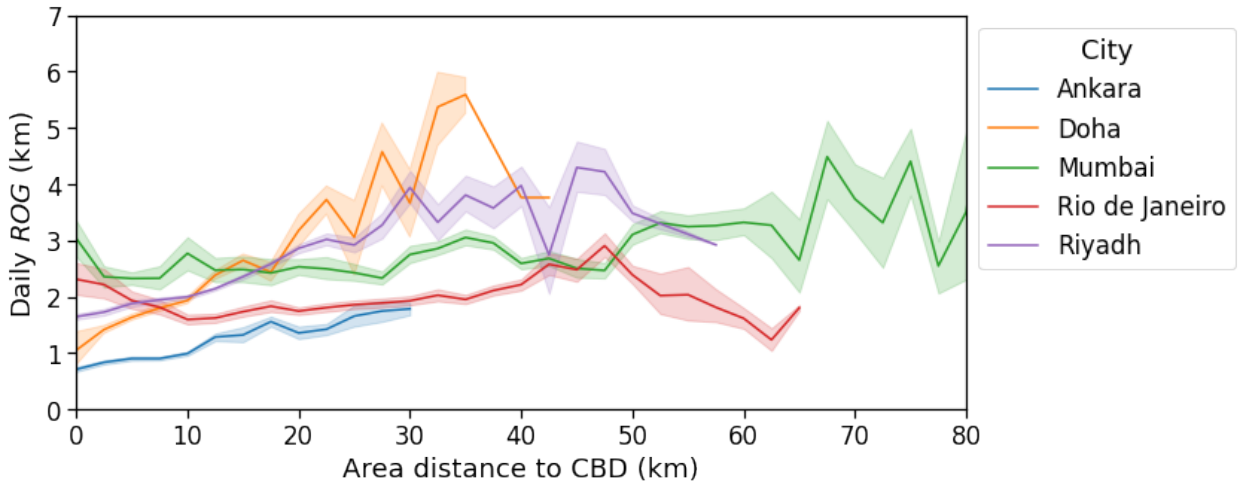


[30]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin_norm',
    y='rog_home',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Users normalized home distance to BD closest to work (km)',
    ylabel=r'Monthly $ROG_{home}$ (km)',
    # title=r'Per user $ROG_{home}$ vs. normalized home distance to BD closest to␣
↪workplace',
    xlim=[0,1.5],
    ylim=(0,60),
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_home_month_user_cbd_closest_norm.pdf'), bbox_
↪inches='tight')
```

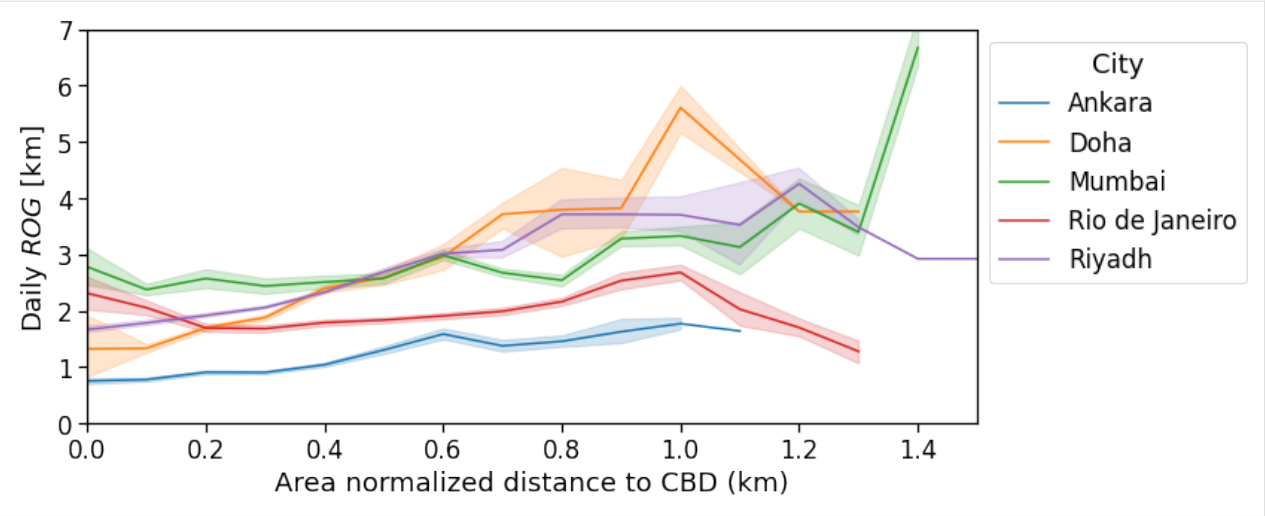


Daily Radius of Gyration (ROG) vs. distance to Business District closest to workplace

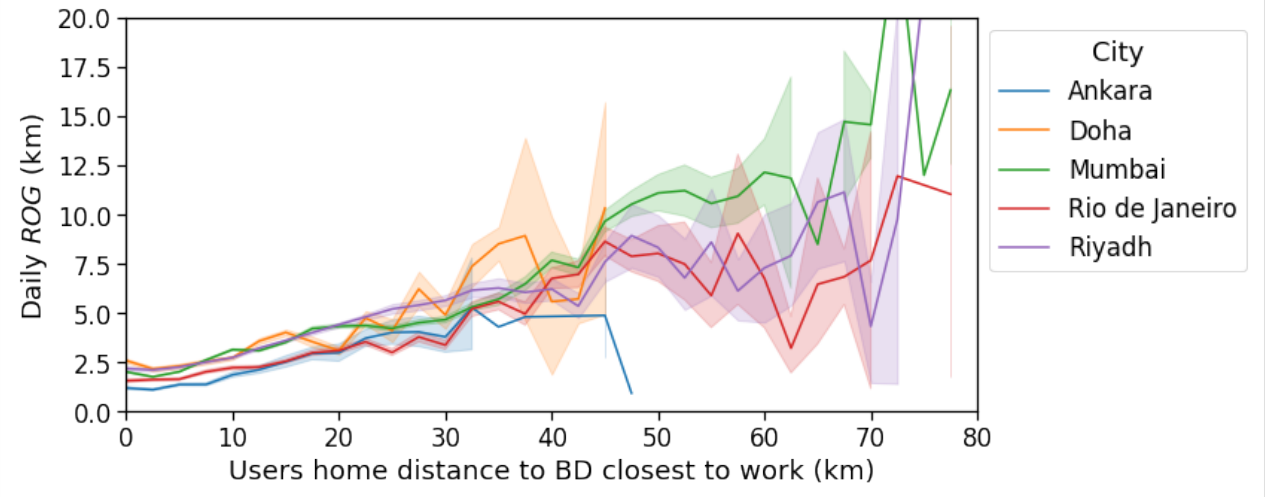
[31]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
    y='rog_mean',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel=r'Daily $ROG$ (km)',
    # title=r'Per area $ROG$ vs. distance to CBD',
    xlim=[0,80],
    ylim=(0,7),
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_daily_area_cbd.pdf'), bbox_inches='tight')
```



[32]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin_norm',
    y='rog_mean',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area normalized distance to CBD (km)',
    ylabel=r'Daily $ROG$ [km]',
    # title=r'Per area $ROG$ vs. normalized distance to CBD',
    ylim=(0,7),
    xlim=[0,1.5]
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_daily_area_cbd_norm.pdf'), bbox_inches='tight')
```



[33]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin',
    y='rog',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Users home distance to BD closest to work (km)',
    ylabel=r'Daily $ROG$ (km)',
    # title=r'Per user $ROG$ vs. home distance to BD closest to workplace',
    ylim=(0,20),
    xlim=[0,80],
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_daily_user_cbd_closest.pdf'), bbox_inches=
↪'tight')
```

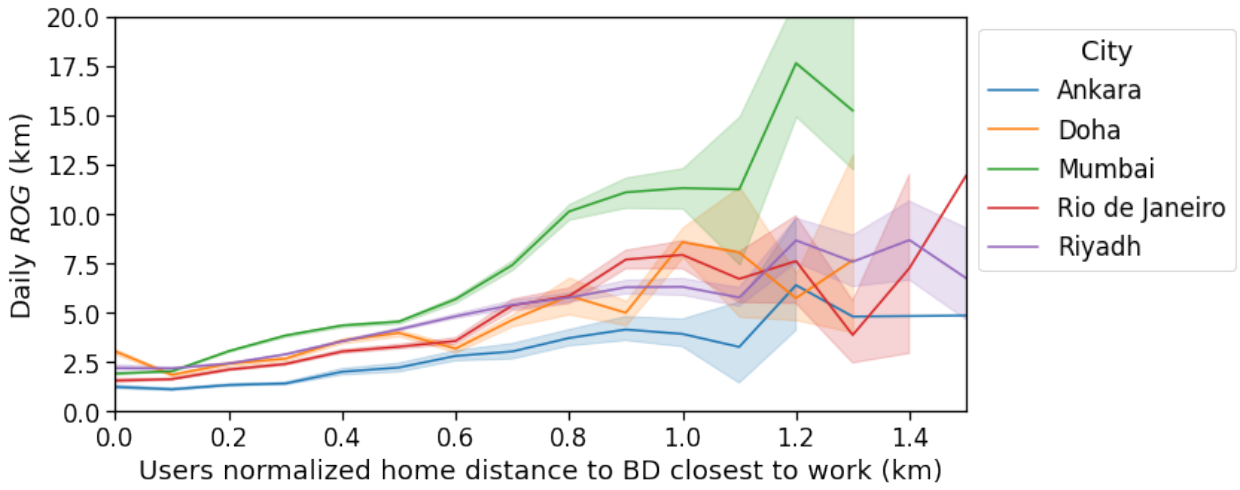


[34]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin_norm',
    y='rog',
```

(continues on next page)

(continued from previous page)

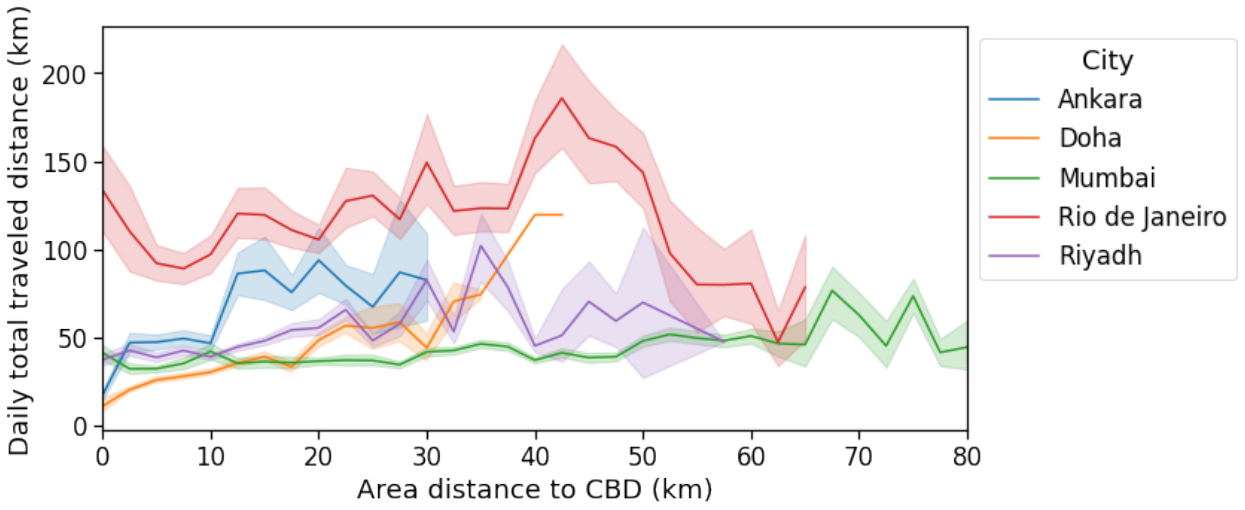
```
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Users normalized home distance to BD closest to work (km)',
    ylabel=r'Daily $ROG$ (km)',
    # title=r'Per user daily $ROG$ vs. normalized home distance to BD closest to workplace
↪',
    xlim=[0,1.5],
    ylim=(0,20),
)
fig.savefig(os.path.join(OUT_FOLDER, 'rog_daily_user_cbd_closest_norm.pdf'), bbox_inches=
↪'tight')
```



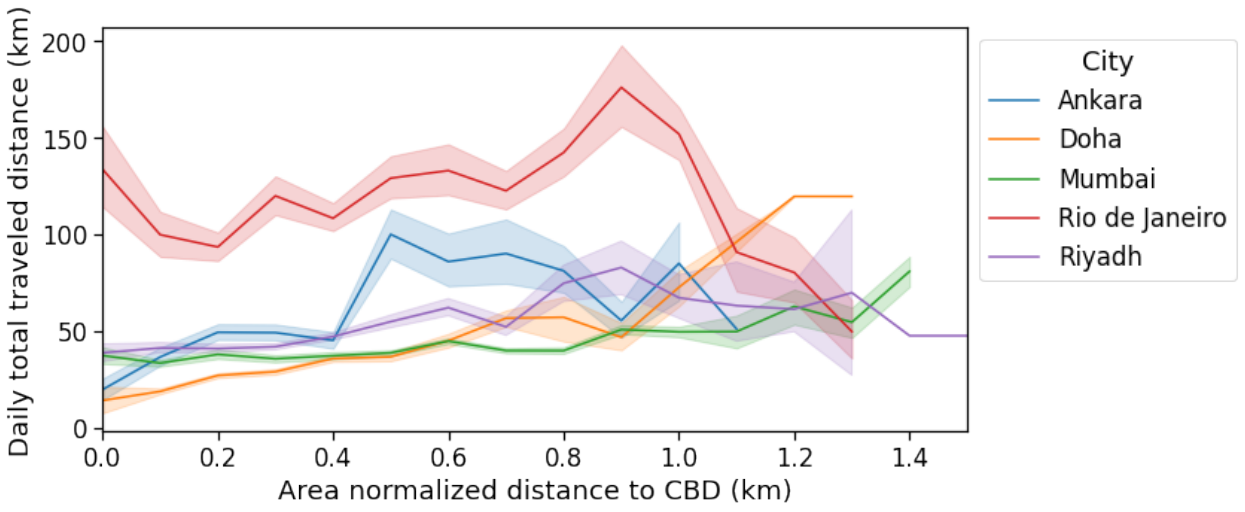
Total Traveled Distance (TTD) vs. distance to Central Business District

For this and the next indicators we observe the same behavior. We simply reports all the plots to display the insights gained with the spatial structure analyses of the previous notebook.

[62]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
    y='ttd_mean',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel='Daily total traveled distance (km)',
    # title=r'Per area daily $TTD$ vs. distance to CBD'
    xlim=[0,80],
)
fig.savefig(os.path.join(OUT_FOLDER, 'ttd_daily_area_cbd.pdf'), bbox_inches='tight')
```



[63]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin_norm',
    y='ttd_mean',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area normalized distance to CBD (km)',
    ylabel=r'Daily total traveled distance (km)',
    # title=r'Per area daily $TTD$ vs. normalized distance to CBD'
    xlim=[0,1.5],
)
fig.savefig(os.path.join(OUT_FOLDER, 'ttd_daily_area_cbd_norm.pdf'), bbox_inches='tight')
```

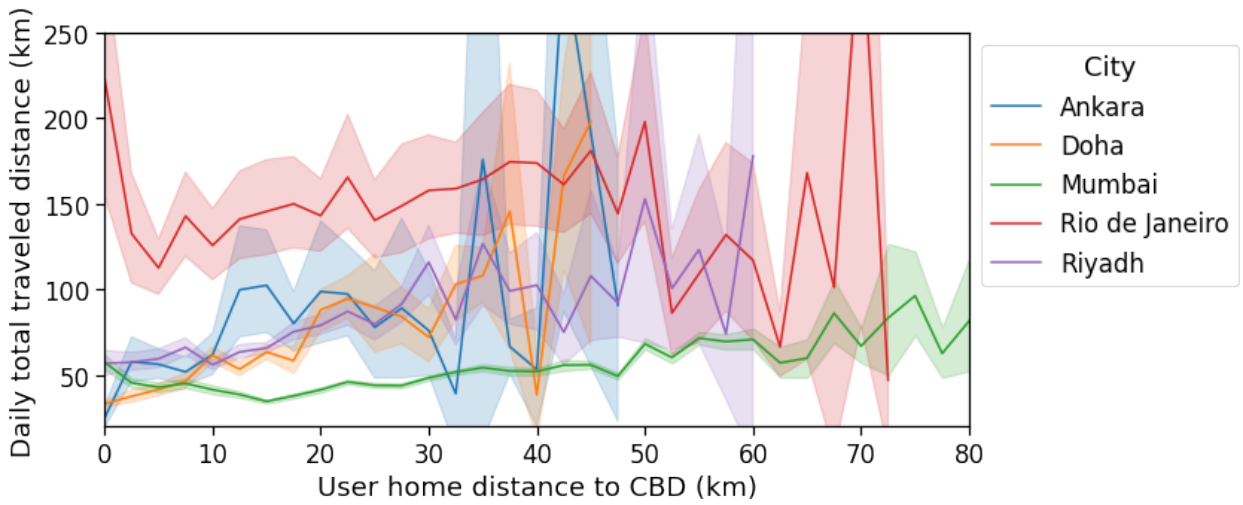


[37]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin',
    y='ttd',
    hue='City',
```

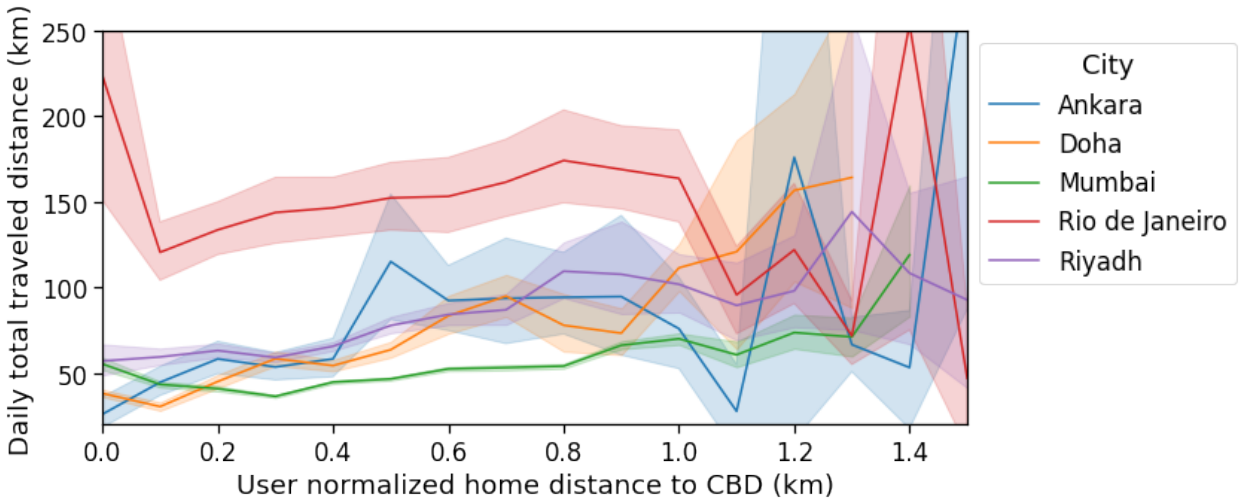
(continues on next page)

(continued from previous page)

```
    hue_order=city_todo_order,
    xlabel=r'User home distance to CBD (km)',
    ylabel=r'Daily total traveled distance (km)',
    # title=r'Per user $TTD$ vs. home distance to CBD',
    xlim=[0,80],
    ylim=(20,250),
)
fig.savefig(os.path.join(OUT_FOLDER, 'ttd_daily_user_cbd.pdf'), bbox_inches='tight')
```

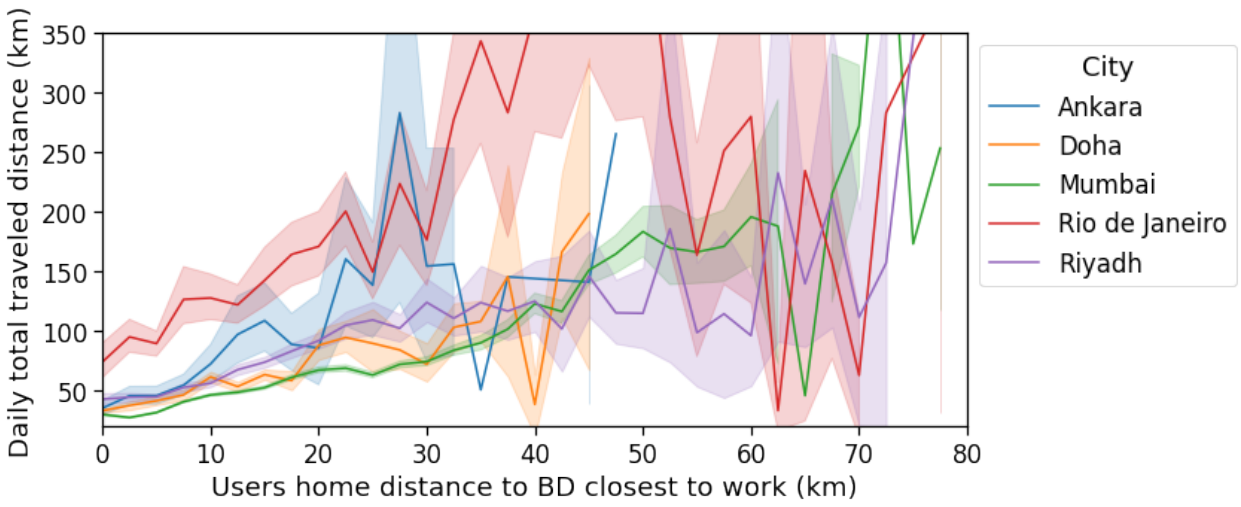


[38]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin_norm',
    y='ttd',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User normalized home distance to CBD (km)',
    ylabel=r'Daily total traveled distance (km)',
    # title=r'Per user $TTD$ vs. normalized home distance to CBD',
    xlim=[0,1.5],
    ylim=(20,250),
)
fig.savefig(os.path.join(OUT_FOLDER, 'ttd_daily_user_cbd_norm.pdf'), bbox_inches='tight')
```

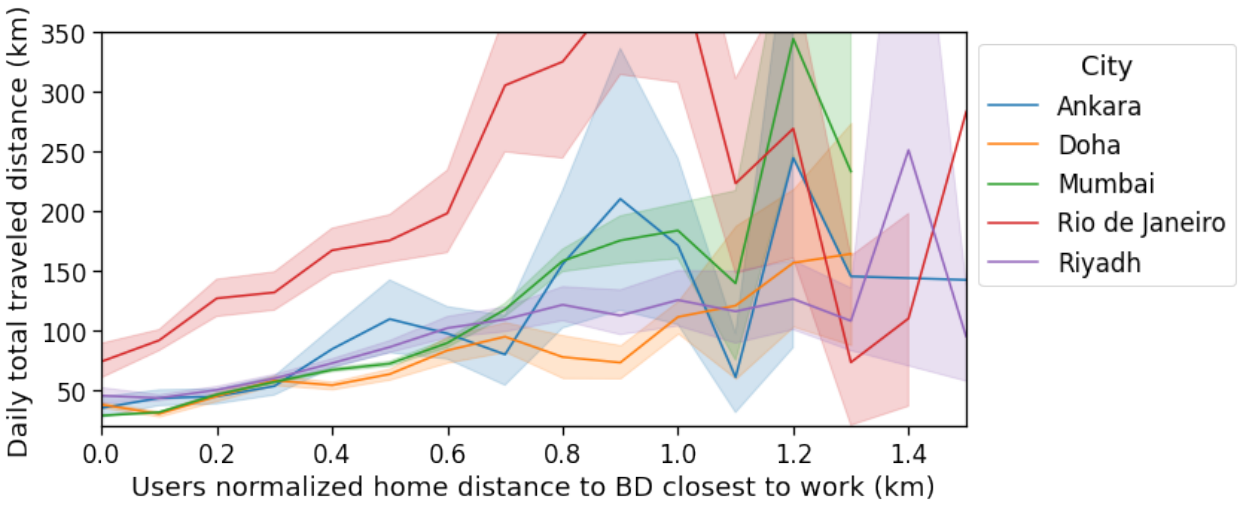


Total Traveled Distance (TTD) vs. distance to Business District closest to workplace

[39]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin',
    y='ttd',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Users home distance to BD closest to work (km)',
    ylabel=r'Daily total traveled distance (km)',
    # title=r'Per user daily $TTD$ vs. home distance to BD closest to workplace',
    xlim=[0,80],
    ylim=(20,350),
)
fig.savefig(os.path.join(OUT_FOLDER, 'ttd_daily_user_cbd_closest.pdf'), bbox_inches=
↪'tight')
```



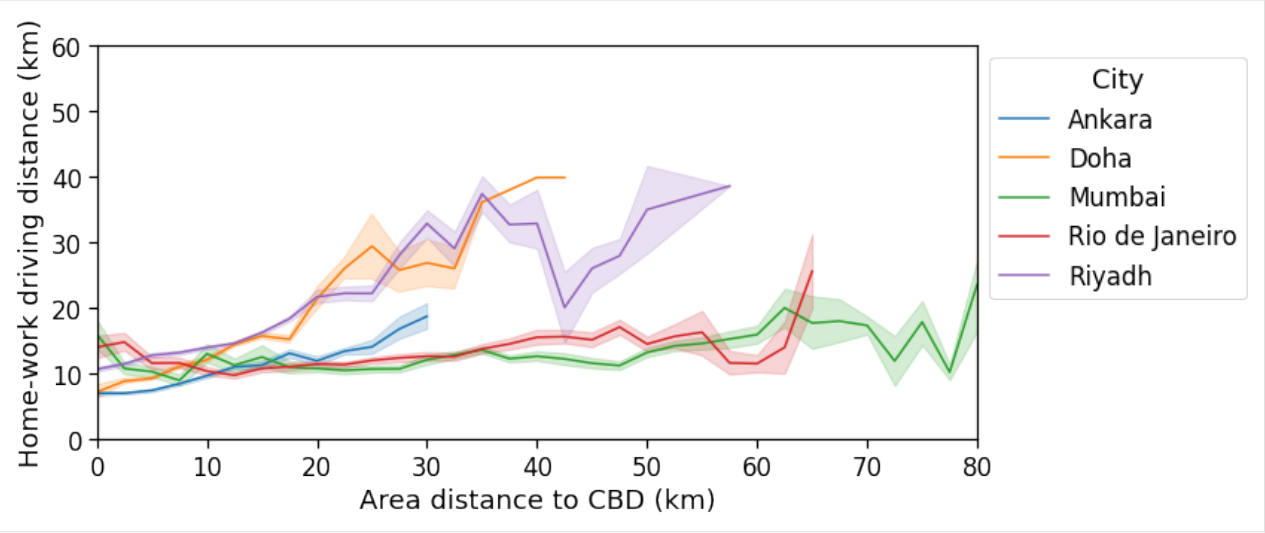
[40]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin_norm',
    y='ttd',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Users normalized home distance to BD closest to work (km)',
    ylabel=r'Daily total traveled distance (km)',
    # title=r'Per user daily $TTD$ vs. normalized home distance to BD closest to workplace
↪',
    xlim=[0,1.5],
    ylim=(20,350),
)
fig.savefig(os.path.join(OUT_FOLDER, 'ttd_daily_user_cbd_closest_norm.pdf'), bbox_inches=
↪'tight')
```



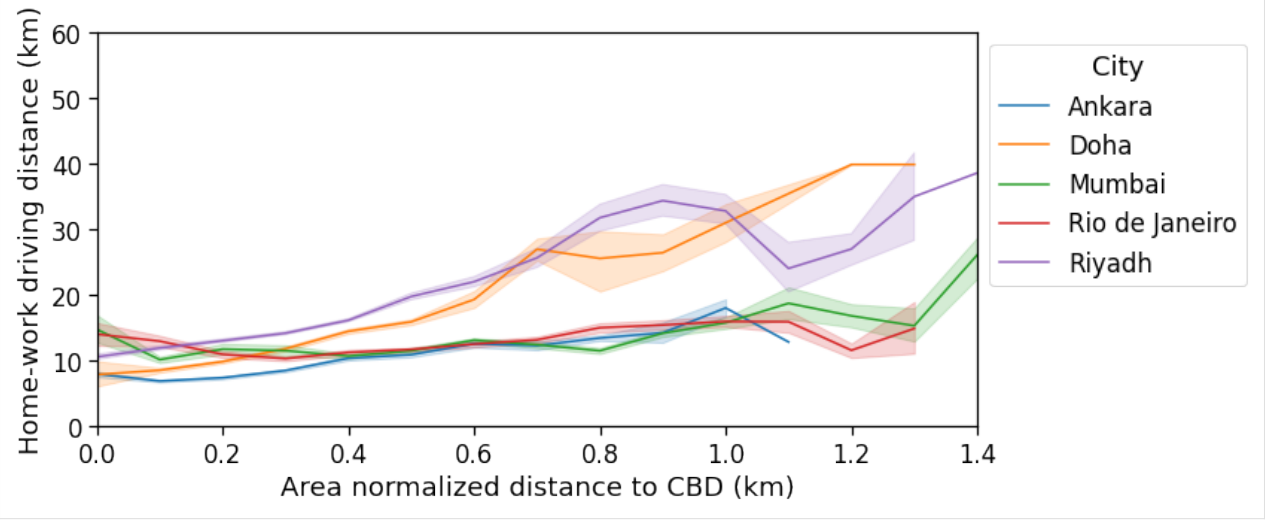
Home-work distance vs. distance to Central Business District

[41]:
```
df_spatial_areas['home_work_osrm_dist_avg_km'] = df_spatial_areas['home_work_osrm_dist_
↪avg'] / 1e3
```

[42]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
    y='home_work_osrm_dist_avg_km',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel=r'Home-work driving distance (km)',
    # title=r'Per area $D^{OSRM}_{home,work}$ vs. normalized distance to CBD',
    xlim=[0,80],
    ylim=(0,60),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_dist_area_cbd.pdf'), bbox_inches='tight')
```



[43]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin_norm',
    y='home_work_osrm_dist_avg_km',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area normalized distance to CBD (km)',
    ylabel=r'Home-work driving distance (km)',
    # title=r'Per area $D^{OSRM}_{home,work}$ vs. normalized distance to CBD',
    xlim=[0,1.4],
    ylim=(0,60),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_dist_area_cbd_norm.pdf'), bbox_inches=
↪'tight')
```

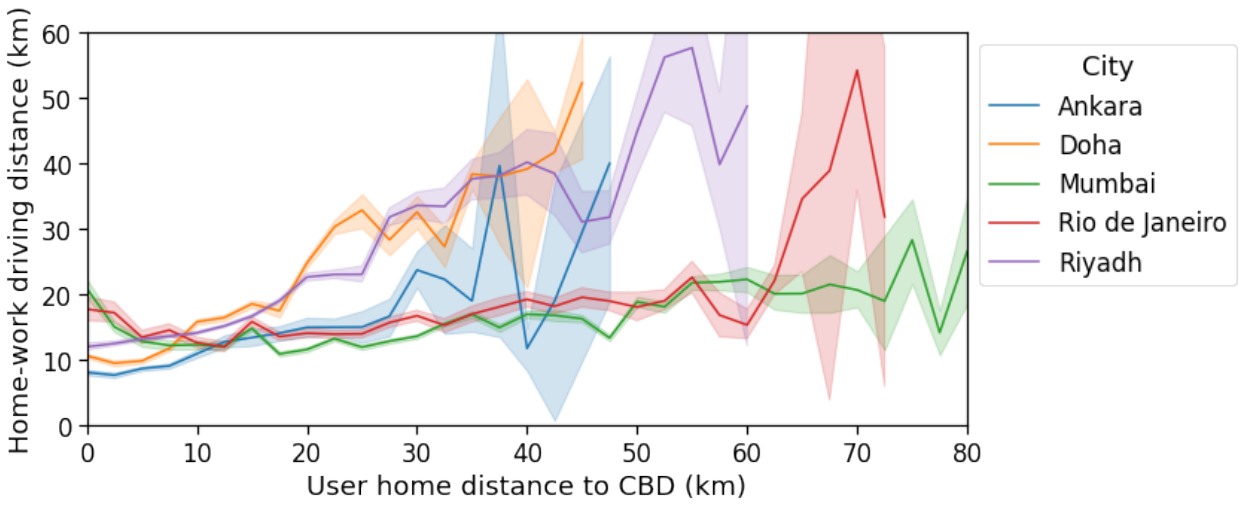


[44]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin',
```

(continues on next page)

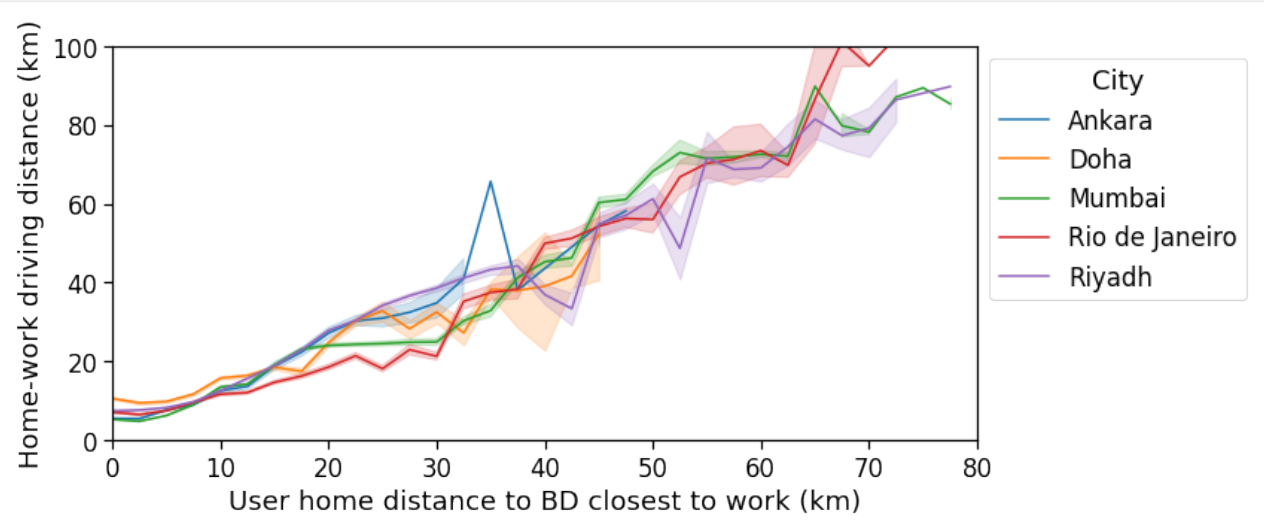
(continued from previous page)

```
    y='home_work_osrm_dist_km',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to CBD (km)',
    ylabel=r'Home-work driving distance (km)',
    # title=r'Per user $D^{OSRM}_{home,work}$ vs. distance to CBD',
    xlim=[0,80],
    ylim=(0,60),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_dist_user_cbd.pdf'), bbox_inches='tight')
```



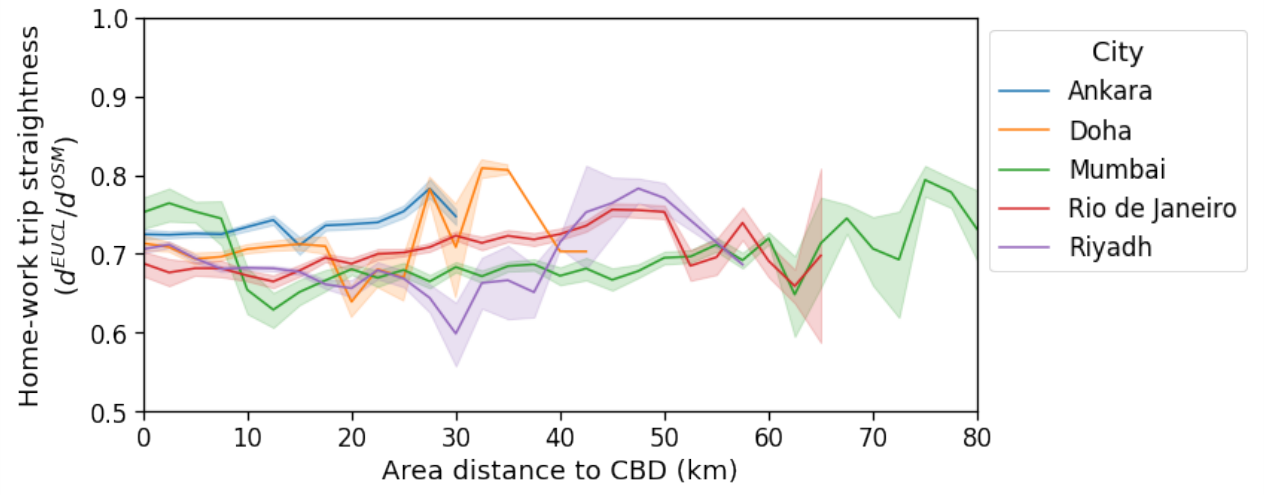
Home-work distance vs. distance to Business District closest to homeplace

[45]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin',
    y='home_work_osrm_dist_km',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to BD closest to work (km)',
    ylabel=r'Home-work driving distance (km)',
    # title=r'Per user $D^{OSRM}_{home,work}$ vs. home distance to BD closest to workplace
↪',
    xlim=[0,80],
    ylim=(0,100),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_dist_user_cbd_closest.pdf'), bbox_inches=
↪'tight')
```

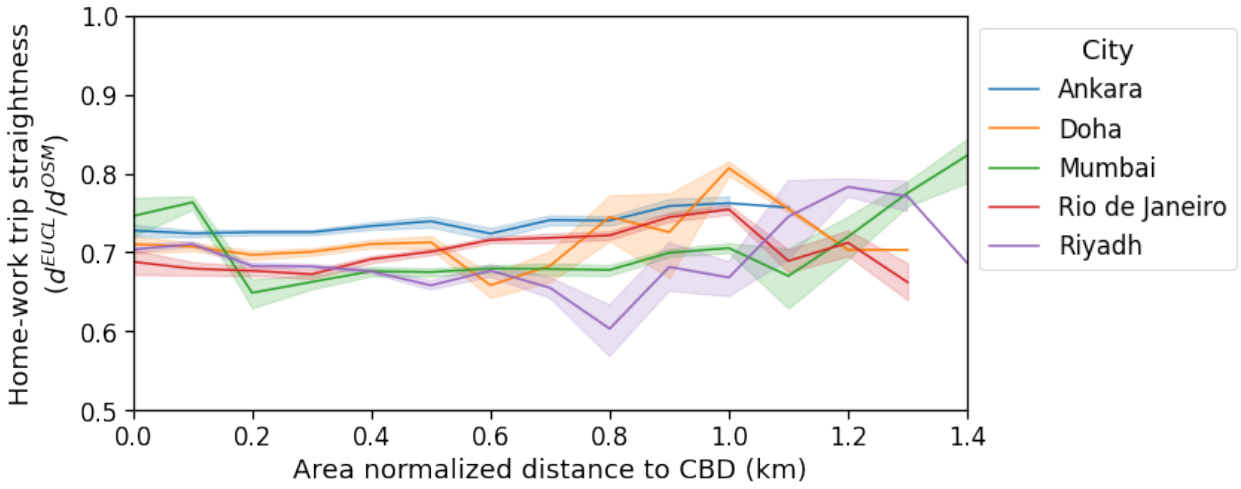


Home-work distance ratio (Euclidean/osrm) vs. distance to Central Business District

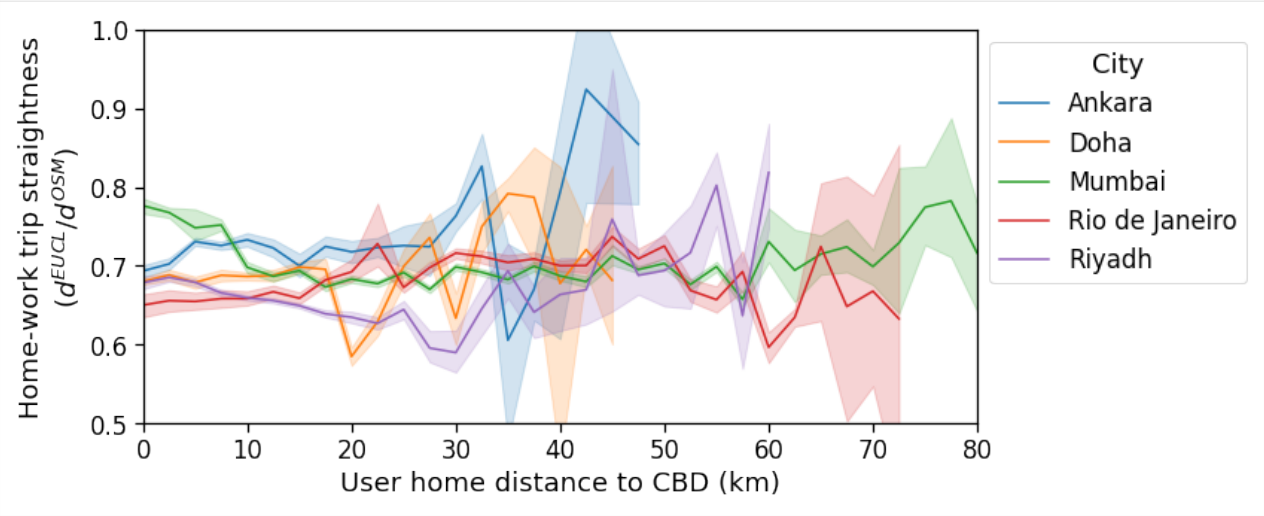
[46]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
    y='avg_realD_frac_osmD',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel='Home-work trip straightness\n' + r'$(d^{EUCL} / d^{OSM})$',
    # title=r'Per area home-work $D^{EUCL} / D^{OSRM}$ distance vs. norm. distance to CBD
↪',
    xlim=[0,80],
    ylim=(.5,1.),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_dist_fraction_area_cbd.pdf'), bbox_
↪inches='tight')
```



[47]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin_norm',
    y='avg_realD_frac_osmD',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area normalized distance to CBD (km)',
    ylabel='Home-work trip straightness\n' + r'$(d^{EUCL} / d^{OSM})$',
    # title=r'Per area home-work $D^{EUCL} / D^{OSRM}$ distance vs. norm. distance to CBD
↪',
    xlim=[0,1.4],
    ylim=(.5,1.),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_dist_fraction_area_cbd_norm.pdf'), bbox_
↪inches='tight')
```

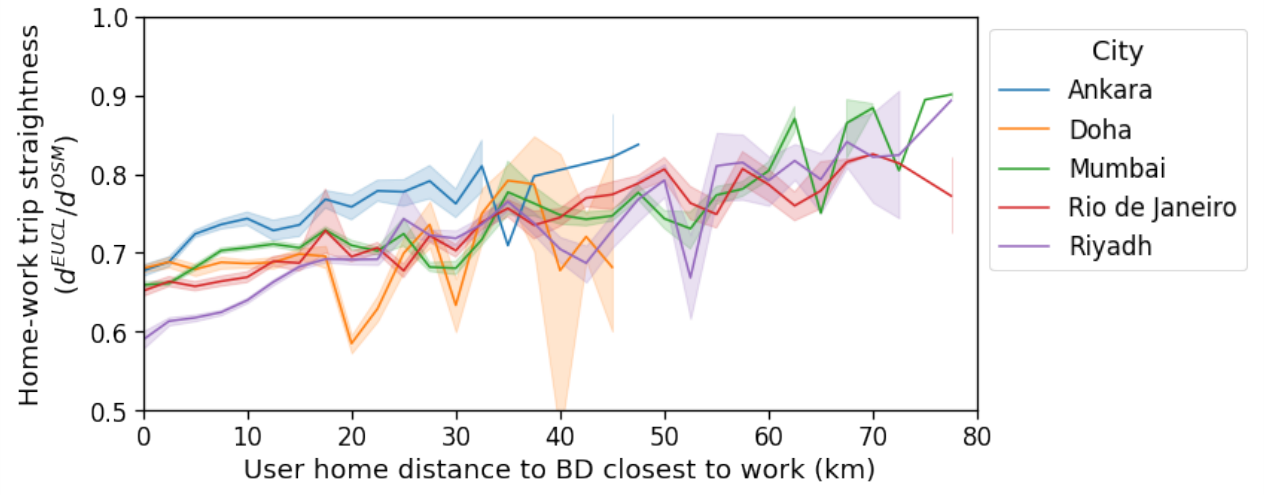


[48]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin',
    y='avg_realD_frac_osmD',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to CBD (km)',
    ylabel='Home-work trip straightness\n' + r'$(d^{EUCL} / d^{OSM})$',
    # title=r'Per user home-work $D^{EUCL} / D^{OSRM}$ distance vs. home distance to CBD',
    xlim=[0,80],
    ylim=(.5,1.),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_dist_fraction_user_cbd.pdf'), bbox_
↪inches='tight')
```



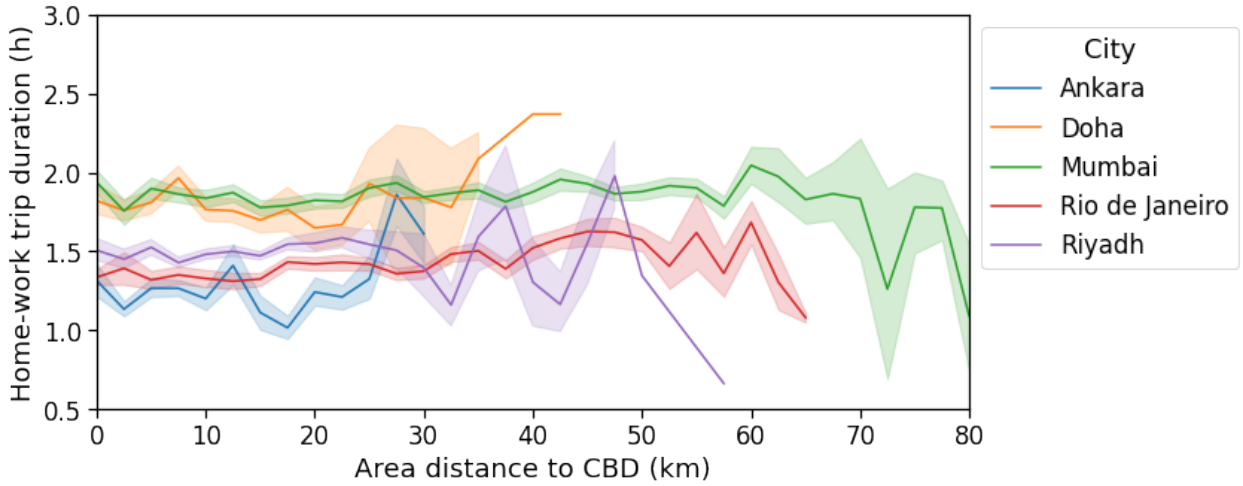
Home-work distance ratio (Euclidean/osrm) vs. distance to Business District closest to workplace

[49]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin',
    y='avg_realD_frac_osmD',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to BD closest to work (km)',
    ylabel='Home-work trip straightness\n' + r'$(d^{EUCL} / d^{OSM})$',
    # title=r'Per user home-work $D^{EUCL} / D^{OSRM}$ dist. vs. home dist. to BD closest to work',
    xlim=(0,80),
    ylim=(.5,1),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_dist_fraction_user_cbd_closest.pdf'), bbox_inches='tight')
```

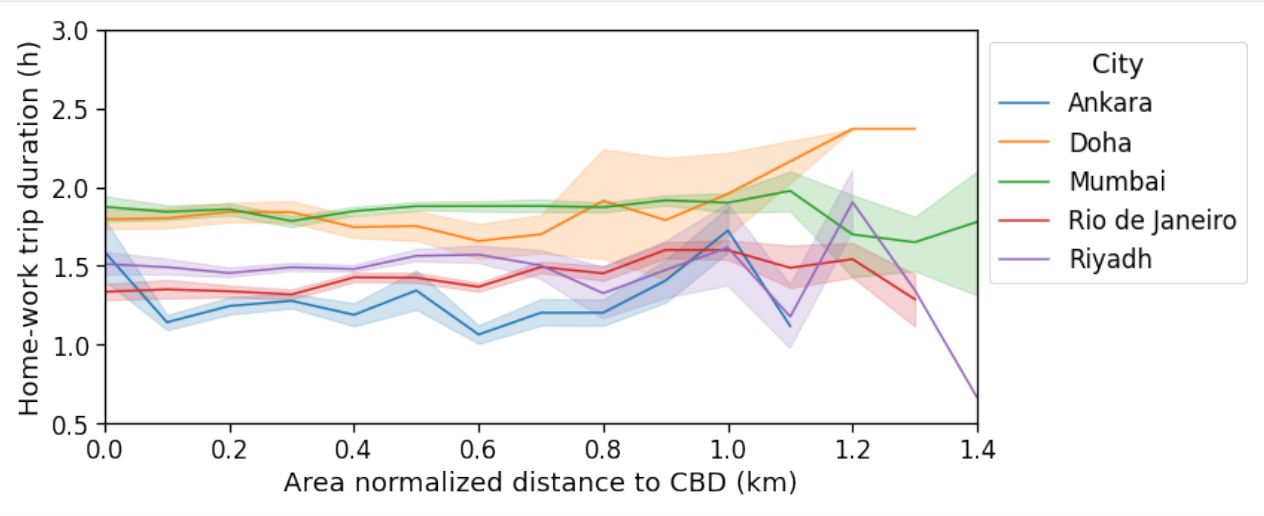


Home-work duration vs. distance to Central Business District

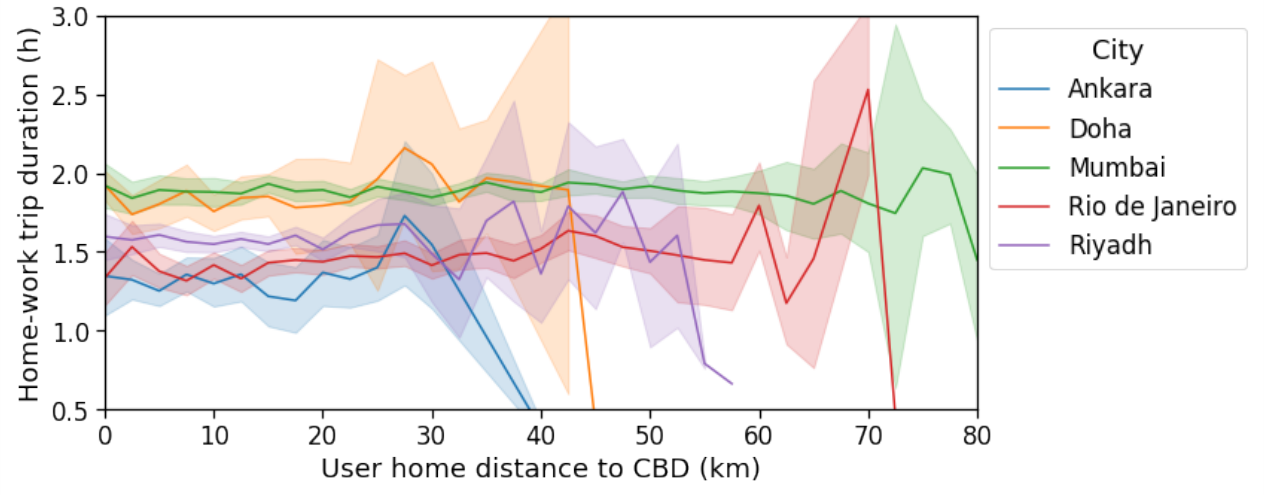
[50]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
    y='time_trips_hw_avg',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel='Home-work trip duration (h)',
    # title=r'Per area home-work $t^{REAL}$ duration vs. norm. distance to CBD',
    xlim=(0,80),
    ylim=(.5,3),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_duration_real_area_cbd.pdf'), bbox_
↪inches='tight')
```



[51]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin_norm',
    y='time_trips_hw_avg',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area normalized distance to CBD (km)',
    ylabel='Home-work trip duration (h)',
    # title=r'Per area home-work $t^{REAL}$ duration vs. norm. distance to CBD',
    xlim=(0,1.4),
    ylim=(.5,3),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_duration_real_area_cbd_norm.pdf'), bbox_
↪inches='tight')
```

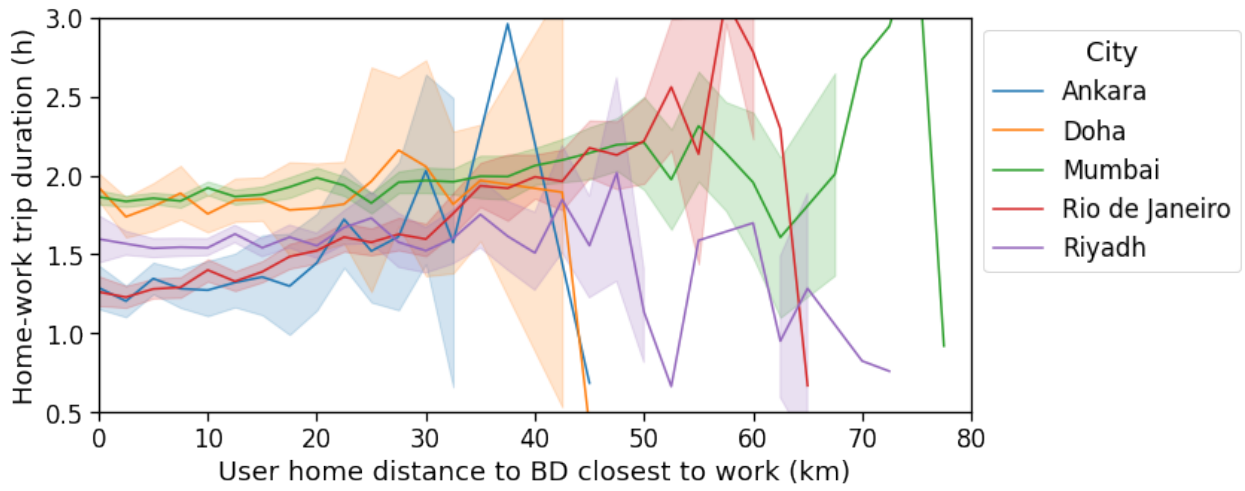


[52]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin',
    y='time_trips_hw',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to CBD (km)',
    ylabel='Home-work trip duration (h)',
    # title=r'Per user home-work $t^{REAL}$ duration vs. home distance to CBD',
    xlim=(0,80),
    ylim=(.5,3),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_duration_real_user_cbd.pdf'), bbox_
↪inches='tight')
```



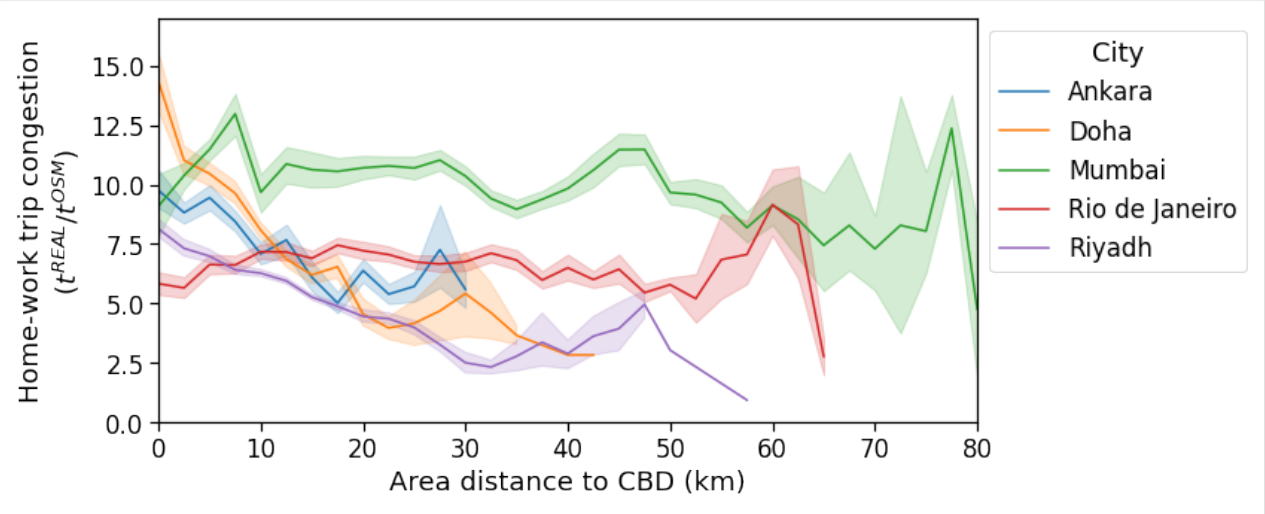
Home-work duration vs. distance to Business District closest to workplace

[53]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin',
    y='time_trips_hw',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to BD closest to work (km)',
    ylabel='Home-work trip duration (h)',
    #  title=r'Per user home-work $t^{REAL}$ duration vs. home dist. to BD closest to␣
↪workplace',
    xlim=(0,80),
    ylim=(.5,3),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_duration_real_user_cbd_closest.pdf'),␣
↪bbox_inches='tight')
```

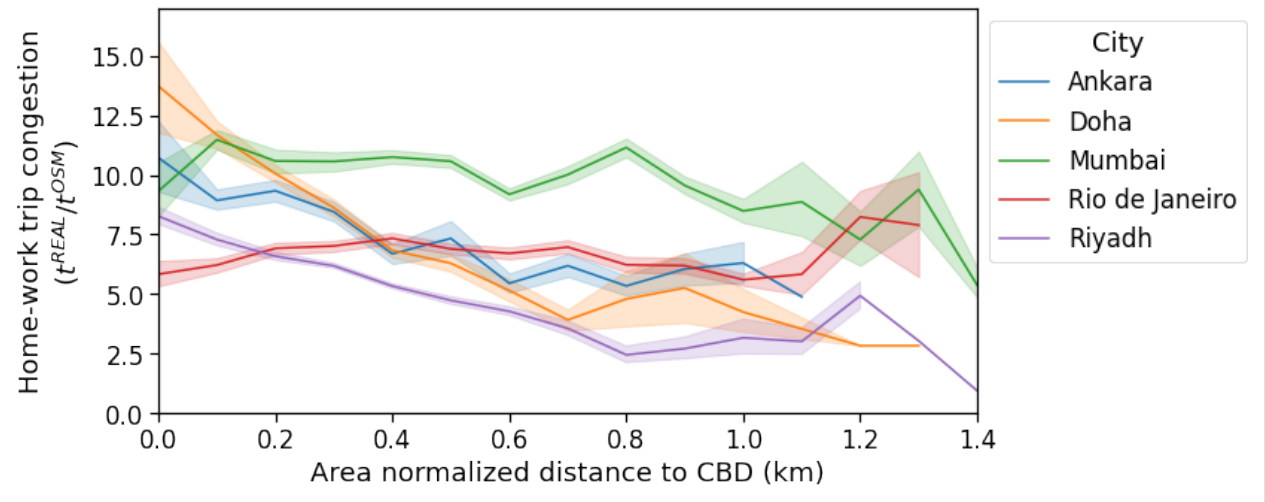


Home-work duration ratio (real/osrm) vs. distance to Central Business District

[54]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
    y='avg_realT_frac_osmT',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel='Home-work trip congestion\n' + r'($t^{REAL} / t^{OSM}$)',
    # title=r'Per area home-work $t^{REAL} / t^{OSRM}$ duration vs. norm. distance to CBD
↪',
    xlim=(0,80),
    ylim=(0,17),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_duration_fraction_area_cbd.pdf'), bbox_
↪inches='tight')
```



[55]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin_norm',
    y='avg_realT_frac_osmT',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area normalized distance to CBD (km)',
    ylabel='Home-work trip congestion\n' + r'($t^{REAL} / t^{OSM}$)',
    # title=r'Per area home-work $t^{REAL} / t^{OSRM}$ duration vs. norm. distance to CBD
↪',
    xlim=(0,1.4),
    ylim=(0,17),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_duration_fraction_area_cbd_norm.pdf'),␣
↪bbox_inches='tight')
```

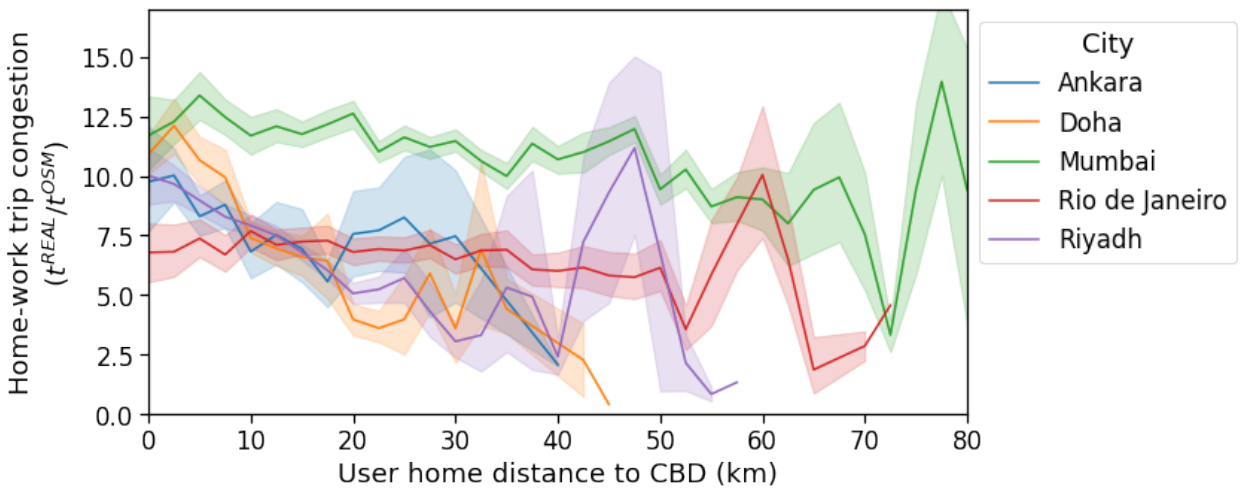


[56]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin',
```

(continues on next page)

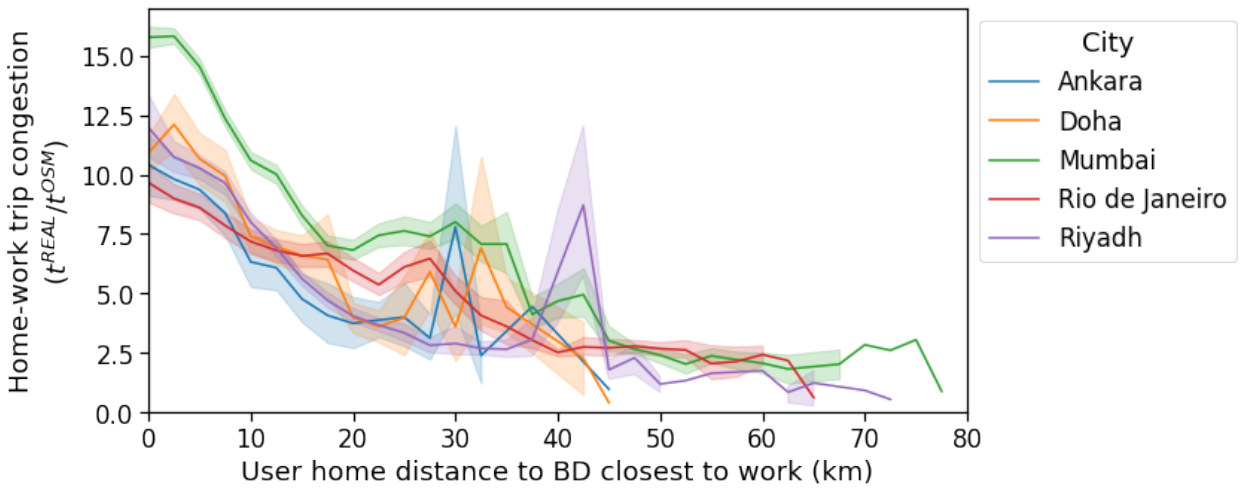
(continued from previous page)

```
    y='avg_realT_frac_osmT',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to CBD (km)',
    ylabel='Home-work trip congestion\n' + r'($t^{REAL} / t^{OSM}$)',
    # title=r'Per user home-work $t^{REAL} / t^{OSRM}$ duration vs. home distance to CBD',
    xlim=(0,80),
    ylim=(0,17),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_duration_fraction_user_cbd.pdf'), bbox_
↪inches='tight')
```



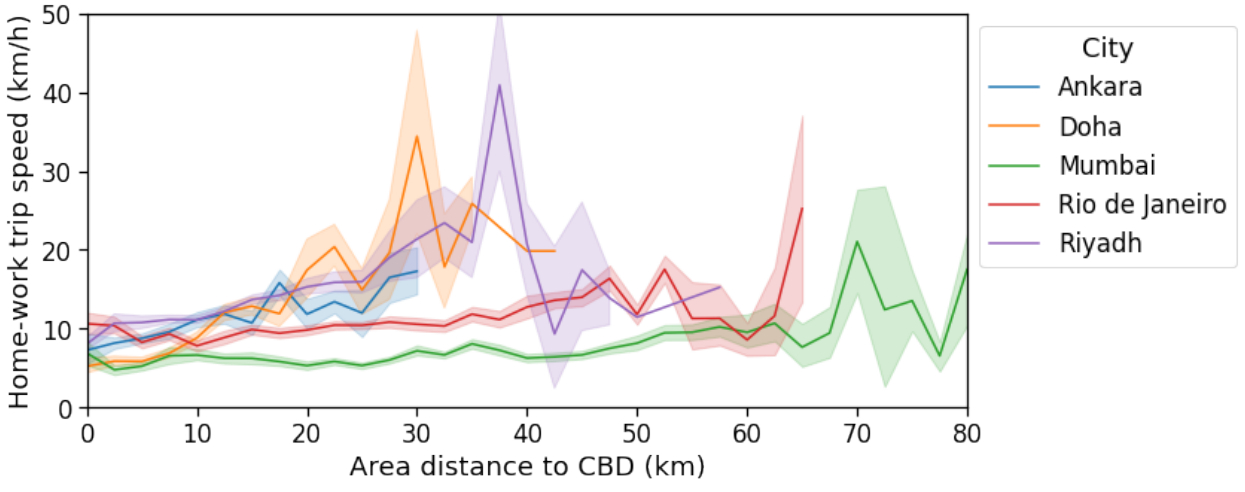
Home-work duration ratio (real/osrm) vs. distance to Business District closest to workplace

[57]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin',
    y='avg_realT_frac_osmT',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to BD closest to work (km)',
    ylabel='Home-work trip congestion\n' + r'($t^{REAL} / t^{OSM}$)',
    #  title=r'Per user home-work $t^{REAL} / t^{OSRM}$ duration vs. home dist. to BD␣
↪closest to workplace',
    xlim=(0,80),
    ylim=(0,17),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_duration_fraction_user_cbd_closest.pdf'),
↪ bbox_inches='tight')
```

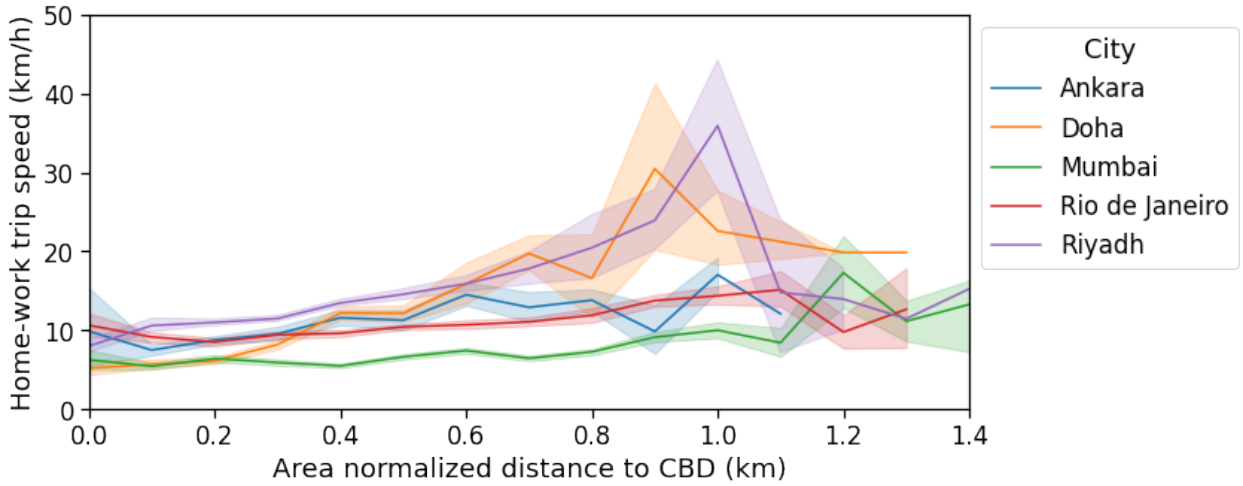


Home-work travel speed vs. distance to Central Business District

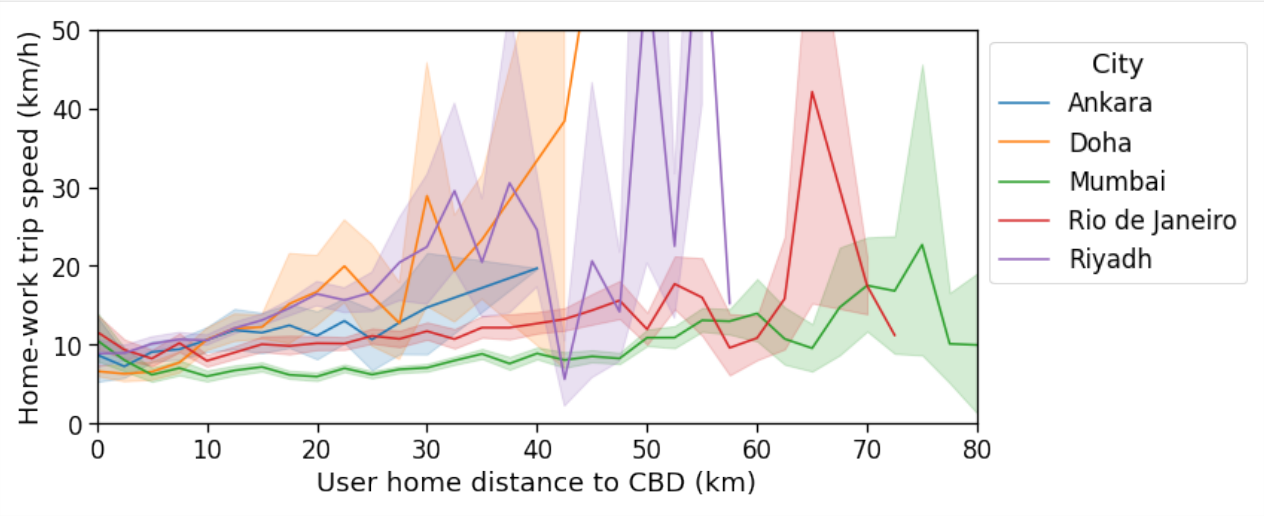
[58]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin',
    y='speed_trips_hw_avg',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area distance to CBD (km)',
    ylabel=r'Home-work trip speed (km/h)',
    # title=r'Per area home-work $v^{REAL}$ speed vs. norm. distance to CBD',
    xlim=(0,80),
    ylim=(0,50),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_speed_real_area_cbd.pdf'), bbox_inches=
↪'tight')
```



[59]:
```
fig, ax = plotLines(
    data=df_spatial_areas,
    x='cbd_dist_bin_norm',
    y='speed_trips_hw_avg',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'Area normalized distance to CBD (km)',
    ylabel=r'Home-work trip speed (km/h)',
    # title=r'Per area home-work $v^{REAL}$ speed vs. norm. distance to CBD',
    xlim=(0,1.4),
    ylim=(0,50),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_speed_real_area_cbd_norm.pdf'), bbox_
↪inches='tight')
```

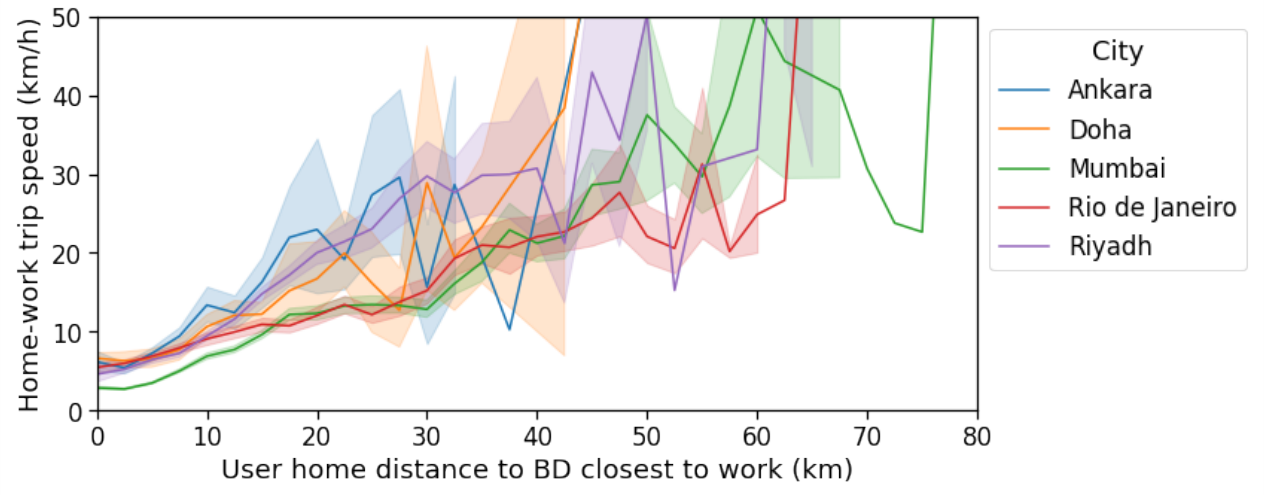


[60]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='cbd_dist_bin',
    y='speed_trips_hw',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to CBD (km)',
    ylabel=r'Home-work trip speed (km/h)',
    # title=r'Per user home-work $v^{REAL}$ speed vs. home distance to CBD',
    xlim=(0,80),
    ylim=(0,50),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_speed_real_user_cbd.pdf'), bbox_inches=
↪'tight')
```



Home-work travel speed vs. distance to Business District closest to workplace

[61]:
```
fig, ax = plotLines(
    data=df_spatial_users,
    x='closest_cbd_dist_bin',
    y='speed_trips_hw',
    hue='City',
    hue_order=city_todo_order,
    xlabel=r'User home distance to BD closest to work (km)',
    ylabel=r'Home-work trip speed (km/h)',
    # title=r'Per user home-work $v^{REAL}$ speed vs. home dist. to BD closest to␣
↪workplace',
    xlim=(0,80),
    ylim=(0,50),
)
fig.savefig(os.path.join(OUT_FOLDER, 'home_work_speed_real_user_cbd_closest.pdf'), bbox_
↪inches='tight')
```



[ ]:

6.8 Loading data

load_raw_files(pattern[, version, timezone, ...])	Function that loads the files and returns the dataframe.
load_from_skmob(df[, uid, npartitions])	Loads a dataframe imported with skmobility and returns a dask dataframe.
dask_to_skmob(df, **kwargs)	Ports a dataframe from dask to skmob.

loader.py contains a set of tools to load and prepare the database from raw files.

mobilkit.loader.**dask_to_skmob**(*df*, ***kwargs*)

Ports a dataframe from dask to skmob. Given the structure of skmob it is done only to a *skmob.TrajDataFrame*.

Parameters

- **df** (*dask.dataframe*) – A dask dataframe with at least the `uid`, `lat` and `lng` columns.
- ****kwargs** – Will be passed to `skmob.TrajDataFrame`.

Returns

df_sp – A skmob.TrajDataFrame containing the input columns.

Return type

skmob.dataframe

mobilkit.loader.**load_from_skmob**(*df*, *uid='user'*, *npartitions=10*)

Loads a dataframe imported with skmobility and returns a dask dataframe.

Parameters

- **df** (*scikit-mobility.dataframe*) – A dataframe as imported from scikit-mobility. May already contains the `tile_ID` and `uid` columns. If no `uid` column is found it will be initialized to the `uid` value.
- **uid** (*str, optional*) – The `uid` to be used, otherwise uses the present ones if the `uid` column is there.
- **npartitions** (*int, optional*) – The number of partition for the dataframe to be split into.

Returns

df_sp – A *dask.dataframe* containing the input columns plus the accuracy `acc` (with dummy 1 value) and possibly the `uid` one if it was missing.

Return type

dask.dataframe

mobilkit.loader.**load_raw_files**(*pattern*, *version='hflb'*, *timezone=None*, *start_date=None*, *stop_date=None*, *minAcc=300*, *sep='\t'*, *file_schema=None*, ***kwargs*)

Function that loads the files and returns the dataframe.

Parameters

- **pattern** (*str*) – The pattern of the raw files with bash syntax. For example: `'sample_data/20*/part-*.csv.gz'`.
- **version** (*str, optional*) – One of *hflb*, *wb* or *csv*, the format in which data are stored.

- **timezone** (*str, optional*) – The timezone in pytz syntax (e.g., "Europe/Rome" or "America/Mexico_City") to be used to localize the Unix Time Stamp time-stamp in the raw-data. If no timezone is specified (default) it defaults to UTC.
- **start_date** (*str, optional*) – The starting date when to consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given.
- **stop_date** (*str, optional*) – The end day up to which consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given. This day will be INCLUDED.
- **minAcc** (*int, optional*) – The minimum accuracy for a point to be kept. If accuracy is larger than this the point will be discarded. **NOTE that the accuracy column must be called** `acc`.
- **sep** (*str, optional*) – The delimiter of the fields in the files.
- **file_schema** (*list of tuples*) – The schema of the file. By default will be `dask_schemas.eventLineRAW`. If *version=wb file_schema* is a dictionary telling how to translate the original colums in the mobilkit nomenclature. **NOTE that the accuracy column must be called** `acc`.
- ****kwargs** – Will be passed to *mobilkit.loader.load_raw_files_hflb* if *version='hflb'* otherwise to *mobilkit.loader.load_raw_files_wb* if *version='wb'*.

Returns
df – A representation of the dataframe.

Return type
dask.dataframe

mobilkit.loader.**load_raw_files_custom**(*pattern*, *timezone=None*, *start_date=None*, *stop_date=None*, *minAcc=300*, *sep='\t'*, *file_schema=None*, *partition_size=None*, ***kwargs*)

Function that loads the files and returns the dask dataframe. Note that this function is **lazy** meaning that it only construct the dataframe and does not build it (it will be built the first time a query is performed on it).

Parameters

- **pattern** (*str*) – The pattern of the raw files with bash syntax. For example: `'sample_data/20*/part-*.csv.gz'`.
- **timezone** (*str, optional*) – The timezone in pytz syntax (e.g., "Europe/Rome" or "America/Mexico_City") to be used to localize the Unix Time Stamp time-stamp in the raw-data. If no timezone is specified (default) it defaults to UTC.
- **start_date** (*str, optional*) – The starting date when to consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given.
- **stop_date** (*str, optional*) – The end day up to which consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given. This day will be INCLUDED.
- **minAcc** (*int, optional*) – The minimum accuracy for a point to be kept. If accuracy is larger than this the point will be discarded. **NOTE that the accuracy column must be called** `acc`.
- **sep** (*str, optional*) – The delimiter of the fields in the files.
- **file_schema** (*list of tuples*) – The schema of the file. By default will be `dask_schemas.eventLineRAW`. **NOTE that the accuracy column must be called** `acc`.

Returns
df – A representation of the dataframe.

Return type
dask.dataframe

mobilkit.loader.**load_raw_files_wb**(*pattern*, *timezone=None*, *header=False*, *start_date=None*, *stop_date=None*, *minAcc=300*, *sep='\t'*, *file_schema=None*, ***kwargs*)

Function that loads the files and returns the dask dataframe. Note that this function is **lazy** meaning that it only construct the dataframe and does not build it (it will be built the first time a query is performed on it).

Parameters

- **pattern** (*str*) – The pattern of the raw files with bash syntax. For example: `'sample_data/20*/part-*.csv.gz'`.
- **timezone** (*str, optional*) – The timezone in pytz syntax (e.g., "Europe/Rome" or "America/Mexico_City") to be used to localize the Unix Time Stamp time-stamp in the raw-data. If no timezone is specified (default) it defaults to UTC.
- **start_date** (*str, optional*) – The starting date when to consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given.
- **stop_date** (*str, optional*) – The end day up to which consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given. This day will be INCLUDED.
- **minAcc** (*int, optional*) – The minimum accuracy for a point to be kept. If accuracy is larger than this the point will be discarded. **NOTE that the accuracy column must be called** `acc`.
- **sep** (*str, optional*) – The delimiter of the fields in the files.
- **file_schema** (*list of tuples*) – The dict to rename the original columns to the mobilkit ones. **NOTE that the accuracy column must be called** `acc`.

Returns
df – A representation of the dataframe.

Return type
dask.dataframe

6.9 mobilkit package

6.9.1 Submodules

mobilkit.dask_schemas module

The typical types of the dataframe being processed and the fixed column names.

mobilkit.dask_schemas.**eventLineDT = [('UTC', <class 'int'>), ('uid', <class 'str'>), ('OS', <class 'int'>), ('lat', <class 'float'>), ('lng', <class 'float'>), ('acc', <class 'int'>), ('timezone', <class 'int'>), ('datetime', <class 'datetime.datetime'>)]**

The default type for raw files with datetime added. It is composed by:

- *UTC* : unix time stamp (int)
- *UID* : unique user identifier (string)
- *OS* : The os of the user (int)
- *lat* : latitude (float)
- *lon* : longitude (float)
- *acc* : accuracy (float)
- *tz* : time zone offset (float)

- *date* : datetime (datetime)

mobilkit.dask_schemas.**eventLineDTzone = [('UTC', <class 'int'>), ('uid', <class 'str'>), ('OS', <class 'int'>), ('lat', <class 'float'>), ('lng', <class 'float'>), ('acc', <class 'int'>), ('timezone', <class 'int'>), ('datetime', <class 'datetime.datetime'>), ('tile_ID', <class 'int'>)]**

The default type for raw files with datetime and zone index (int) added. It is composed by:

- *UTC* : unix time stamp (int)
- *UID* : unique user identifier (string)
- *OS* : The os of the user (int)
- *lat* : latitude (float)
- *lon* : longitude (float)
- *acc* : accuracy (float)
- *tz* : time zone offset (float)
- *date* : datetime (datetime)
- *ZONE_IDX* : the index of the containing area (int, -1 if outside of shapefile)

mobilkit.dask_schemas.**eventLineRAW = [('UTC', <class 'int'>), ('uid', <class 'str'>), ('OS', <class 'int'>), ('lat', <class 'float'>), ('lng', <class 'float'>), ('acc', <class 'int'>), ('timezone', <class 'int'>)]**

The default type for raw files. It is composed by:

- *UTC* : unix time stamp (int)
- *uid* : unique user identifier (string)
- *OS* : The os of the user (int)
- *lat* : latitude (float)
- *lng* : longitude (float)
- *acc* : accuracy (float)
- *timezone* : time zone offset (float)

mobilkit.dask_schemas.**eventLineZone = [('UTC', <class 'int'>), ('uid', <class 'str'>), ('OS', <class 'float'>), ('lat', <class 'float'>), ('lng', <class 'float'>), ('acc', <class 'float'>), ('timezone', <class 'float'>), ('tile_ID', <class 'int'>)]**

The default type for raw files with zone index (int) added. It is composed by:

- *UTC* : unix time stamp (int)
- *UID* : unique user identifier (string)
- *OS* : The os of the user (int)
- *lat* : latitude (float)
- *lon* : longitude (float)
- *acc* : accuracy (float)
- *tz* : time zone offset (float)
- *ZONE_IDX* : the index of the containing area (int, -1 if outside of shapefile)

mobilkit.displacement module

Tools and functions to analyze the displacement based on pings rather than their localized tile IDs.

mobilkit.displacement.**calc_displacement**(*df_usr_day*, *df_usr_home*)

Given all the events of a (user,date) it computes the average and min distance from home, the closest point to home plus the original home location of the user on that day.

Parameters

df_usr_day (*dask.DataFrame*) – A dataframe containing all the events as returned by *mobilkit.temporal.filter_daynight_time*.

df_usr_home

[pd.DataFrame or dask.DataFrame] A *uid,homelat,homelon* df containing the lat and lon of the user's home to be joined with *df_usr_day*. Note that only events belonging to *uid* contained in this df will be considered.

Returns

df – A df containing, for each *uid,date* couple observed:

- *mindist,avgdist* the minimum and average distance from home recorded on that *date*
- *lat,lng* coordinates of the closest point to home recorded on *date*
- *homelat,homelon* the user's home coordinates
- ***rg,ttd* the radius of gyration and the total traveled distance for a** user on *date*.

Return type

dask.DataFrame

mobilkit.displacement.**process_user_day_displacement_pings**(*g*)

Given all the events of a (user,date) it computes the average and min distance from home, the closest point to home plus the original home location of the user.

Parameters

g (*grouped df or pd.DataFrame*) – A group of all the events of a user *uid* recorded in a day *date* or dataframe as returned by *mobilkit.temporal.filter_daynight_time* and joined with a *uid,homelat,homelon* df with at least the *uid,date,homelat,homelon,lat,lng*

Returns

df – A one-row df containing the original *uid,date,homelat,homelon* columns plus the minimum and average distance from home *mindist,avgdist* recorded on that *date* plus the *lat,lng* coordinates of the closest point to home.

Return type

pd.DataFrame

mobilkit.displacement.**process_user_displacement_pings**(*g*)

Given all the events of a user *uid* it computes the average and min distance from home, the closest point to home plus the original home location of the user for each *date* observed.

Parameters

g (*grouped df or pd.DataFrame*) – A group of all the events of a user *uid* recorded in all the *dates* or dataframe as returned by *mobilkit.temporal.filter_daynight_time* and joined with a *uid,homelat,homelon* df with at least the *uid,date,homelat,homelon,lat,lng*

Returns

df – A df containing the original *uid,homelat,homelon* columns plus the minimum and average

distance from home *mindist,avgdist* recorded on that *date* plus the *lat,lng* coordinates of the closest point to home for each observed day.

Return type
pd.DataFrame

mobilkit.loader module

loader.py contains a set of tools to load and prepare the database from raw files.

mobilkit.loader.**compute_datetime_col**(*df*, *selected_tz*)

mobilkit.loader.**crop_date**(*dff*, *startdt*, *enddt*, *timezone='America/Mexico_City'*)

mobilkit.loader.**crop_spatial**(*dff*, *bbox*)
Filters *dff* with a *box=[minlon,minlat,maxlon,maxlat]*.

mobilkit.loader.**crop_time**(*dff*, *nighttime_start*, *nighttime_end*, *timezone*)

mobilkit.loader.**dask_to_skmob**(*df*, ***kwargs*)
Ports a dataframe from dask to skmob. Given the structure of skmob it is done only to a *skmob.TrajDataFrame*.

Parameters

- **df** (*dask.dataframe*) – A dask dataframe with at least the `uid`, `lat` and `lng` columns.
- ****kwargs** – Will be passed to `skmob.TrajDataFrame`.

Returns
df_sp – A skmob.TrajDataFrame containing the input columns.

Return type
skmob.dataframe

mobilkit.loader.**filterStartStopDates**(*df*, *start_date*, *stop_date*, *tz*)

mobilkit.loader.**fromunix2date**(*x*, *timezone='America/Mexico_City'*)
Inherited from D4R.

mobilkit.loader.**fromunix2fulldate**(*x*, *timezone='America/Mexico_City'*)
Inherited from D4R.

mobilkit.loader.**fromunix2time**(*x*, *timezone='America/Mexico_City'*)
Inherited from D4R.

mobilkit.loader.**loadGeolifeData**(*path*, *acc_default=1*, *timezone='Asia/Shanghai'*)
Loads the *Geolife v1.3* trajectories with files ordered in the

GeolifeTrajectories1.3/Data/000/Trajectory/20090401202331.plt

structure with 6 useless rows at the beginning and the

lat,lng,0,altitude,days,date,time

format.

Parameters

- **path** (*str*) – The path to the root of the geolife data, usually called *data/GeolifeTrajectories1.3*.
- **acc_default** (*float*) – The default accuracy to give to each point to replicate the *mobilkit* format.

- **timezone** (*str*) – The code of the timezone the data has been recorded in.

Returns

df –

The dataframe containing the
uid,UTC,datetime,acc,lat,lng columns.

Return type
pd.DataFrame

mobilkit.loader.**load_from_skmob**(*df*, *uid='user'*, *npartitions=10*)

Loads a dataframe imported with skmobility and returns a dask dataframe.

Parameters

- **df** (*scikit-mobility.dataframe*) – A dataframe as imported from scikit-mobility. May already contains the `tile_ID` and `uid` columns. If no `uid` column is found it will be initialized to the `uid` value.
- **uid** (*str, optional*) – The `uid` to be used, otherwise uses the present ones if the `uid` column is there.
- **npartitions** (*int, optional*) – The number of partition for the dataframe to be split into.

Returns
df_sp – A *dask.dataframe* containing the input columns plus the accuracy `acc` (with dummy 1 value) and possibly the `uid` one if it was missing.

Return type
dask.dataframe

mobilkit.loader.**load_raw_files**(*pattern*, *version='hflb'*, *timezone=None*, *start_date=None*, *stop_date=None*, *minAcc=300*, *sep='\t'*, *file_schema=None*, ***kwargs*)

Function that loads the files and returns the dataframe.

Parameters

- **pattern** (*str*) – The pattern of the raw files with bash syntax. For example: `'sample_data/20*/part-*.csv.gz'`.
- **version** (*str, optional*) – One of *hflb*, *wb* or *csv*, the format in which data are stored.
- **timezone** (*str, optional*) – The timezone in pytz syntax (e.g., "Europe/Rome" or "America/Mexico_City") to be used to localize the Unix Time Stamp time-stamp in the raw-data. If no timezone is specified (default) it defaults to UTC.
- **start_date** (*str, optional*) – The starting date when to consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given.
- **stop_date** (*str, optional*) – The end day up to which consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given. This day will be INCLUDED.
- **minAcc** (*int, optional*) – The minimum accuracy for a point to be kept. If accuracy is larger than this the point will be discarded. **NOTE that the accuracy column must be called** `acc`.
- **sep** (*str, optional*) – The delimiter of the fields in the files.
- **file_schema** (*list of tuples*) – The schema of the file. By default will be `dask_schemas.eventLineRAW`. If *version=wb file_schema* is a dictionary telling how to translate the original colums in the mobilkit nomenclature. **NOTE that the accuracy column must be called** `acc`.

- ****kwargs** – Will be passed to *mobilkit.loader.load_raw_files_hflb* if *version='hflb'* otherwise to *mobilkit.loader.load_raw_files_wb* if *version='wb'*.

Returns
df – A representation of the dataframe.

Return type
dask.dataframe

mobilkit.loader.**load_raw_files_custom**(*pattern*, *timezone=None*, *start_date=None*, *stop_date=None*, *minAcc=300*, *sep='\t'*, *file_schema=None*, *partition_size=None*, ***kwargs*)

Function that loads the files and returns the dask dataframe. Note that this function is **lazy** meaning that it only construct the dataframe and does not build it (it will be built the first time a query is performed on it).

Parameters

- **pattern** (*str*) – The pattern of the raw files with bash syntax. For example: `'sample_data/20*/part-*.csv.gz'`.
- **timezone** (*str, optional*) – The timezone in pytz syntax (e.g., "Europe/Rome" or "America/Mexico_City") to be used to localize the Unix Time Stamp time-stamp in the raw-data. If no timezone is specified (default) it defaults to UTC.
- **start_date** (*str, optional*) – The starting date when to consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given.
- **stop_date** (*str, optional*) – The end day up to which consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given. This day will be INCLUDED.
- **minAcc** (*int, optional*) – The minimum accuracy for a point to be kept. If accuracy is larger than this the point will be discarded. **NOTE that the accuracy column must be called** `acc`.
- **sep** (*str, optional*) – The delimiter of the fields in the files.
- **file_schema** (*list of tuples*) – The schema of the file. By default will be `dask_schemas.eventLineRAW`. **NOTE that the accuracy column must be called** `acc`.

Returns
df – A representation of the dataframe.

Return type
dask.dataframe

mobilkit.loader.**load_raw_files_hflb**(*pattern*, *timezone=None*, *start_date=None*, *stop_date=None*, *minAcc=300*, *sep='\t'*, *file_schema=None*, *partition_size=None*)

Function that loads the files and returns the dask dataframe. Note that this function is **lazy** meaning that it only construct the dataframe and does not build it (it will be built the first time a query is performed on it).

Parameters

- **pattern** (*str*) – The pattern of the raw files with bash syntax. For example: `'sample_data/20*/part-*.csv.gz'`.
- **timezone** (*str, optional*) – The timezone in pytz syntax (e.g., "Europe/Rome" or "America/Mexico_City") to be used to localize the Unix Time Stamp time-stamp in the raw-data. If no timezone is specified (default) it defaults to UTC.
- **start_date** (*str, optional*) – The starting date when to consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given.
- **stop_date** (*str, optional*) – The end day up to which consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given. This day will be INCLUDED.

- **minAcc** (*int, optional*) – The minimum accuracy for a point to be kept. If accuracy is larger than this the point will be discarded. **NOTE that the accuracy column must be called** `acc`.
- **sep** (*str, optional*) – The delimiter of the fields in the files.
- **file_schema** (*list of tuples*) – The schema of the file. By default will be `dask_schemas.eventLineRAW`. **NOTE that the accuracy column must be called** `acc`.

Returns
df – A representation of the dataframe.

Return type
dask.dataframe

`mobilkit.loader.`**`load_raw_files_wb`**(*pattern*, *timezone=None*, *header=False*, *start_date=None*, *stop_date=None*, *minAcc=300*, *sep='\t'*, *file_schema=None*, ***kwargs*)

Function that loads the files and returns the dask dataframe. Note that this function is **lazy** meaning that it only construct the dataframe and does not build it (it will be built the first time a query is performed on it).

Parameters

- **pattern** (*str*) – The pattern of the raw files with bash syntax. For example: `'sample_data/20*/part-*.csv.gz'`.
- **timezone** (*str, optional*) – The timezone in pytz syntax (e.g., "Europe/Rome" or "America/Mexico_City") to be used to localize the Unix Time Stamp time-stamp in the raw-data. If no timezone is specified (default) it defaults to UTC.
- **start_date** (*str, optional*) – The starting date when to consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given.
- **stop_date** (*str, optional*) – The end day up to which consider data in the "yyyy-mm-dd" format. This will be localized in *timezone* if given. This day will be INCLUDED.
- **minAcc** (*int, optional*) – The minimum accuracy for a point to be kept. If accuracy is larger than this the point will be discarded. **NOTE that the accuracy column must be called** `acc`.
- **sep** (*str, optional*) – The delimiter of the fields in the files.
- **file_schema** (*list of tuples*) – The dict to rename the original columns to the mobilkit ones. **NOTE that the accuracy column must be called** `acc`.

Returns
df – A representation of the dataframe.

Return type
dask.dataframe

`mobilkit.loader.`**`loaddata_takeapeek`**(*dirpath*, *sep*, *ext*)

`mobilkit.loader.`**`localizeDatetimeNaive`**(*date*, *tz*, *date_format='%Y-%m-%d'*)

`mobilkit.loader.`**`persistDF`**(*df*, *path*, *overwrite=True*, *header=True*, *index=False*, *out_format='csv'*)

Save a dask dataframo file.

Parameters

- **df** (*dask.DataFrame*) – The dataframe to save
- **path** (*str*) – The path where to save the dataframe.
- **overwrite** (*bool*) – Whether or not to force overwrite.
- **header** (*bool*) – Whether or not to put the header in the output file.

- **index** (*bool*) – Whether or not to put the index column in the output file.
- **out_format** (*bool*) – One of `csv, parquet` the format to use. If the df has arrays in it use `parquet`.

`mobilkit.loader.`**`reloadDF`**(*path*, *header=True*, *in_format='csv'*)

Load a dask dataframe from file.

Parameters

- **path** (*str*) – The path where to read the dataframe from.
- **header** (*bool*) – Whether or not to read the header in the output file.
- **in_format** (*bool*) – One of `csv, parquet` the format used to persist the df.

Returns
df – The loaded dataframe.

Return type
dask.DataFrame

`mobilkit.loader.`**`syntheticGeoLifeDay`**(*df_geolife*, *selected_day*)

`mobilkit.loader.`**`syntheticGeoLifeWeek`**(*df_geolife*, *selected_week*)

mobilkit.spatial module

Tools and functions to spatially analyze the data.

Note: When determining the home location of a user, please consider that some data providers, like *Cuebiq*, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy.

This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern.

However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

`mobilkit.spatial.`**`assignAreasDF`**(*df*, *zones_gdf*)

Returns the geo-dataframe with an additional column the *ZONE_IDX* column. Non overlapping areas are guaranteed to be found there with a negative -1 value; The order of the original index and columns is preserved.

`mobilkit.spatial.`**`box2poly`**(*box*)

[min_lon, min_lat, max_lon, max_lat]

`mobilkit.spatial.`**`computeUsersLocations`**(*stops_df*, *method='dbscan'*, *link_dist=150*, *min_stops_count=2*, *return_locations=True*)

TODO DOC

`mobilkit.spatial.`**`compute_ROG`**(*df*, *which='both'*, *df_hw_locs=None*)

`mobilkit.spatial.`**`compute_medoid_index`**(*distM*)

Returns the row index of the necessarily symmetric matrix that minimizes the sum of distances to all the other points.

Parameters
distM (*np.array*) – The distances matrix.

Returns
index – The row (column) index of the medoid.

Return type
int

mobilkit.spatial.**compute_poi_index_dist**(*g*, *tree_model=None*, *lon_col='lng_proj'*, *lat_col='lat_proj'*)

Parameters

- **g** (*pandas.DataFrame*) – A dataframe containing at least the *lat_col* and *lon_col* columns with the raw points' coordinates projected to the same projection of the *tree_model*.
- **tree_model** (*sklearn.neighbors.KDTree*) – A KDTree trained on the POIs projected in the same proj of lat on lon points. Distance will be computed by the tree.
- **lat_col, lon_col** (*str*) – The names of the columns containing the latitude and longitude of the points in *g*. By default they match the ones used in *mobilkit.spatial.compute_poi_visit*.

Returns

g –

The original data frame with two additional columns named:

- ***'poi_distance'* the distance of the closest poi found in the tree in KM**;
- *'_POI_INDEX_'* the 0-based index of the closest tree leaf.

Return type
pd.DataFrame

mobilkit.spatial.**compute_poi_visit**(*df_pings*, *df_homes*, *df_POIs*, *from_crs='EPSG:4326'*, *to_crs='EPSG:6362'*, *min_home_dist_km=0.2*, *visit_time_bin='1H'*, *lat_lon_tol_box=0.02*)

Computes the set of users and number of users visiting a given POI for each *visit_time_bin* period of time found in the pings dataframe.

Parameters

- **df_ping** (*dask.DataFrame*) – A dataframe containing at least the *uid*, *datetime*, *lat* and *lng* columns with the raw points' coordinates. The coordinates must be given in the *from_crs* projection.
- **df_homes** (*Dataframe*) – A *pandas* or *dask* Dataframe with the *uid*, *homelat* and *homelon* home coordinates of all the users in the df. The coordinates must be given in the *from_crs* projection. Note that the three dataframes of pings, homes and POIs **must** feature the same initial projection equal to *from_crs*.
- **df_POIs** (*Dataframe*) – A *pandas* or *dask* Dataframe with at least the *radius*, *poilat* and *poilon* columns stating the radius to be considered in the POI (in km) and the POI's coordinates. The coordinates must be given in the *from_crs* projection.
- **from_crs, to_crs** (*str*) – The codes of the original and target projections to use. Will be used to compute planar distances in km using a euclidean distance so use the appropriate reference system for your ROI (e.g., use *to_crs='EPSG:6362"* for the Mexico area). Will be passed to *mobilkit.spatial.convert_df_crs*.

- **min_home_dist** (*float*) – The minimum distance for a point to be from the user's home to be considered valid (in km).
- **visit_time_bin** (*str*) – The frequency of the time bin to use. Each *datetime* will be floored to this time frequency.
- **lat_lon_tol_box** (*float*) – The pings will be filtered within the box of the maximum/minimum latitude/longitude of the POIs original projection's dataframe. This is the margin added around this box to account for pings right outside of the POIs' boundaries that may still fall into their radius.

Returns

pings_merged_home_poi, results –

- ***pings_merged_home_poi* is a view on the *dask.DataFrame* containing, for all the points falling within the POIs radius and far enough from users' home:**
  - the original pings columns plus their projected coords in *{lat,lng}_proj*;
  - the *home* and 'poi' original and projected (with '_proj' suffix) lat coords;
  - **'poi_distance', '_POI_INDEX_' the distance (in km) and the unique index of the** closest POI;
  - **all the *df_POIs* columns related to this POI (if common names of columns are** found they will be inserted with the *_FROM_POI_TABLE* suffix);
  - 'home_dist' the distance in km from the user's home;
  - 'time_bin' the original datetime floored to *visit_time_bin* freq.
- ***results* is a dataframe containing, for each unique *_POI_INDEX_* and *time_bin* as given by *visit_time_bin*:**
  - all the *df_POIs* columns related to this POI;
  - ***users,num_users* the columns containing the list of the *uid*-s of the users** found in that POI and that *time_bin* and their number.

Return type

dask.DataFrame, pd.DataFrame

`mobilkit.spatial.`**`compute_population_density`**(*df*, ***kwargs*)

Parameters

- **df** (*dask.DataFrame*) – A dataframe as returned by `mobilkit.temporal.filter_daynight_time` with at least the *date,daytime,uid,lat,lng* columns containing the date rounded to day, a bool stating if the point is in daytime or nightime, the user id and the coordinates of the point.
- ****kwargs** – Will be passed to `mobilkit.spatial.meanshift`.

Returns

df – A dataframe with a multi index of *date,daytime,uid* and as columns the *lat* and *lng* coordinates of the mean shift location of the user on that part of the day on that date.

Return type

pandas.DataFrame

mobilkit.spatial.**convert_df_crs**(*df*, *lat_col='lat'*, *lon_col='lng'*, *from_crs='EPSG:4326'*, *to_crs='EPSG:6362'*, *return_gdf=False*)

Parameters

- **df** (*pandas.DataFrame*) – A dataframe containing the *lat_col* and *lon_col* columns at least.
- **lat_col, lon_col** (*str*) – The names of the columns containing the latitude and longitude of the points in *df*.
- **from_crs, to_crs** (*str, optional*) – The codes of the original and target projections to use. If *to_crs* is None no reprojection is done.
- **return_gdf** (*bool, optional*) – If *True* returns the newly created gdf otherwise the original df with two additional columns telling the projected lat and lon.

Returns

df – If *return_gdf* the df ported to a geodataframe in the *to_crs* projection. Otherwise, the original data frame with two additional columns named *lat_col* + *'_proj'* and *lon_col* + *'_proj'* containing the original coordinates projected to *to_crs*.

Return type

pd.DataFrame or gpd.GeoDataFrame

mobilkit.spatial.**density_map**(*latitudes*, *longitudes*, *center*, *bins*, *radius*)

Parameters

- **latitudes, longitudes** (*array-like*) – The arrays containing the latitude and longitude coordinates of each user's location.
- **center** (*tuple-like*) – The (lat, lon) of the center where to compute the population density around.
- **bins** (*int*) – The number of bins to use horizontally and vertically in the region around the center.
- **radius** (*float*) – The space to consider above, below, left and right of the center (same unity of the center).

Returns

density – The 2d histogram of the population.

Return type

np.array

mobilkit.spatial.**distanceHomeDF**(*g*, ***kwargs*)

Parameters

- **g** (*pandas.DataFrame*) – A dataframe containing at least the *lat_col* and *lon_col* columns with the raw points' coordinates and the home coordinates in *homelon* and *homelat* columns of all the users.
- ****kwargs** – Such as *lat_col*, *lon_col* will be passed to *mobilkit.spatial.distanceHomeUser*.

Returns

g – The original data frame with an additional column named *'home_dist'* containing the haversine distance between each point and the home location of the user of that row **in kilometers**.

Return type

pd.DataFrame

mobilkit.spatial.**distanceHomeUser**(*g*, *lon_col='lng'*, *lat_col='lat'*, *h_lon_col='homelon'*, *h_lat_col='homelat'*)

Parameters

- **g** (*pandas.DataFrame*) – A dataframe containing at least the *lat_col* and *lon_col* columns with the raw points' coordinates and the home coordinates in *homelon* and *homelat* columns. **Must contain all the data of one user only.**
- **lat_col, lon_col** (*str*) – The names of the columns containing the latitude and longitude of the points in *g*.
- **h_lat_col, h_lon_col** (*str*) – The names of the columns containing the latitude and longitude of the home user *g*.

Returns

g – The original data frame with an additional column named *'home_dist'* containing the haversine distance between each point and the home location **in kilometers**.

Return type

pd.DataFrame

Note: When determining the home location of a user, please consider that some data providers, like *Cuebiq*, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy.

This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern.

However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

---

mobilkit.spatial.**expandStops**(*df*, *freq: str = '1h'*, *explode_stop: bool = True*)

Given a dataframe containing the single stops of one or more users (as returned by *mobilkit.spatial.findStops*) it explodes them to be repeated once every *freq* time bin they traverse (if *explode_stop*) or it lists them in the `mobilkit.dask_schemas.stpColName` column.

Parameters

- **df** (*dask.datframe*) – A dataframe containing the stops of one or more users (as returned by *mobilkit.spatial.findStops*).
- **freq** (*str, optional*) – The time bin to use to replicate/explode the stop. Currently all the valid *freq* arguments to *pandas.date_range* are accepted.
- **explode_stop** (*bool, optional*) – If *True* the `mobilkit.dask_schemas.stpColName` column will contain the exploded list of time bins touched by the stop, else the list itself.

Returns

exploded_stops_df – The initial dataframe with the additional column `mobilkit.dask_schemas.stpColName` containing the time bins (or their list, depending on *explode_stop*) touched by the stop.

Return type

dask.datframe

mobilkit.spatial.**filter_to_box**(*df*, *minlon*, *maxlon*, *minlat*, *maxlat*, *lat_col='lat'*, *lon_col='lng'*)

Parameters

- **df** (*DataFrame*) – A dataframe containing at least the *lat_col* and *lon_col* columns with the raw points' coordinates.
- **{min,max}{lat,lon}** (*float*) – The min and max values of lat and lon (will keep all coords >= min and <= max).
- **lat_col, lon_col** (*str*) – The names of the columns containing the latitude and longitude of the points in *df*.

Returns
df – The original data frame filtered to the points within the box.

Return type
pd.DataFrame

`mobilkit.spatial.`**`findStops`**`(`*df*, *tesselation_shp=None*, *stay_locations_kwds=None*, *filterAreas=True*`)`

Computes the stops of a group of users using the *scikit-mobility* tools. Note that the *mobilkit[complete]* or *[skmob]* version should be installed to use this tool.

Parameters

- **df** (*dask.dataframe*) – A dataframe containing the raw pings with at least the latitude, longitude and datetime columns.
- **tessellation_shp** (*str, optional*) – The path to be used to tessellate the stops (if *None*, no tessellation will be performed).
- **stay_locations_kwds** (*dict, optional*) – The custom keywords to be passed to `mobilkit.spatial._find_stops`. If not told otherwise, the stay location keywords passed to *skmob.preprocessing.detection* are:
  - *minutes_for_a_stop=5.0*
  - *spatial_radius_km=0.2*
  - *no_data_for_minutes=60*12*
  - *leaving_time=True*
- **filterAreas** (*bool, optional*) – Whether or not to filter out stops found outside of the tessellation when tessellating.

Returns

stops_df – The dataframe with the latitude and longitude of each stop together with: - its starting time (in the `mobilkit.dask_schemas.dttColName`) - its ending time (saved into the `mobilkit.dask_schemas.ldtColName` column). - the duration of the stop in seconds in the `mobilkit.dask_schemas.durColName` column. - if a tessellation file is specified, an additional `mobilkit.dask_schemas.zidColName`

is telling in which grid cell the stop is falling.

Return type
dask.dataframe

`mobilkit.spatial.`**`haversine_pairwise`**`(`*X*, *Y=None*, *isRadians=False*`)`

Parameters

- **X, Y** (*np.array*) – a Nx * 2 and Ny*2 arrays of (lat,lon) coordinates. If *Y* is *None* it will be assigned to *X* (computes the matrix of distances of *X* items).
- **isRadians** (*bool, optional*) – Whether the supplied coordinates are already in radians. If not they will be automatically converted.

Returns
distances – a Nx*Ny matrix of distances in kilometers

Return type
np.array

mobilkit.spatial.**makeVoronoi**(*gdf*)

mobilkit.spatial.**meanshift**(*df*, *bw=0.01*, *maxpoints=100*, ***kwargs*)

Given the points of a user finds the home location with MeanShift clustering.

Parameters

- **df** (*pandas.DataFrame*) – With at least *latcol,loncol*.
- **bw** (*float*) – Bandwidth to be used in MeanShift.
- **maxpoints** (*int*) – The maximum number of points to be used in meanshift. If more, a fraction of the df to have *maxpoints* will be sampled.
- **kwargs** – Will be passed to *sklearn.cluster.MeanShift* constructor.

Returns
clust_center – The center of the cluster found in the *(longitude,latitude)* format.

Return type
tuple

Note: When determining the home location of a user, please consider that some data providers, like *Cuebiq*, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy.

This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern.

However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

---

mobilkit.spatial.**plotActivityCount**(*df_act*, *gdf*, *what='pings'*, *ax=None*, *kwargs_map=None*)

Plots a colormap of the number of pings (or unique users) observed in a given area in a given period.

Parameters

- **df_act** (*pandas.dataframe*) – A dataframe as returned by `mobilkit.spatial.areaStats` with at least the *tile_ID* and *pings* and/or *users* columns and passed to pandas.
- **gdf** (*geopandas.GeoDataFrame*) – A geo-dataframe as returned by *mobilkit.spatial.tessellate*.
- **what** (*str*) – The `pings` or `users` string, telling whether to plot the number of pings recorded in an area or the number of unique users seen there.
- **ax** (*pyplot.axes, optional*) – The axes where to plot. If `None` (default) creates a new figure.
- **kwargs_map** (*dict, optional*) – Will be passed to the geopandas plot function plotting the boundaries and colormap.

Returns
ax – The axes of the figure.

Return type
pyplot.axes, optional

mobilkit.spatial.**plotHomeWorkPoints**(*uid*, *df_hw*, *gdf*, *ax=None*, *kwargs_bounds=None*, *kwargs_points=None*)

Plots the points in home and work hours for an user on the map.

Parameters

- **uid** (*(str or int, depending on the uid type)*) – The id of the user to plot.
- **df_hw** (*dask.dataframe*) – A dataframe as returned by *mobilkit.stats.userHomeWork* with at least the *uid*, *tile_ID* and *isHome* and *isWork* columns.
- **gdf** (*geopandas.GeoDataFrame*) – A geo-dataframe as returned by *mobilkit.spatial.tessellate*.
- **ax** (*pyplot.axes, optional*) – The axes where to plot. If `None` (default) creates a new figure.
- **kwargs_bounds** (*dict, optional*) – Will be passed to the geopandas plot function plotting the boundaries.
- **kwargs_bounds** (*dict, optional*) – Will be passed to the geopandas plot function plotting the boundaries.

Returns
ax – The axes of the figure.

Return type
pyplot.axes, optional

mobilkit.spatial.**plotHomeWorkUserCount**(*df_hw_locs*, *gdf*, *what='home'*, *ax=None*, *kwargs_map=None*)

Plots a colormap of the number of people living (or working) in each area.

Parameters

- **df_hw_locs** (*pandas.dataframe*) – A dataframe as returned by *mobilkit.stats.userHomeWorkLocation* with at least the *uid*, *home_tile_ID work_tile_ID* columns and passed to pandas.
- **gdf** (*geopandas.GeoDataFrame*) – A geo-dataframe as returned by *mobilkit.spatial.tessellate*.
- **what** (*str*) – The `home` or `work` string, telling whether to plot the number of people living or working in an area.
- **ax** (*pyplot.axes, optional*) – The axes where to plot. If `None` (default) creates a new figure.
- **kwargs_map** (*dict, optional*) – Will be passed to the geopandas plot function plotting the boundaries and colormap.

Returns

- **ax** (*pyplot.axes, optional*) – The axes of the figure.
- **gdf** (*geopandas.GeoDataFrame*) – The original geo dataframe with an additional column (`n_users_home` if counting home or `n_users_work` if counting work). If the column is already in the df it will be overwritten.
- **df** (*pandas.DataFrame*) – The `tile_ID` -> count of users mapping.

mobilkit.spatial.**points_to_medoid**(*df*, *latColName='lat'*, *lonColName='lng'*)

Returns the pd.Series containing the latitude and longitude of the medoid for the current df.

Parameters

- **df** (*dask.DataFrame or pd.DataFrame*) – The dataframe with at least the *latColName* and *lonColName*.
- **latColName, lonColName** (*str, optional*) – The columns of *df* containing the latitude and longitude.

Returns

medoid – A series with columns `mobilkit.dask_schemas.medLatColName` and `mobilkit.dask_schemas.medLonColName` containing the latitude and longitude of the medoid.

Return type

pd.Series

mobilkit.spatial.**rad_of_gyr**(*coords: array*, *center_of_mass=None*) → array

Parameters

- **coords** (*np.array*) – a Nx*2 array of (lat,lon) coordinates.
- **center_of_mass** (*np.array, optional*) – A (1,2) np array containing the latitude and longitude of the center of mass to be used to compute the ROG (for instance, the user's home).

Returns

radius_of_gyrations – The radius of gyration for the selected coords.

Return type

float

mobilkit.spatial.**replaceAreaID**(*df*, *mapping*)

Function that replaces all the `tile_ID` with a new id given in the mapping.

Parameters

- **df** (*dask.DataFrame*) – A dataframe with at least the `tile_ID` column.
- **mapping** (*dict*) – A mapping between the original `tile_ID` and the new desired one. MUST CONTAIN ALL THE `tile_ID` s present in df.

Returns

df_out – A copy of the original dataframe with the `tile_ID` replaced.

Return type

dask.DataFrame

mobilkit.spatial.**selectAreasFromBounds**(*gdf*, *relation='within'*, *min_lon=-99.15913*, *max_lon=-99.10032*, *min_lat=19.41353*, *max_lat=19.461*)

Function to select areas from a geodataframe given the bounds of a selected region.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – A geodataframe with at least the `tile_ID` and `geometry` columns as returned by *mobilkit.spatial.tessellate*.
- **relation** (*str, optional*) – The relation between the bounds and the areas. "within" or "intersects"
- **min/max_lon/lat** (*float, optional*) – The minimum and maximum latitude and longitude of the box.

Returns
areas_ids – The set of the areas **within** or **intersecting** the given bounds

Return type
set

mobilkit.spatial.**stack_density_map**(*df*, *dates*, *center*, *daytime=True*, *bins=200*, *radius=1*)

Parameters

- **df** (*pd.DataFrame*) – A dataframe as returned by *mobilkit.spatial.compute_population_density* with a multi index of *date,daytime,uid* and as columns the *lat* and *lng* coordinates of the mean shift location of the user on that part of the day on that date.
- **dates** (*list pof datetime*) – A list of dates when to compute the density.
- **center** (*tuple-like*) – The (lat, lon) of the center where to compute the population density around.
- **daytime** (*bool*) – Whether to compute the density on the daytime or nightime part of selected dates.
- **bins** (*int*) – The number of bins to use horizontally and vertically in the region around the center.
- **radius** (*float*) – The space to consider above, below, left and right of the center (same unity of the center).

Returns
maps, results – *maps* is the tensor of shape *(len(dates),bins,bins)* storing for each date the x-y density map as computed by *mobilkit.spatial.density_map*. *results* stores the x and y bins.

Return type
np.array, tuple

mobilkit.spatial.**stats_density_map**(*df*, *dates*, *center*, *daytime=True*, *bins=200*, *radius=1*, *clip_std=0.0001*)

Parameters

- **df** (*pd.DataFrame*) – A dataframe as returned by *mobilkit.spatial.compute_population_density* with a multi index of *date,daytime,uid* and as columns the *lat* and *lng* coordinates of the mean shift location of the user on that part of the day on that date.
- **dates** (*list pof datetime*) – A list of dates when to compute the density.
- **center** (*tuple-like*) – The (lat, lon) of the center where to compute the population density around.
- **daytime** (*bool*) – Whether to compute the density on the daytime or nightime part of selected dates.
- **bins** (*int*) – The number of bins to use horizontally and vertically in the region around the center.
- **radius** (*float*) – The space to consider above, below, left and right of the center (same unity of the center).
- **clip_std** (*float*) – Pixels with a 0 or *nan* std will be clipped to this value when computing the z-score. The same pixels will be set to -1 on output.

Returns

results –

A dictionary containing teh key-values:

- ***stack* the tensor of shape *(len(dates),bins,bins)* storing for each date**
  the x-y density map as computed by `mobilkit.spatial.density_map`.
- ***avg*, *std* the average and standard deviation population density with**
  shape *(1,bins,bins)*.
- ***x_bins*, *y_bins* the bins of the 2d histogram as produced by**
  `mobilkit.spatial.density_map`.

Return type
dict

`mobilkit.spatial.`**`tessellate`**`(`*df*, *tesselation_shp*, *filterAreas=False*, *partitions_number=None*, *latCol='lat'*, *lonCol='lng'*`)`

Function to assign to each point a given area index.

Parameters

- **df** (*dask.DataFrame*) – A dataframe as returned from `mobilkit.loader.load_raw_files` or imported from `scikit-mobility` using `mobilkit.loader.load_from_skmob`.
- **tesselation_shp** (*str*) – The path (relative or absolute) to the shapefile containing the tesselation of space. If the shapefile does not contain a *tile_ID* field it will be initialized here and included in the returned geodataframe.
- **filterAreas** (*bool*) – If tesselation is specified, keeps only the points within the specified shapofile.
- **partitions_number** (*int, optional*) – The batch size of the geopandas sjoin function to be applied. Leave it as is unless you know what you're doing.
- **latCol, lonCol** (*str, optional*) – The names of the columns containing the latitude and longitude coordinates.

Returns

- **df_tile** (*dask.dataframe*) – The initial dataframe with the additional `tile_ID` column telling the int id of the area the point is belonging to (-1 if the point is outside of the shapefile bounds).
- **tessellation_gdf** (*geopandas.GeoDataFrame*) – The geo-dataframe with the possibly missing *tile_ID* column added.

`mobilkit.spatial.`**`totalUserTravelDistance`**`(`*df_pings*, *doROG=False*, *freq='1d'*`)`

Computes the total distance traveled (km computed on the straight lines between each point) by a user i each *freq* time bin.

Parameters

- **df_pings** (*dask.DataFrame*) – The dataframe containing at least the `mobilkit.dask_schemas.uidColName`, `mobilkit.dask_schemas.latColName`, `mobilkit.dask_schemas.lonColName` and `mobilkit.dask_schemas.dttColName`.
- **doROG** (*bool, optional*) – If *True* also computes the ROG on the pings of that day.
- **freq** (*str, optional*) – The datetime interval to fllor the *dttColName* to (default one day).

Returns

ttd – The dataframe containing the *user,tBin* index and: - *ttd* column with the total traveled distance (in km) for that user on that time bin. - *nPings* the number of pings for that user on that time bin; - if *doROG* a column *rog* with the daily ROG using as center of mass the mean point

of that time bin's coordinates.

Return type
dask.DataFrame

mobilkit.spatial.**total_distance_traveled**(*coords*)

Parameters
coords (*np.array*) – a Nx*2 array of (lat,lon) coordinates.

Returns
total_distance_traveled – The total distance traveled in the dataframe.

Return type
float

mobilkit.spatial.**userHomeWorkDistance**(*r*)

Computes the distance between *lat/lng_home* and *lat/lng_work* for the row of a user.

Returns
dist – None if one of the coords is invalid, the distance in km otherwise.

Return type
float

mobilkit.spatial.**user_dist_cbds**(*df_hw*, *cbds_latlon*, *assign_lat_col='lat_work'*, *assign_lng_col='lng_work'*, *distance_lat_col='lat_home'*, *distance_lng_col='lng_home'*)

Computes the distance from the CBD for each user using the cbd which is closest to the assign lat lons and computing its distance from distance_lat/lng.

mobilkit.stats module

Tools and functions to compute the per-users and per area stats.

Note: When determining the home location of a user, please consider that some data providers, like _Cuebiq_, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy.

This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern.

However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

---

mobilkit.stats.**areaStats**(*df*, *start_date=None*, *stop_date=None*, *hours=(0, 24)*, *weekdays=(1, 2, 3, 4, 5, 6, 7)*)

Computes the stats of a given area in terms of pings and unique users seen in a given area in a given period.

Parameters

- **df** (*dask.dataframe*) – A dataframe as returned by *mobilkit.spatial.tessellate* with at least the *uid*, *tile_ID* and *datetime* columns.
- **start_date** (*datetime.datetime*) – A python datetime object with no timezone telling the date (included) to start from. The default behavior is to keep all the events.
- **stop_date** (*datetime.datetime, optional*) – A python datetime object with no timezone telling the date (excluded) to stop at. Default is to keep all the events.
- **hours** (*tuple, optional*) – The hours when to start (included) and stop (excluded) in float notation (e.g., 09:15 am is 9.25 whereas 10:45pm is 22.75).
- **weekdays** (*tuple or set or list, optional*) – The list or tuple or set of days to be kept in python notation so 0 = Monday, 1 = Tuesday, … and 6 = Sunday.

Returns
df – The `tile_ID -> count of pings/users` mapping.

Return type
dask.DataFrame

`mobilkit.stats.`**`compressLocsStats2hwTable`**`(`*`df`*`)`

Transforms a per location home work stats table into the per user home and work stats table.

Parameters
df (*pd.DataFrame*) – A dataframe containing all the stats of the locations.

Returns
hw_stats – The home work locations of the users as if they were returned by *mobilkit.stats.userHomeWorkLocation*.

Return type
pd.DataFrame

`mobilkit.stats.`**`computeBufferStat`**`(`*`gdf_stat`*`, `*`gdf_grid`*`, `*`column`*`, `*`aggregation`*`, `*`how='inner'`*`, `*`lat_name='lat'`*`, `*`lon_name='lng'`*`, `*`local_EPSG=None`*`, `*`buffer=None`*`)`

Computes the statistics contained in a column of a dataframe containing the lat and lon coordinates of points with respect to a *gdf_grid* tessellation, possibly applying local reprojection and buffer on the points. This is equivalent to a KDE with a flat circular kernel of radius *buffer*.

Parameters

- **gdf_stat, gdf_grid** (*gpd.GeoDataFrame*) – The geo-dataframes containing the statistics in the *column* column and the tessellation system. They must be in the same reference system and will be projected to *local_EPSG*, if specified. The *gdf_grid* will be dissolved on the `mobilkit.dask_schemas.zidColName` after the spatial join with the (possibly buffered) *gdf_stat* geometries.
- **column** (*str*) – The column for which we will compute the statistics.
- **aggregation** (*str or callable*) – The geopandas string or callable to use on the spatially joined geo-dataframe.
- **how** (*str, optional*) – The method to perform the spatial join.
- **lat_name, lon_name** (*str, optional*) – The name of the columns to use as initial coords.
- **local_EPSG** (*int, optional*) – The code of the local EPSG crs.
- **buffer** (*float, optional*) – The local map unit in *local_EPSG* to perform the buffer.

Returns
buffered_stats – The geodataframe with the aggregated stat.

Return type
gpd.GeoDataFrame

mobilkit.stats.**computeHomeWorkSurvival**(*df_stops_stats*, *min_durations=[0]*, *min_day_counts=[0]*, *min_hour_counts=[0]*, *min_delta_counts=[0]*, *min_delta_durations=[0]*, *limit_hw_locs=False*, *loc_col='loc_ID'*)

Given a dataframe of locations (tiles) with home work stats as returned by *mobilkit.stats.stopsToHomeWorkStats* it computes the home and work presence at different thresholds of home and work duration count etc.

Parameters

- **df_stops_stats** (*pandas.DataFrame*) – The locations (or tiles) stats of the users as returned by *mobilkit.stats.stopsToHomeWorkStats*.
- **min_durations** (*iterable, optional*) – The minimum duration of home and work stops to keep lines in the group.
- **min_day_counts, min_hour_counts** (*iterable, optional*) – The minimum count of stops in home and work locations to keep lines in the group.
- **min_delta_counts, min_delta_durations** (*iterable, optional*) – The minimum fraction of home/work hours during which the area/location is the most visited in terms of duration/count of stops for it to be kept.
- **limit_hw_locs** (*bool, optional*) – If *True*, it will limit the home and work candidates to the row(s) featuring *isHome* or *isWork* equal *True*, respectively. If *False* (default), all the rows are kept as candidates.
- **loc_col** (*str, optional*) – The column to use to check if the home and work candidates are in the same location.

Returns

- **user_flags** (*pd.DataFrame*) – A data frame indexed by user containing, for each combination of threshold values in the order of minimum duration, minimum days, minimum hours, min delta count, min delta duration, the flag of: - *out_flags* if the user has a home AND work candidate with the threshold; - *out_has_home* if the user has a home candidate with the threshold; - *out_has_work* if the user has a work candidate with the threshold; - *out_same_locs* if the user has a unique the home and work candidate falling

  under the same *loc_col* ID.

- **df_cnt** (*pd.DataFrame*) – The dataframe in long format containing the count of valid counts for the users for each combination of minimum threshold. The columns are: - 'tot_duration', 'n_days', 'n_hours', 'delta_count', 'delta_duration'

  the values of the constraint for the current count.

  - 'n_users' how many users have both home and work with current settings;
  - 'with_home_users' how many users have a home location with current settings;
  - 'with_work_users' how many users have a work location with current settings;
  - 'home_work_same_area_users' how many users have home and work locations featuring the same *loc_col* ID.
  - 'home_work_same_area_users_frac' the fraction of valid users with home and work that have have home and work locations featuring the same *loc_col* ID.

mobilkit.stats.**computeSurvivalFracs**(*users_stats_df*, *thresholds=[1, 10, 20, 50, 100]*)

Function to compute the fraction of users above threshold.

Parameters

- **users_stats** (*pandas.DataFrame*) – A dataframe with the users stats as returned by `mobilkit.stats.userStats` and passed to pandas with the `toPandas` method.
- **thresholds** (*list or array of ints, optional*) – The values of the threshold to compute. The number of days above the threshold and the fraction of active days above threshold will be saved, for each user, in the `days_above_TTT` and `frac_days_above_TTT` where `TTT` is the threshold value.

Returns

df – The enriched dataframe.

Return type

pandas.DataFrame

mobilkit.stats.**computeTripTimeStats**(*df_trip_times*, *df_hw_locs*, *gdf_grid*, *local_EPSG*, *buffer_m=500*)

mobilkit.stats.**computeUserHomeWorkTripTimes**(*df_hw_locs*, *osrm_url=None*, *direction='both'*, *what='duration'*, *max_trip_duration_h=4*, *max_trip_distance_km=150*)

TODO This is quite slow as it is a serial part, it can be parallelized using a pool or directly mapping in Dask
:rtype: time in seconds, distance in meters

mobilkit.stats.**filterUsers**(*df*, *dfStats=None*, *minPings=1*, *minDaysSpanned=1*, *minDaysActive=1*, *minSuperUserDayFrac=None*, *superUserPingThreshold=None*)

Function to filter the pings and keep only the ones of the users with given statistics.

Parameters

- **df** (*dask.dataframe*) – The dataframe containing the pings.
- **dfStats** (*dask.dataframe, optional*) – The dataframe containing the pre-computed stats of the users as returned by `mobilkit.stats.userStats`. If `None`, it will be automatically computed. In either cases it is returned together with the result.
- **minPings** (*int*) – The minimum number of recorded pings for a user to be kept.
- **minDaysSpanned** (*float*) – The minimum number of days between the first and last ping for a user to be kept.
- **minDaysActive** (*int*) – The minimum number of active days for a user to be kept.
- **minSuperUserDayFrac** (*float*) – The minimum fraction of days with same or more pings than `superUserPingThreshold` for a user to be considered. Must be between 0 and 1.
- **superUserPingThreshold** (*int*) – The minimum number of pings for a user-day to be considered as super user.

Returns

df_out, df_stats, valid_users_set – The dataframe containing the pings of the valid users only, the one containing the stats per user and the set of the valid users.

Return type

dask.dataframe, dask.dataframe, set

mobilkit.stats.**filterUsersFromSet**(*df*, *users_set*)

Function to filter the pings and keep only the ones of the users in `users_set`.

Parameters

- **df** (*dask.dataframe*) – The dataframe containing the pings.
- **users_set** (*set or list*) – The ids of the users to keep.

Returns
df_out – The filtered dataframe containing the pings of the valid users only.

Return type
dask.dataframe

`mobilkit.stats.`**`homeWorkStats`**(*df_hw*)

Given a dataframe returned by *mobilkit.stats.userHomeWork* computes, for each user and area, the total number of pings recorded in that area (`total_pings` column), the pings recorded in home hours (`home_pings` column) and the ones in work hours (`work_pings` column).

Parameters
df_hw (*dask.dataframe*) – A dataframe as returned by *mobilkit.stats.userHomeWork* with at least the *uid*, *tile_ID* and *isHome* and *isWork* columns.

Returns

df_hw_stats –

The dataframe containing, for each user and area id:

- `total_pings`: the total number of pings recorded for that user in that area
- `home_pings`: the pings recorded for that user in home hours in that area
- `work_pings`: the ping in work hours for that user in that area

Return type
dask.dataframe

Note: When determining the home location of a user, please consider that some data providers, like _Cuebiq_, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy.

This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern.

However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

`mobilkit.stats.`**`plotSurvivalDays`**(*users_stats_df*, *min_days=10*, *ax=None*)

Function to plot the survival probability of users by number of days given different pings/day threshold.

Parameters

- **users_stats_df** (*pandas.DataFrame*) – A dataframe with the users stats as returned by *mobilkit.stats.computeSurvivalFracs*.
- **min_days** (*int, optional*) – The minimum number of active days above threshold to be counted as super user in the plot count.
- **ax** (*plt.axes, optional*) – The axes to use. If `None` a new figure will be produced.

Returns
The axes of the figure.

Return type
ax

mobilkit.stats.**plotSurvivalFrac**(*users_stats_df*, *min_frac=0.8*, *ax=None*)

Function to plot the survival probability of users by fraction of active days given different pings/day threshold.

Parameters

- **users_stats_df** (*pandas.DataFrame*) – A dataframe with the users stats as returned by `mobilkit.stats.computeSurvivalFracs`.
- **min_frac** (*0 < float < 1, optional*) – The minimum fraction of active days above threshold to be counted as super user in the plot count.
- **ax** (*plt.axes, optional*) – The axes to use. If `None` a new figure will be produced.

Returns
The axes of the figure.

Return type
ax

mobilkit.stats.**plotUsersHist**(*users_stats*, *min_pings=5*, *min_days=5*, *days='active'*, *cmap='YlGnBu'*, *xbins=100*, *ybins=20*)

Function to plot the 2d histogram of the users stats.

Parameters

- **users_stats** (*pandas.DataFrame*) – A dataframe with the users stats as returned by `mobilkit.stats.userStats` and passed to pandas with the `toPandas` method.
- **min_pings** (*int, optional*) – The number of pings to be used as threshold in the plot counts.
- **min_days** (*int, optional*) – The number of active or spanned days (depending on `days`) to be used as threshold in the plot counts.
- **days** (*str, optional*) – Whether to use active (`active`, default) days or spanned days (`spanned`).
- **cmap** (*str, optional*) – The colormap to use.
- **xbins, ybins** (*int, optional*) – The number of bins to use on the x and y axis.

Returns
The axes of the figure.

Return type
ax

mobilkit.stats.**stopsToHomeWorkStats**(*df_stops*, *home_hours=(21, 7)*, *work_hours=(9, 17)*, *work_days=(0, 1, 2, 3, 4)*, *force_different=False*, *ignore_dynamical=True*, *min_hw_distance_km=0.0*, *min_home_delta_count=0*, *min_home_delta_duration=0*, *min_work_delta_count=0*, *min_work_delta_duration=0*, *min_home_days=0*, *min_work_days=0*, *min_home_hours=0*, *min_work_hours=0*)

Computes the home and work time stats for each user and location (tile).

Parameters

- **df_stop_locs_usr** (*dask.DataFrame or pd.DataFrame*) – The stops of a user as returned by locations or stops TODO;
- **home_hours, work_hours** (*tuple, optional*) – TODO
- **work_days** (*tuple*) – TODO

- **force_different** (*bool, optional*) – TODO
- **ignore_dynamical** (*bool, optional*) – TODO
- **min_hw_distance_km** (*float, optional*) – TODO
- **min_home_delta_count, min_home_delta_duration,**
- **min_work_delta_count, min_work_delta_duration** (*float, optional*) – TODO
- **min_home_days, min_work_days,**
- **min_home_hours, min_work_hours** (*int, optional*) – TODO
- **latCol, lonCol, locCol** (*str, optional*) – TODO

Returns

df_stats – A dataframe with the columns:

- *uid* the user id
- 'loc_id' or 'tile_ID' the location/tile id 0-based;
- 'lat_medoid','lng_medoid' or 'lat', 'lng' the average coordinates of the stops seen within that location/tile;
- '{home,work}_{day/hour}_count' the number of unique days (hours) when the user has been seen as active in the location (tile) at home (work) hours;

- '{home,work}_per_hour_{count,duration}' the list containing, for each hour in the home (work) hours, the number of visits (duration in seconds) spent at the location/tile;
- '{home,work}_{count,duration}' the total number of visits (seconds duration) spent at this location/tile;
- 'tot_seen_{home,work}_{hours,days}' the total number of days and hours where the user has been active during home (work) hours during the valid stops;
- 'tot_seen_{hours,days}' the total number of days and hours where the user has been active during the valid stops, both in home and workj period;
- 'tot_stop_count', 'tot_stop_time' the total number and duration (in seconds) of the user's stops;
- 'frac_{home,work}_{count,duration}' the fraction of stops (duration) spent in this tile/location during home (work) hours;
- '{home,work}_delta_{count,duration}' the fraction of hours in the home (work) range at which the given tile/location was the most visited in terms of stops (duration).
- 'isHome', 'isWork' the flag telling whsther the location is home or work (or potentially both, if *force_different* is False).

Return type

pd.DataFrame

mobilkit.stats.**userBasedBufferedStat**(*df_stat*, *df_user_grid*, *stat_col*, *uid_col='uid'*, *tile_col='tile_ID'*, *explode_col=False*, *how='inner'*, *stats=['min', 'max', 'mean', 'std', 'count']*)

Given a dataframe containing the per user stat *df_stat* in the *stat_col* and a dataframe containing the users per area as returned from *mk.stats.computeBufferStat* computes the *stats* of the *stat_col* merging the two df on the *tile_col*.

Parameters

- **df_stat** (*pd.DataFrame*) – The dataframe containing at least the *uid_col* and *stat_col*. They can also be in the df's index as it will be reset.
- **df_user_grid** (*pd.DataFrame*) – A dataframe containing the users per area (in the *uid_col* and *tile_col*)as returned from *mk.stats.computeBufferStat* using passing as *gdf_stat* the home work locations, *lat_name='lat_home',lon_name='lng_home'*, *column=uidColName* and *aggregation=set*.
- **uid_col, tile_col** (*str, optional*) – The columns containing the user id and the tile id in the two input dfs.
- **explode_col** (*bool, optional*) – Whether we need to explode the *stat_col* before merging (for list-like observations).
- **how** (*str, optional*) – The join method to use between the tile ids in the grid and the df of stats.
- **stats** (*list or str*) – The stats to be compute at the tile level on the *stat_col* column.

Returns

stats – The dataframe containing the tile id as index and with the stats in the *stat_col_{min/max/mean}* format.

Return type

pd.DataFrame

Examples

```
>>> # Compute the per area buffered users based on home location (500m buffer):
>>> users_buffered_per_area = mk.stats.computeBufferStat(
                               gdf_stat=df_hw_locs_pd.reset_index()[['lat_home',
↪'lng_home', uidColName]],
                               gdf_grid=gdf_aoi_grid,
                               column=uidColName,
                               aggregation=set,
                               lat_name='lat_home',
                               lon_name='lng_home',
                               local_EPSG=local_EPSG,
                               buffer=500)
>>> # Compute the per area total daily traveled distance
>>> ttd_daily_user = mk.spatial.totalUserTravelDistance(df_pings, freq='1d')
>>> df_out = mk.stats.userBasedBufferedStat(ttd_daily_user,
                                            users_buffered_per_area,
                                            stat_col='ttd')
>>> df_out.head()
tile_ID   |    ttd_min |    ttd_max |   ttd_mean |    ttd_std |  ttd_count |
12345     |      2.345 |     12.345 |      5.345 |      3.345 |        125 |
```

`mobilkit.stats.`**`userHomeWork`**(*df*, *homeHours=(19.5, 7.5)*, *workHours=(9.0, 18.5)*, *weHome=False*)

Computes, for each row of the dataset, if the ping has been recorded in home or work time. Can be used in combination with `mobilkit.stats.homeWorkStats' and :attr:'mobilkit.stats.userHomeWorkLocation` to determine the home and work locations of a user.

Parameters

- **df** (*dask.dataframe*) – The loaded dataframe with at least *uid*, *datetime* and *tile_ID* columns.

- **homeHours** (*tuple, optional*) – The starting and end hours of the home period in 24h floating numbers. For example, to put the house period from 08:15pm to 07:20am put `homeHours=(20.25, 7.33)`.
- **workHours** (*tuple, optional*) – The starting and end hours of the work period in 24h floating numbers. For example, to put the work period from 09:15am to 06:50pm put `workHours=(9.25, 18.8333)`. **Note that work hours are counted only from Monday to Friday.**
- **weHome** (*bool, optional*) – If `False` (default) counts only weekend hours within the home hours as valid home hours. If `True`, all the pings recorded during the weekend (Saturday and Sunday) are counted as home pings.

Returns

out – The dataframe with two additional columns: *isHome* and *isWork* telling if a given ping has been recorded during home or work time (or none of them).

Return type

dask.dataframe

Note: When determining the home location of a user, please consider that some data providers, like _Cuebiq_, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy.

This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern.

However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

---

mobilkit.stats.**userHomeWorkLocation**(*df_hw: <module 'dask.dataframe' from '/home/docs/checkouts/readthedocs.org/user_builds/mobilkit/envs/latest/lib/python3.11/site-packages/dask/dataframe/__init__.py'>*, *force_different: bool = False*)

Given a dataframe returned by *mobilkit.stats.userHomeWork* computes, for each user, the home and work area as well as their location. The home/work area is the one with more pings recorded and the location is assigned to the mean point of this cloud.

Parameters

- **df_hw** (*dask.dataframe*) – A dataframe as returned by *mobilkit.stats.userHomeWork* with at least the *uid*, *tile_ID* and *isHome* and *isWork* columns.
- **force_different** (*bool, optional*) – Whether we want to force the work location to be different from the home location.

Returns

df_hw_locs –

A dataframe containing, for each `uid` with at least one ping at home or work:

- `pings_home`: the total number of pings recorded in the home area
- `pings_work`: the total number of pings recorded in the work area
- `tile_ID_home`: the tile id of the home area
- `tile_ID_work`: the tile id of the work area

- `lng_home`: the longitude of the home location
- `lat_home`: the latitude of the home location
- `lng_work`: the longitude of the work location
- `lat_work`: the latitude of the work location

Return type
dask.dataframe

Note: When determining the home location of a user, please consider that some data providers, like _Cuebiq_, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy.

This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern.

However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

`mobilkit.stats.`**`userRealHomeWorkTimes`**(*df_stops*, *home_work_locs*, *direction='both'*, *uid_col='uid'*, *location_col='tile_ID'*, *additional_hw_cols=['home_work_straight_dist', 'home_work_osrm_time', 'work_home_osrm_time', 'home_work_osrm_dist', 'work_home_osrm_dist']*, ***kwargs*)

Computes the real homework commuting time looking at the sequence of the user's stops.

Parameters

- **df_stops** (*dd.DataFrame*) – A dask dataframe containing the stops (or pings) of the users to be analyzed. It might feature a location id column (as when returned by *mobilkit.spatial.computeUsersLocations* applied on the output of *mobilkit.spatial.findStops*) or the tile id column (as in the df returned by *mobilkit.spatial.tessellate*).
- **home_work_locs** (*dd.DataFrame or pd.DataFrame*) – The necessarily pre-cleaned dataframe containing the stats on users home and work. This can be either a dataframe as returned by *mobilkit.stats.userHomeWorkLocation* or the one obtained by chaining the *mobilkit.stats.stopsToHomeWorkStats* and the `mobilkit.stats.`
- ****kwargs** – Are used to tune the functioning of `mobilkit.stats._per_user_real_home_work_times`

`mobilkit.stats.`**`userStats`**(*df*)

Computes the stats per user:

- days spanned (time from first to last ping);
- days active (actual number of days being active);
- number of pings per user;
- number of pings per user per active day;

Parameters
df (*dask.dataframe*) – The dataframe extracted or imported. Must contains the `uid` column and the `datetime` one.

Returns

df_out –

A dask dataframe with the stats per user in three columns:

- `daysActive` the number of different days where the user has been active;
- `daysSpanned` the days spanned between the first and last recorded ping;
- `pings` the number of pings recorded for the user;
- `pingsPerDay` the number of pings recorded for the user in every active day;
- `avg` the average number of pings per recorded day for the user.

Return type
dask.DataFrame

Example

```
>>> df_out = mk.stats.userStats(df)
>>> df_out.head()
uid   |    min_day |    max_day |  pings |  daysActive |   avg | daysSpanned |␣
↪pingsPerDay    |
'abc' | 2017-08-12 | 2017-12-22 |   3452 |         124 | 27.83 |         222 | [12,
↪22,...,13] |
```

mobilkit.temporal module

Tools and functions to analyze the data in time.

`mobilkit.temporal.`**`computeDisplacementFigures`**(*df_disp*, *minimum_pings_per_night=5*)

Given a dataframe returned by *mobilkit.temporal.homeLocationWindow* computes a pivoted dataframe with, for each user, the home area for every time window, plus the arrays of displaced and active people per area and the arrays with the (per user) cumulative number of areas where the user slept.

Parameters

- **df_disp** (*pandas.dataframe*) – A dataframe as returned by *mobilkit.temporal.homeLocationWindow*.
- **minimum_pings_per_night** (*int, optional*) – The number of pings recorded during a night for a user to be considered.

Returns

df_pivoted, first_user_area, heaps_arrays, count_users_per_area –

- **`df_pivoted` is a dataframe containing one row per user and with the column being the sorted**
  time windows of the analysis period. Each cell contains the location where the user (row) has slept in night t (column), `Nan` if the user was not active that night.
- **`first_user_area` is a dict telling, for each user, the `tile_ID` where he has been sleeping**
  for the first time.
- **`heaps_arrays` is a (n_users x n_windows) array telling the cumulative number of areas where**
  a users slept up to window t.

- **counts_users_per_area is a dictionary {tile_ID: {"active": [...], "displaced": [...]}}**
  telling the number of active and displaced people per area in time.

Return type
pandas.dataframe, dict, array, dict

mobilkit.temporal.**computeResiduals**(*df_activity*, *signal_column*, *profile*)

Function that computes the average, z-score and residual activity of an area in a given time period and for a given time bin.

Parameters

- **df_activity** (*dask.DataFrame*) – As returned by *mobilkit.temporal.computeTemporalProfile*, a dataframe with the columns and periods volumes and normalization (if needed) already computed.
- **profile** (*str*) – The temporal profile used for normalization in *mobilkit.temporal.computeTemporalProfile*.
- **signal_column** (*str*) – The columns to use as proxy for volume. Usually one of "users", "pings", "frac_users", "frac_pings"

Returns

Two dictionaries containing the aggregated results in numpy arrays. results has four keys:

- raw the raw signal in the area_index,period_index,period_hour_index indexing;
- **mean the mean over the periods of the raw signal in the** area_index,period_hour_index shape;
- **zscore the zscore of the area signal (with respect to its average and std) in the** area_index,period_hour_index shape;
- **residual the residual activity computed as the difference between the area's zscore** and the global average zscore at a given hour in the area_index, period_hour_index shape;

On the other hand, mappings contains the back and forth mapping between the numpy indexes and the original values of the areas (idx2area and area2idx), periods, and, hour of the period. These will be useful later for plotting.

Return type
results, mappings

mobilkit.temporal.**computeTemporalProfile**(*df_tot*, *timeBin*, *byArea=False*, *profile='week'*, *weekdays=None*, *normalization=None*, *start_date=None*, *stop_date=None*, *date_format=None*, *sliceName=None*, *selected_areas=None*, *areasName=None*, *split_out=10*)

Function to compute the normalized profiles of areas. The idea is to have a dataframe with the count of users and pings per time bin (and per area is byArea=True) together with a normalization column (computed if normalization is not None over a different time window profile) telling the total number of pings and users seen in that period (and in that area if byArea). If normalization is specified, also the fraction of users and pings recorded in an area at that time bin are given.

Parameters

- **df_tot** (*dask.DataFrame*) – A dataframe as returned from *mobilkit.loader.load_raw_files* or imported from scikit-mobility using *mobilkit.loader.*

load_from_skmob. If using `byArea` the df must contain the `tile_ID` column as returned by *mobilkit.spatial.tessellate*.

- **timeBin** (*str*) – The width of the time bin to use to aggregate activity. Currently supported: ["W", "MS", "M", "H", "D", "T"] You can implement others found in [pandas time series aliases](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliases). For instance:
  - *'B'* business day frequency
  - *'D'* calendar day frequency
  - *'W'* weekly frequency
  - *'M'* month end frequency
  - *'MS'* month start frequency
  - *'SMS'* semi-month start frequency (1st and 15th)
  - *'BH'* business hour frequency
  - *'H'* hourly frequency
  - *'T','min'* minutely frequency
- **byArea** (*bool, optional*) – Whether or not to compute the activity per area (default `False`). If `False` will compute the overall activity.
- **profile** (*str*) – The base of the activity profile: must be `"week"` to compute the weekly profile or `"day"` for the daily one or `"month"` for one month period (*month_end* to use month end). Each profile of area / week or day (depending on profile) will be computed separately. **NOTE** that this period should be equal or longer than the `timeBin` (i.e., `"weekly"` or `"monthly"` if `timeBin="week"`) otherwise the normalization will fail.
- **weekdays** (*set or list, optional*) – The weekdays to consider (0 Monday -> 6 Sunday). Default `None` equals to keep all.
- **normalization** (*str, optional*) – One of `None, "area", "total"`. Normalize nothing (`None`), on the total period of the area (`area`) or on the total period of all the selected areas (`total`).
- **start_date** (*str, optional*) – The starting date when to consider data in the `date_format` format.
- **stop_date** (*str, optional*) – The end date when to consider. Must have the same format as `start_date`.
- **date_format** (*str, optional*) – The python date format of the dates, if given.
- **sliceName** (*str, optional*) – The name that will be saved in *timeSlice* column, if given.
- **selected_areas** (*set or list, optional*) – The set or list of selected areas. If `None` (default) uses all the areas. Use `mobilkit.spatial.selecteAreasFromBounds` to select areas from given bounds.
- **areasName** (*str, optional*) – The name that will be saved in *areaName* column, if given.
- **split_out** (*int, optional*) – The number of partitions to split the results in (for large number of areas and time bins). The default value of 10 should work in most of the cases.

Returns

df_normalized – A dataframe with these columns: - one with the same name as `timeBin` with the date truncated at the selected width.

- `pings` the number of pings recorded in that time bin and area (if `byArea=True`).
- `users` the number of users seen in that time bin and area (if `byArea=True`).
- **`pings_per_user` the average number of pings per user in that time bin and area** (if `byArea=True`).
- `tile_ID` (if `byArea=True`) the area where the signal has been recorded.
- **the additional columns `timeSlice` and ``areaName`` if the two names are given,** plus, if `normalization` is not `None`:
- **`tot_pings/users` the total number of pings and users seen in the area (region) in** the profile period if normalize is `"area"` (`total`).
- **`frac_pings/users` the fraction of pings and users seen in that area, at that time bin** with respect to the total volume of the area (region) depending on the normalization.
- **`profile_hour` the zero-based hour of the typical month, week or day (depending on the** value of `profile`).

Return type
dask.DataFrame

`mobilkit.temporal.`**`computeTimeBinActivity`**(*df*, *byArea=False*, *timeBin='hour'*, *split_out=10*)

Basic function to compute, for each time bin and area, the activity profile in terms of users and pings recorded. It also computes the set of users seen in that bin for later aggregations.

Parameters

- **df** (*dask.DataFrame*) – A dataframe as returned from *mobilkit.loader.load_raw_files* or imported from `scikit-mobility` using *mobilkit.loader.load_from_skmob*. If using `byArea` the df must contain the `tile_ID` column as returned by *mobilkit.spatial.tessellate*.
- **byArea** (*bool, optional*) – Whether or not to compute the activity per area (default `False`). If `False` will compute the overall activity.
- **timeBin** (*str, optional*) – The width of the time bin to use to aggregate activity. Must be one of the ones found in [pandas time series aliases](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliases). For instance:
  - *'B'* business day frequency
  - *'D'* calendar day frequency
  - *'W'* weekly frequency
  - *'M'* month end frequency
  - *'MS'* month start frequency
  - *'SMS'* semi-month start frequency (1st and 15th)
  - *'BH'* business hour frequency
  - *'H'* hourly frequency
  - *'T','min'* minutely frequency
- **split_out** (*int, optional*) – The number of dask dataframe partitions after the groupby aggregation.

Returns

df_activity – A dataframe with these columns:

- one with the same name as `timeBin` with the date truncated at the selected width.
- `pings` the number of pings recorded in that time bin and area (if `byArea=True`).
- `users` the number of users seen in that time bin and area (if `byArea=True`).
- `users_set` the set of users seen in that time bin and area (if `byArea=True`). Useful to normalize later analysis.
- `pings_per_user` the average number of pings per user in that time bin and area (if `byArea=True`).
- `tile_ID` (if `byArea=True`) the area where the signal has been recorded.

Return type
dask.dataframe

`mobilkit.temporal.`**`computeVolumeProfile`**(*df: DataFrame*, *what: str = 'pings'*, *normalized: bool = True*, *freq='1d'*) → DataFrame

Computes the volume of pings or users in a given interval given by *freq*.

Parameters

- **df** (*dask.dataframe.DataFrame*) – The dataframe containing the pings with at least the `mobilkit.dask_schemas.dttColName` and the `mobilkit.dask_schemas.uidColName` columns.
- **what** (*str*) – *pings users* or *both*, the volume to count.
- **`normalize`** (*bool*) – If *True* will normalize the curve in the 0-1 range, otherwise returns the raw count.
- **`freq`** (*str*) – A valid datetime interval up to which the dates will be floored.

Returns
volume – A dataframe whose index is the time bin and whose value is the observed volume.

Return type
pd.DataFrame

`mobilkit.temporal.`**`filter_daynight_time`**(*df*, *filter_from_h=21.5*, *filter_to_h=8.5*, *previous_day_until_h=4.0*, *daytime_from_h=9.0*, *daytime_to_h=21.0*)

Prepares a raw event df for the ping-based displacement analysis.

Parameters

- **df** (*dask.DataFrame*) – A dataframe containing at least the *uid,datetime,lat,lng* columns as returned by `mobilkit.loader.load_raw_files` or similar functions.
- **filter_{from,to}_h** (*float*) – The starting and ending float hours to consider. If *from_hour<to_hour* only pings whose float hour *h* are *from_hour <= h < to_hour* are considered otherwise all the pings with *h >= from_hour* or *h < to_hour*. Note that float hour *h* for datetime *dt* is *h = dt.hour + dt.minute/60.* so to express 9:45am put *9.75*.
- **previous_day_until_h** (*float*) – All the valid events with float hour *h < previous_day_until_h* will be projected to the previous day. Put 0 or a negative number to keep all events of one day to its *date*.
- **daytime_{from,to}_h** (*float*) – The starting and ending float hours to consider in daytime (other will be put in nightime. All events with *from_hour<= float_hour <= to_hour*

will have a 1 entry in the daytime column, others 0. from hour **must** be smaller than to hour. Note that float hour *h* for datetime *dt* is *h = dt.hour + dt.minute/60.* so to express 9:45am put *9.75*.

Returns

df – The same initial dataframe filtered accordingly to *from_hour,to_hour* and with three additional columns:

- *float_hour*: the day-hour expressed as *h=dt.hour + dt.minutes*
- ***date*: the *datetime* column floored to the day. All events with *float_hour* < *previous_day_until_h* will be further advanced by one day.**
- ***daytime*: 1 if the event's *float_hour* is between *daytime_from_h* and *daytime_to_h***

Return type
dask.DataFrame

mobilkit.temporal.**homeLocationWindow**(*df_hw*, *initial_days_home=None*, *home_days_window=3*, *start_date=None*, *stop_date=None*)

Given a dataframe returned by `mobilkit.stats.userHomeWork` computes, for each user, the home area for every window of `home_days_window` days after the initial date. Note that the points before 12pm will be assigned to the previous day's night and the one after 12pm to the same day's night.

Parameters

- **df_hw** (*dask.dataframe*) – A dataframe as returned by `mobilkit.stats.userHomeWork` with at least the *uid*, *tile_ID*, *datetime* and *isHome* and *isWork* columns.
- **initial_days_home** (*int, optional*) – The number of initial days to be used to compute the original home area. If `None` (default) it will just compute the home for every window since the beginning.
- **home_days_window** (*int, optional*) – The number of days to use to assess the home location of a user (default 3). For each day `d` in the `start_date` to `stop_date - home_days_window` it computes the home location between the `[d, d+home_days_window)` period.
- **start_date** (*datetime.datetime*) – A python datetime object with no timezone telling the date (included) to start from. The default behavior is to keep all the events.
- **stop_date** (*datetime.datetime, optional*) – A python datetime object with no timezone telling the date (excluded) to stop at. Default is to keep all the events.

Returns
df_hwindow – The dataframe containing, for each user and active day of user the `tile_ID` of the user's home and the number of pings recorded there in the time window. The date is saved in `window_date` and refers to the start of the time window (whose index is saved in `timeSlice`). For the initial home window the date corresponds to its end.

Return type
pandas.dataframe

Note: When determining the home location of a user, please consider that some data providers, like _Cuebiq_, obfuscate/obscure/alter the coordinates of the points falling near the user's home location in order to preserve privacy.

This means that you cannot locate the precise home of a user with a spatial resolution higher than the one used to obfuscate these data. If you are interested in the census area (or geohash) of the user's home alone and you

are using a spatial tessellation with a spatial resolution wider than or equal to the one used to obfuscate the data, then this is of no concern.

However, tasks such as stop-detection or POI visit rate computation may be affected by the noise added to data in the user's home location area. Please check if your data has such noise added and choose the spatial tessellation according to your use case.

---

mobilkit.temporal.**plotDisplacement**(*count_users_per_area*, *pivoted*, *gdf*, *area_key='tile_ID'*, *epicenter=[18.584, 98.399]*, *bins=5*)

Parameters

- **count_users_per_area** (*dict*) – The dict returned with the pivot table, the original home location, and the Heaps law of visited areas by *mobilkit.temporal.homeLocationWindow*.
- **pivoted** (*pandas.DataFrame*) – The pivoted dataframe of the visited location during the night as returned with the the original home location, the Heaps law of visited areas and the count of users per area and date by *mobilkit.temporal.homeLocationWindow*.
- **gdf** (*geopandas.GeoDataFrame*) – The geodataframe used to tessellate data. Must contain the *area_key* column.
- **area_key** (*str*) – The column containing the ID of the tessellation areas used to join the displacement data and the GeoDataFrame.
- **epicenter** (*tuple*) – The *(lat,lon)* coordinates of the center to be used to split areas in *bins* bins based on their distance from this point.
- **bins** (*int*) – The number of linear distance bins to compute from the epicenter.

mobilkit.temporal.**plotMonthlyActivity**(*df_activity*, *timeBin*, *what='users'*, *ax=None*, *log_y=False*, ***kwargs*)

Basic function to plot the monthly activity of areas or total region.

Parameters

- **df_activity** (*dask.DataFrame*) – A dataframe as returned from *mobilkit.temporal.computeTimeBinActivity*.
- **timeBin** (*str*) – The width of the time bin used in *mobilkit.temporal.computeTimeBinActivity*.
- **what** (*str, optional*) – The quantity to plot. Must be one amongst `'users'`, `'pings'`, `'pings_per_user'`.
- **ax** (*axis, optional*) – The axis to use. If `None` will create a new figure.
- **log_y** (*bool, optional*) – Whether or not to plot with y log scale. Default `False`.
- ****kwargs** – Will be passed to `seaborn.lineplot` function.

Returns

- **df** (*pandas.DataFrame*) – Thee aggregated data plotted.
- **ax** (*axis*) – The axis of the figure.

mobilkit.tools module

mobilkit.tools.**checkScore**(*results_clusters*, *score='scores'*)

Function to plot the score of clustering.

Parameters

- **results_clusters** (*dict*) – As returned by *mobilkit.tools.computeClusters*.
- **score** (*str, optional*) – One of `"scores", "inconsistents"`. If `scores`, the best split is at a local maximum of the score.

Returns

ax – The ax of the figure.

Return type

axis

mobilkit.tools.**computeClusters**(*results*, *signal*, *metric='cosine'*, *nClusters=[2, 3, 4, 5, 6, 7, 8, 9]*)

Function to compute the clusters.

Parameters

- **results** (*dict*) – As returned from `mobilkit.spatial.computeResiduals`.
- **signal** (*str*) – One of `'mean', 'zscore', 'residual'`, the indicator to use to cluster the areas.
- **metric** (*str, optional*) – One of `'cosine', 'euclidean'`, the metric used to compute the linkage matrix. Default to `cosine`.
- **nClusters** (*list, optinal*) – The list or set of number of clusters to try.

Returns

results_clusters – A dict with the results to be used for plotting and inspection.

Return type

dict

mobilkit.tools.**computeGDFbounds**(*gdf: GeoDataFrame*) → dict

Computes the bounds of a *geopandas.GeoDataFrame gdf* and returns a dictionary with the *min(x,y)* and *max(x,y)* keys and their value as values (minx is the minimum longitude).

Parameters

gdf (*geopandas.GeoDataFrame*) – The GeoDataFrame with at least the `geometry` column.

Returns

bounds – The mapping between the min/max x and y and their values.

Return type

dict

mobilkit.tools.**osrm_time_trip**(*latlon_start*, *latlon_end*, *osrm_url*, *what='duration'*, *max_trip_duration=10800*, *max_trip_distance=100.0*)

Queries an OSRM backend server on *osrm_url* for the travel time or distance (depending on *what*) between *latlon_start* and *latlon_end*.

Parameters

- **latlon_start, latlon_end** (*list or np.array*) – The latitude and longitude of the initial and final trip point (in WGS84, EPSG:4326 crs).
- **osrm_url** (*str*) – A valid table endpoint url of a OSRM-bakend service (like *http://localhost:5000/table/v1/car/*).

- **what** (*str, optional*) – Whether you want the *duration* or *distance* or *duration,distance* of the trip.
- **max_trip_duration** (*float, optional*) – The maximum trip duration in seconds that will be put in place of NaNs or invalid trips.
- **max_trip_distance** (*float, optional*) – The maximum trip distance in km that will be put in place of NaNs or invalid trips.

Returns

duration – The duration (length) of the trip in seconds (meters) or the two (duration,distance) if *what==duration,distance*.

Return type

float or tuple

`mobilkit.tools.`**`plotClustersMap`**(*gdf*, *results_clusters*, *mappings*, *nClusts=5*)

Function to plot the similarity matrix between areas.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – The GeoDataFrame with at least the `tile_ID` and `geometry` columns.
- **results_clusters** (*dict*) – As returned by *mobilkit.tools.computeClusters*.
- **mappings** (*dict*) – The mappings as returned by `mobilkit.spatial.computeResiduals`.
- **nClusts** (*int, optional*) – The number of clusters to use.

Returns

- **gdf** (*geopandas.GeoDataFrame*) – The original GeoDataFrame with an additional column `'cluster'` containing the cluster assigned to that area.
- **ax** (*axis*) – The ax of the figure.

`mobilkit.tools.`**`plotCommunities`**(*results_clusters*, *nClusts*)

Function to plot the similarity matrix between areas.

Parameters

- **results_clusters** (*dict*) – As returned by *mobilkit.tools.computeClusters*.
- **nClusts** (*int, optional*) – The number of clusters to use.

Returns

ax – The ax of the figure.

Return type

axis

`mobilkit.tools.`**`userHomeWorkTravelTimeOSRM`**(*r*, *osrm_url*, *direction='hw'*, *what='duration'*, *max_trip_duration_h=4*, *max_trip_distance_km=150*)

Queries an OSRM backend server on *osrm_url* for the travel time or distance (depending on *what*) between the home and work location for the user in the row *r*.

Parameters

- **r** (*dict or row of a pd.Dataframe*) – The row with the home and work latitude and longitude (in WGS84, EPSG:4326 crs) in the *lat_home* format, as returned by *mobilkit.stats.userHomeWorkLocation*.
- **osrm_url** (*str*) – A valid table endpoint url of a OSRM-bakend service (like *http://localhost:5000/table/v1/car/*).

- **direction** (*str, optional*) – Whether you want the *home->work* ("hw") or *work->home* ("wh") time of the trip.
- **what** (*str, optional*) – Whether you want the *duration*, *distance* or *duration,distance* of the trip.
- **max_trip_duration_h** (*float, optional*) – The maximum trip duration in hours (or km if asking for distance) that will be put in place of NaNs or invalid trips.

Returns
duration – The duration (length) of the trip in seconds (meters) or the two (duration,distance) if *what==duration,distance*.

Return type
float

mobilkit.tools.**visualizeClustersProfiles**(*results_clusters*, *nClusts=5*, *showMean=False*, *showMedian=True*, *showCurves=True*, *together=False*)

Function to plot the temporal profiles of clustering.

Parameters

- **results_clusters** (*dict*) – As returned by `mobilkit.tools.computeClusters`.
- **nClusts** (*int, optional*) – The number of clusters to use.
- **showMean** (*bool, optional*) – Whether or not to show the mean curve of the cluster.
- **showMedian** (*bool, optional*) – Whether or not to show the median curve of the cluster. If both median and mean are to be plotted, only the median will be shown.
- **showCurves** (*bool, optional*) – Whether or not to show the curves of the cluster in transparency.
- **together** (*bool, optional*) – Whether to plot all the cluster in one plot. Default `False`.

Returns
ax – The ax of the figure.

Return type
axis

mobilkit.viz module

mobilkit.viz.**compareLinePlot**(*x_scatter*, *x_line*, *y*, *data*, *xlim=None*, *ylim=None*, *xlabel=None*, *ylabel=None*, *doScatter=True*, *doLine=True*, *scatterkws={}*, *lineplotkws={}*, *figsize=(7, 4)*, *ax=None*)

Compares a scattered data with its line estimated.

Parameters

- **x_scatter, x_line, y** (*str*) – The columns to use for x in the scatter and line plot (in the line plot you might want to use a binned version of the x) and as y.
- **data** (*pd.Dataframe*) – The dataframe to use.
- **xlim, ylim** (*tuple, optional*) – The limits to put in the x and y axis.
- **xlabel, ylabel** (*str, optional*) – The x and y axis labels
- **scatterkws** (*dict, optional*) – The keywords to pass to *seaborn.scatterplot*. By default thay are: *{'alpha':.075}*

- **lineplotkws** (*dict, optional*) – The keywords to pass to *seaborn.lineplot*. By default thay are: `{'color':'C3',

  'estimator':lambda g: np.percentile(g, 50), 'n_boot':200}`

- **figsize** (*tuple, optional*) – The figure size in inches.

Returns
fig, ax – The figure and axes handle.

Return type
tuple

mobilkit.viz.**plot_density_map**(*latitudes*, *longitudes*, *center*, *bins*, *radius*, *ax=None*, *annotations=None*)

Parameters

- **latitudes, longitudes** (*array-like*) – Array contaning the lat and lon coordinates of each user on a selected day.
- **center** (*tuple-like*) – The (lat, lon) of the center where to compute the population density around.
- **bins** (*int*) – The number of bins to use horizontally and vertically in the region around the center.
- **radius** (*float*) – The space to consider above, below, left and right of the center (same unity of the center).
- **ax** (*matplotlib.axes*) – The axes to use. If *None* a new figure will be created.
- **annotations** (*dict*) – A dictionary of annotations to be put on the map in the form of *{"Text": {kwargs to ax.annotate} }*. Will be used as *ax.annotate(key, **value)*.

Returns
The putput of *ax.hist2d* and the axis itself.

Return type
res, ax

mobilkit.viz.**plot_pop**(*df*, *title*, *empiric_pop='POBTOT'*, *data_pop='POP_HFLB'*, *alpha=0.1*, *verbose=True*)

Plot the scatter-plot between empiric_pop and data_pop columns.

Parameters
df (*pandas.DataFrame*) – As dataframe containing the two columns selected.

mobilkit.viz.**shori_density_map**(*data*, *xbins*, *ybins*, *ax=None*, *annotations=None*, *vmin=-2*, *vmax=2*)

Parameters

- **data** (*array-like*) – Array contaning the raster of the population density to be plot.
- **xbins, ybins** (*array-like*) – The bins used to construct the raster. Will be used to limit the plot area.
- **ax** (*matplotlib.axes*) – The axes to use. If *None* a new figure will be created.
- **annotations** (*dict*) – A dictionary of annotations to be put on the map in the form of *{"Text": {kwargs to ax.annotate} }*. Will be used as *ax.annotate(key, **value)*.
- **vmin, vmax** (*float*) – The values to be passed to the colormap.

Returns
The output of *ax.imshow*.

Return type
res

mobilkit.viz.**visualize_boundarymap**(*boundary*)

mobilkit.viz.**visualize_simpleplot**(*df*)

6.9.2 Module contents

The *mobilkit* module documentation.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

L

M

O

P

R

S

T

U

V