SCIKIT-MOBILITY: A PYTHON LIBRARY FOR THE ANALYSIS, GENERATION AND RISK ASSESSMENT OF MOBILITY DATA

Luca Pappalardo
ISTI-CNR, Italy
luca.pappalardo@isti.cnr.it

Filippo Simini
University of Bristol, UK
f.Simini@bristol.ac.uk

Gianni Barlacchi *
FBK, Italy and Amazon, Germany
barlacchi@fbk.eu

Roberto Pellungrini
University of Pisa, Italy
name.surname@gmail.com

ABSTRACT

The last decade has witnessed the emergence of massive mobility data sets, such as tracks generated by GPS devices, call detail records, and geo-tagged posts from social media platforms. These data sets have fostered a vast scientific production on various applications of human mobility analysis, ranging from computational epidemiology to urban planning and transportation engineering. A strand of literature addresses data cleaning issues related to raw spatiotemporal trajectories, while the second line of research focuses on discovering the statistical "laws" that govern human movements. A significant effort has also been put on designing algorithms to generate synthetic trajectories able to reproduce, realistically, the laws of human mobility. Last but not least, a line of research addresses the crucial problem of privacy, proposing techniques to perform the re-identification of individuals in a database. Despite the increasing importance of human mobility analysis for many scientific and industrial domains, a view on state of the art cannot avoid noticing that there is no statistical software that can support scientists and practitioners with all the aspects mentioned above of mobility data analysis. To fill this gap, we propose scikit-mobility, a Python library that has the ambition of providing an environment to reproduce existing research, analyze mobility data, and simulate human mobility habits. scikit-mobility is efficient and easy to use as it extends the well-known standard pandas, a popular Python library for data analysis. Moreover, scikit-mobility provides the user with many functionalities, from visualizing trajectories to generating synthetic data, from analyzing the statistical patterns of trajectories to assessing the privacy risk related to the analysis of mobility data sets.

***K*eywords** data science · human mobility · big data · network science · data mining · python · mathematical modelling · migration models · privacy

* Work done prior joining Amazon

1 Introduction

The last decade has witnessed the emergence of massive data sets of digital traces that portray human movements at an unprecedented scale and detail [Giannotti et al., 2013]. Examples include tracks generated by GPS devices embedded in personal smartphones [Zheng et al., 2008], private vehicles [Pappalardo et al., 2013] or boats [Fernandez Arguedas et al., 2018]; call detail records produced as a by-product of the communication between cellular phones and the mobile phone network [González et al., 2008, Barlacchi et al., 2015]; geotagged posts from the most disparate social media platforms [Noulas et al., 2012]; and even traces describing the activity on the sports fields of amateurs or professional athletes [Rossi et al., 2018]. The availability of big mobility data has attracted enormous interests from scientists of diverse disciplines, fueling advances in several applications, from computational health [Tizzoni et al., 2012, Barlacchi et al., 2017] to the estimation of air pollution [Nyhan et al., 2018], from the design of recommender systems [Wang et al., 2011] to the optimization of mobile and wireless networks [Karamshuk et al., 2011], from transportation engineering and urban planning [Zhao et al., 2016] to the estimation of migratory flows [Simini et al., 2012, Ahmed et al., 2016], from the well-being status of municipalities, regions and countries [Pappalardo et al., 2016b] to the prediction of traffic and future displacements [Zhang et al., 2017, Rossi et al., 2019].

It is hence not surprising that the last decade has also witnessed a vast scientific production on various aspects of human mobility [Blondel et al., 2015, Barbosa et al., 2018]. The first strand of literature addresses data cleaning issues related to mobility data, such as how to extract meaningful locations from raw spatiotemporal trajectories, how to filter, reconstruct, compress and segment them, or how to cluster and classify them [Zheng, 2015]. As a result, in the literature, there is a vast repertoire of techniques that allow scientists and professionals to improve the quality of their mobility data.

The second line of research focuses instead on discovering the statistical laws that govern human mobility. These studies document that, far from being random, human mobility is characterized by predictable patterns, such as a stunning heterogeneity of human travel patterns [González et al., 2008]; a strong tendency to routine and a high degree of predictability of individuals' future whereabouts [Song et al., 2010b]; the presence of the so-called returners/explorers dichotomy [Pappalardo et al., 2015]; the evidence of a conservative quantity in the number of locations actively visited by individuals [Alessandretti et al., 2018], and more. All these quantifiable patterns are universal across different territories and data sources and are usually referred to as the "laws" of human mobility.

The third strand of literature focuses on designing generative algorithms, i.e., mathematical models that can generate synthetic trajectories able to reproduce, realistically, the laws of human mobility. In the literature, a class of algorithms aims to realistically reproduce spatial properties of mobility [Song et al., 2010a, Pappalardo et al., 2016a], while another class of algorithms focuses on the accurate representation of the time-varying behavior of individuals [Barbosa et al., 2015, Alessandretti et al., 2018]. More recently, some approaches rely on machine learning to propose generative algorithms that are realistic with respect to both spatial and temporal properties of human mobility [Pappalardo and Simini, 2018, Jiang et al., 2016]. Although the generation of realistic trajectories is a complex and still open problem, the existing algorithms act as baselines for the evaluation of new approaches.

Finally, a line of research addresses the crucial problem of privacy: people's movements might reveal confidential personal information or allow the re-identification of individuals in a database, creating serious privacy risks [de Montjoye et al., 2013]. Since 2018, the EU General Data Protection Regulation (GDPR) explicitly imposes on data controllers an assessment of the impact of data protection for the riskiest data analyses. Driven by these sensitive issues, in recent years researchers have developed algorithms, methodologies, and frameworks to estimate and mitigate the individual privacy risks associated with the analysis of big data in general [Monreale et al., 2014] and mobility data in particular [Pellungrini et al., 2017].

Despite the increasing importance of human mobility analysis for many scientific and industrial domains, a view on the state of the art cannot avoid noticing that there is no statistical software that can support scientists and practitioners with all the aspects of mobility analysis mentioned above (Section 10).

To fill this gap, we propose *scikit-mobility*, a python library that has the ambition of providing scientists and practitioners with an environment to reproduce existing research and perform analysis of human mobility data. In particular, the library allows the user to:

1. load and represent mobility data, both at the individual and the collective level, through easy-to-use data structures – namely `TrajDataFrame` and `FlowDataFrame` – based on the standard python libraries *numpy*, *pandas* and *geopandas* (Section 2), as well as to visualize trajectories and fluxes on interactive maps based on the python libraries *folium* and *matplotlib* (Section 4);

2. clean and preprocess mobility data using state-of-the-art techniques, such as trajectory clustering, compression, segmentation, and filtering. The library also provides the user with a way to track all the operations performed on the original data (Section 3);

3. analyze mobility data by using the main measures characterizing human mobility patterns both at the individual and at the collective level (Section 5), such as the computation of travel and characteristic distances, user and location entropies, location frequencies, waiting times, origin-destination matrices, and more;
4. run the most popular generative algorithms to simulate individual human mobility, such as Random Walks, the EPR model and its variants (Section 6), and commuting and migratory flows, such as the Gravity Model and the Radiation Model (Section 7);
5. estimate the privacy risk associated with the analysis of a given mobility data set through the simulation of the re-identification risk associated with a vast repertoire of privacy attacks (Section 8).

scikit-mobility is publicly available on GitHub at the following link `https://scikit-mobility.github.io/scikit-mobility/`. The documentation describing all the classes and functions of *scikit-mobility* is available at https://scikit-mobility.github.io/scikit-mobility/.

2 Data Structures

scikit-mobility provides two data structures to deal with raw trajectories and flows between places. Both the data structures are an extension of the DataFrame implemented in the data analysis library *pandas*. Thus, both - `TrajDataFrame` and `FlowDataFrame` - inherit all the functionalities provided by the DataFrame as well as all the efficient optimizations for reading and writing tabular data (e.g., human mobility datasets).

2.1 Trajectory

Mobility data describe the movements of a set of objects during a period of observation. The objects may represent individuals [González et al., 2008], animals [Ramos-Fernández et al., 2004], private vehicles [Pappalardo et al., 2015], boats [Fernandez Arguedas et al., 2018] and even players on a sports field [Rossi et al., 2018]. Mobility data are generally collected in an automatic way as a by-product of human activity on electronic devices (e.g., mobile phones, GPS devices, social networking platforms, video cameras) and stored as *trajectories*, a temporally ordered sequence of spatio-temporal points where an object stopped in or went through. In the literature of human mobility analytics, a trajectory is often formally defined as follows [Zheng et al., 2014, Zheng, 2015]:

Definition 2.1 (Trajectory)**.** The trajectory of an object u is a temporally ordered sequence of tuples $T_u = \langle (l_1, t_1), (l_2, t_2), \ldots, (l_n, t_n) \rangle$, where $l_i = (x_i, y_i)$ is a location, x_i and y_i are the coordinates of the location, and t_i is the corresponding timestamp, with $t_i < t_j$ if $i < j$.

In *scikit-mobility*, a set of trajectories is described by a `TrajDataFrame` (Figure 1), an extension of the *pandas* DataFrame that has specific columns names and data types. A row in the `TrajDataFrame` represents a point of the trajectory, described by three mandatory fields (aka columns): `latitude` (type: float), `longitude` (type: float) and `datetime` (type: datetime). Additionally, two optional columns can be specified. The first one `uid` (type: string) and identifies the object associated with the point of the trajectory. If `uid` is not present, *scikit-mobility* assumes that the `TrajDataFrame` contains trajectories associated with a single moving object. The second one is `tid` and specifies the identifier of the trajectory to which the point belongs to. Similar to `tid`, if `tid` is not present *scikit-mobility* assumes that the `TrajDataFrame` contains a single trajectory. Note that, besides the mandatory columns, the user can add to a `TrajDataFrame` as many columns as they want. Additionally, users can add as many columns as they want since the data structures inherit all the Pandas DataFrame functionalities.

Each `TrajDataFrame` also has two mandatory attributes:

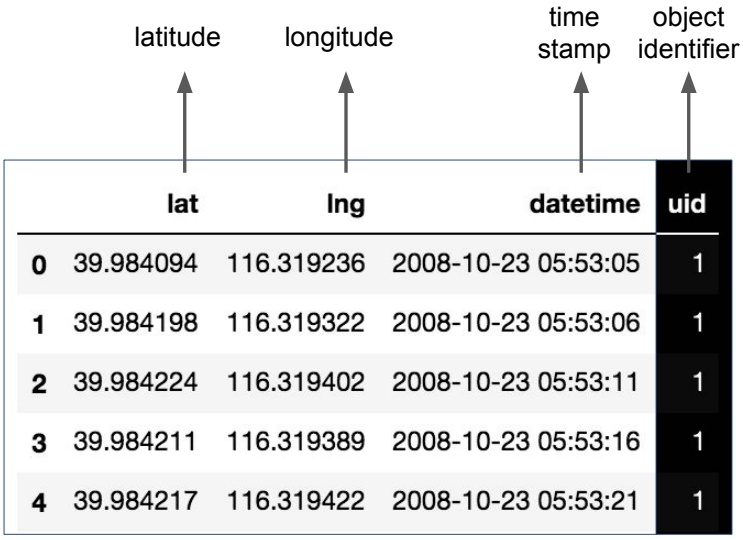
- `crs` (type: dictionary): indicates the coordinate reference system associated with the trajectories. By default it is `epsg:4326` (the latitude/longitude reference system);
- `parameters` (type: dictionary): indicates the list of operations that have been applied to the `TrajDataFrame`. This attribute is a dictionary the key of which is the signature of the function applied (see Section 3 for more details).

scikit-mobility provides functions to create a `TrajDataFrame` from mobility data stored in different formats (e.g., dictionaries, lists, *pandas* DataFrames). To load a `TrajDataFrame` from a file, we first import the library.

```
Python> import skmob
```

Then, we use the method `from_file` of the `TrajDataFrame` class to load the mobility data from the file path.

```
Python> tdf = skmob.TrajDataFrame.from_file('geolife_sample.txt.gz')
```



	lat	lng	datetime	uid
0	39.984094	116.319236	2008-10-23 05:53:05	1
1	39.984198	116.319322	2008-10-23 05:53:06	1
2	39.984224	116.319402	2008-10-23 05:53:11	1
3	39.984211	116.319389	2008-10-23 05:53:16	1
4	39.984217	116.319422	2008-10-23 05:53:21	1

Figure 1: Representation of a `TrajDataFrame`. Each row represents a point of an object's trajectory, described by three mandatory columns (`latitude`, `longitude`, `datetime`) and eventually by the column `uid` and `tid`, indicating the object associated with the point and the trajectory id, respectively.

The `crs` attribute of the loaded `TrajDataFrame` provides the coordinate reference system, while the `parameters` attribute provides a dictionary with meta-information about the data. When we load the data from a file, *scikit-mobility* adds to the `parameters` attribute the key "`from_file`", which indicates the path of the file.

```
Python> print(tdf.crs)

{'init': 'epsg:4326'}

Python> print(tdf.parameters)

{'from_file': 'geolife_sample.txt.gz'}
```

Once loaded, we can visualize a portion of the `TrajDataFrame` using the `print` function and the `head` function, which visualize the first five rows of the `TrajDataFrame`. Note that, since the `uid` column is present in the file, the `TrajDataFrame` created contains the corresponding column.

```
Python> print(tdf.head())

         lat         lng            datetime  uid
0  39.984094  116.319236 2008-10-23 05:53:05    1
1  39.984198  116.319322 2008-10-23 05:53:06    1
2  39.984224  116.319402 2008-10-23 05:53:11    1
3  39.984211  116.319389 2008-10-23 05:53:16    1
4  39.984217  116.319422 2008-10-23 05:53:21    1
```

2.2 Flows

Origin-destination matrices, aka flows, are another common representation of mobility data. While trajectories refer to movements of single objects, flows refer to aggregated movements of objects between a set of locations. An example of flows is the daily commuting flows between the neighborhoods of a city. Formally, we define an origin-destination matrix as:

Definition 2.2 (Origin-Destination matrix or Flows)**.** An Origin-Destination matrix T is a $n \times m$ matrix where n is the number of distinct "origin" locations, m is the number of distinct "destination" locations, T_{ij} is the number of objects traveling from location i to location j.

In *scikit-mobility*, an origin-destination matrix is described by the `FlowDataFrame` structure. A `FlowDataFrame` is an extension of the *pandas* DataFrame that has specific column names and data types. A row in a `FlowDataFrame` represents a flow of objects between two locations, described by three mandatory columns: `origin` (type: string), `destination` (type: string) and `flow` (type: integer). Again, the user can add to a `FlowDataFrame` as many columns as they want. A `FlowDataFrame` has the `Tessellation`, a *geopandas* GeoDataFrame that contains two mandatory columns: `tile_ID` (type: integer) indicates the identifier of a location; `geometry` indicates the polygon (or point) that describes the geometric shape of the location on a territory (e.g., a square, a voronoi shape, the shape of a neighborhood). It is important to note that each location identifier in the `origin` and `destination` columns of a `FlowDataFrame` must be present in the associated `Tessellation`.

The code below loads a `Tessellation` and a `FlowDataFrame` from the corresponding files. First, we import the *scikit-mobility* and the *geopandas* libraries.

```
Python> import skmob
Python> import geopandas as gpd
```

Then, we load the `Tessellation` and the `FlowDataFrame` using the `from_file` method of the classes GeoDataFrame and `TrajDataFrame`, respectively. Note that the `from_file` for loading a `FlowDataFrame` requires to specify the associated `Tessellation` through the "tessellation" argument.

```
Python> tessellation = gpd.GeoDataFrame.from_file("NY_counties_2011.geojson")
Python> fdf = skmob.FlowDataFrame.from_file("NY_commuting_flows_2011.csv",
tessellation=tessellation)
```

The `Tessellation` and `FlowDataFrame` have the structure shown below.

```
Python> print(tessellation.head())
```

```
   tile_ID  population                                           geometry
51   36001      304564  POLYGON ((-73.933672 42.76071, -73.809603 42.7...
23   36003       48787  POLYGON ((-78.308611 42.086675, -78.3088390000...
18   36005     1397366  POLYGON ((-73.783519 40.881033, -73.74806 40.8...
57   36007      199346  POLYGON ((-75.850388 42.327731, -75.8437919999...
20   36009       79819  POLYGON ((-79.06070800000001 42.347917, -79.06...
```

```
Python> print(fdf.head())
```

```
     flow origin destination
0  121606  36001       36001
1       5  36001       36005
2      29  36001       36007
3      11  36001       36017
4      30  36001       36019
```

3 Trajectory preprocessing

As any analytical process, mobility data analysis requires data cleaning and preprocessing steps [Zheng, 2015]. In its current version, the `preprocessing` module allows the user to perform three main preprocessing steps: noise filtering, stop detection, and trajectory compression. Note that, if `TrajDataFrame` contains multiple trajectories from multiple users, the preprocessing methods automatically apply to the single trajectory and, when necessary, to the single object.

3.1 Noise filtering

Trajectory data are in general noisy, usually because of recording errors like poor signal reception. Some of these errors can be fixed directly with map-matching algorithms that adjust the coordinate of points via proper heuristics. However, when the error associated with the coordinates of points is large, the best solution is to filter out these points. In *scikit-mobility*, the standard method `filter` filters out a point if the speed from the previous point is higher than the parameter `max_speed`, which is by default set to 500km/h. To use the `filter` function, we first import the preprocessing module:

```
Python> import skmob
Python> from skmob import preprocessing
```

Then, we apply the filtering, setting max speed as 10 km/h, on a `TrajDataFrame` containing GPS trajectories:

```
Python> tdf = skmob.TrajDataFrame.from_file('geolife_sample.txt.gz')
Python> print(tdf.head())

         lat         lng            datetime  uid
0  39.984094  116.319236 2008-10-23 05:53:05    1
1  39.984198  116.319322 2008-10-23 05:53:06    1
2  39.984224  116.319402 2008-10-23 05:53:11    1
3  39.984211  116.319389 2008-10-23 05:53:16    1
4  39.984217  116.319422 2008-10-23 05:53:21    1

Python> ftdf = preprocessing.filtering.filter(tdf, max_speed_kmh=10.)
Python> print(ftdf.head())

         lat         lng             datetime  uid
0  39.984094  116.319236  2008-10-23 05:53:05  1
1  39.984211  116.319389  2008-10-23 05:53:16  1
2  39.984217  116.319422  2008-10-23 05:53:21  1
3  39.984555  116.319728  2008-10-23 05:53:43  1
4  39.984579  116.319769  2008-10-23 05:53:48  1
```

As we can see, few points are filtered out. The intensity of the filter is controlled by the `max_speed` parameter. Lower is the value, more intense is the filter.

3.2 Stop detection

Some points in a trajectory can represent Point-Of-Interests (POIs) such as schools, restaurants, and bars, or they can represent user-specific places such as home and work locations. These points are usually called *Stay Points* or *Stops*, and they can be detected in different ways. A common approach is to apply spatial clustering algorithms to cluster trajectory points by looking at their spatial proximity [Hariharan and Toyama, 2004]. In *scikit-mobility*, the `stops` function, contained in the `detection` module, finds the stay points visited by an object. For instance, to identify the stops where the object spent at least `minutes_for_a_stop` minutes within a distance `spatial_radius_km` $\times$ `stop_radius_factor`, from a given point, we can use the following code:

```
Python> from preprocessing import detection
Python> stdf = detection.stops(ctdf, stop_radius_factor=0.5, minutes_for_a_stop=20.0,
        spatial_radius_km=0.2, leaving_time=True)

         lat         lng             datetime  uid   leaving_datetime
0  39.978253  116.327275  2008-10-23 14:01:05  1  2008-10-23 18:32:53
1  40.013819  116.306532  2008-10-23 19:10:09  1  2008-10-24 07:46:02
2  39.978987  116.326686  2008-10-24 08:10:39  1  2008-10-24 09:48:57
3  39.981316  116.310181  2008-10-24 09:56:47  1  2008-10-24 11:21:09
4  39.981344  116.309998  2008-10-24 01:55:47  1  2008-10-24 03:20:36
```

As showed in the code snippet, a new column `leaving_datetime` is added to the `TrajDataFrame` in order to indicate the time when the user left the stop location.

3.3 Trajectory compression

The goal of trajectory compression is to reduce the number of trajectory points while preserving the structure of the trajectory. This step is generally applied right after the stop detection step, and it results in a significant reduction of the number of trajectory points. In *scikit-mobility*, we can use one of the methods in the `compression` module under the preprocessing module. For instance, to merge all the points that are closer than $0.2km$ from each other, we can use the following code:

```
Python> from preprocessing import compression
Python> print(ftdf.head())
```

```
         lat         lng            datetime  uid
0  39.984094  116.319236 2008-10-23 05:53:05  1
1  39.984211  116.319389 2008-10-23 05:53:16  1
2  39.984217  116.319422 2008-10-23 05:53:21  1
3  39.984555  116.319728 2008-10-23 05:53:43  1
4  39.984579  116.319769 2008-10-23 05:53:48  1
```

```
Python> ctdf = compression.compress(ftdf, spatial_radius_km=0.2)
```

```
   lat        lng         datetime            uid
0  39.984334  116.320778  2008-10-23 05:53:05  1
1  39.979642  116.322241  2008-10-23 05:58:33  1
2  39.978051  116.327538  2008-10-23 06:01:47  1
3  39.970511  116.341455  2008-10-23 10:32:53  1
```

Once compressed, the trajectory will present a smaller number of points, allowing then an easy plotting of them by using the data visualization functionalities of *scikit-mobility* described in Section 4.

4 Plotting

One of the use cases for *scikit-mobility* is the exploratory data analysis of mobility data sets, which includes the visualization of trajectories and flows. To this end, both `TrajDataFrame` and `FlowDataFrame` have methods that allow the user to produce interactive visualizations generated using the python library *folium*.

4.1 Visualizing trajectories

A `TrajDataFrame` has three main plotting methods: `plot_trajectory` plots a line connecting the trajectory points on a map; `plot_stops` plots the location of stops on a map; and `plot_diary` plots the sequence of visited locations over time.

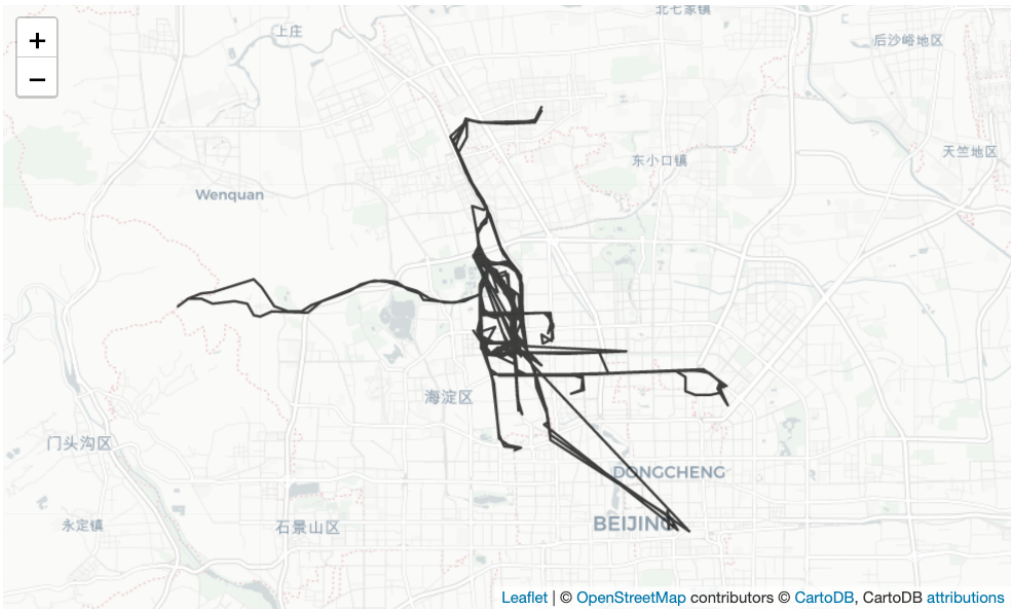
4.1.1 Plot trajectories

The `TrajDataFrame`'s method `plot_trajectory` plots the time-ordered trajectory points connected by straight lines on a map. If the column `uid` is present and contains more than one object, the trajectory points are first grouped by `uid` and then sorted by `datetime`. Large `TrajDataFrames` with many points can be computationally intensive to visualize. Two arguments can be used to reduce the amount of data to plot: `max_users` (type: `int`, default: 10) limits the number of objects whose trajectories should be plotted, while `max_points` (type: `int`, default: 1000) limits the number of trajectory points per object to plot, i.e., if necessary, an object's trajectory will be down-sampled and at most `max_points` points will be plotted. The plot style can be customized via arguments to specify the color, weight, and opacity of the trajectory lines, as well as the type of map tiles to use. The user can also plot markers denoting the start points and the end points of the trajectory.

The method returns a `folium.Map` object, which can be used by other *folium* and *scikit-mobility* functions in order to visualize additional data on the same map. A `folium.Map` object can be passed to `plot_trajectory` via the argument `map_f` (default: `None`, which means that the trajectory is plotted on a new map).

An example of plot generated by the `plot_trajectory` method is shown below:

```
Python> import skmob
Python> tdf = skmob.TrajDataFrame.from_file('geolife_sample.txt.gz')
Python> map_f = tdf.plot_trajectory(max_users=1, hex_color='#000000')
Python> map_f
```



4.1.2 Plot stops

The `TrajDataFrame`'s method `plot_stops` plots the locations of the stops as markers on a map. This method requires a `TrajDataFrame` with the column `constants.LEAVING_DATETIME`, which is created by the *scikit-mobility* functions to detect stops (see 3). The argument `max_users` (type: `int`, default: 10) limits the number of objects whose stops should be plotted. The plot style can be customized via arguments to specify the color, radius, and opacity of the markers, as well as the type of the map tiles to use. The argument `popup` (default: `False`) allows enhancing the plot's interactivity displaying popup windows that appear when the user clicks on a marker. A stop's popup window includes information like coordinates, object's `uid`, arrival, and leaving times.

The method returns a `folium.Map` object, which can be used by other *folium* and *scikit-mobility* functions in order to visualize additional data on the same map. A `folium.Map` object can be passed to `plot_stops` via the argument `map_f` (default: `None`, which means that the stops are plotted on a new map).

We show below an example of a plot generated by the `plot_stops` method. Note that if the `cluster` column is present in the `TrajDataFrame`, as it happens for instance when the `cluster` method is applied (Section 3), the stops are automatically colored according to the value of that column (so as to identify different clusters of stops).

```
Python> from skmob.preprocessing import detection, clustering
Python> tdf = skmob.TrajDataFrame.from_file('geolife_sample.txt.gz')
Python> stdf = detection.stops(tdf)
Python> cstdf = clustering.cluster(stdf)
Python> cstdf.plot_stops(max_users=1, map_f=mapf)
```

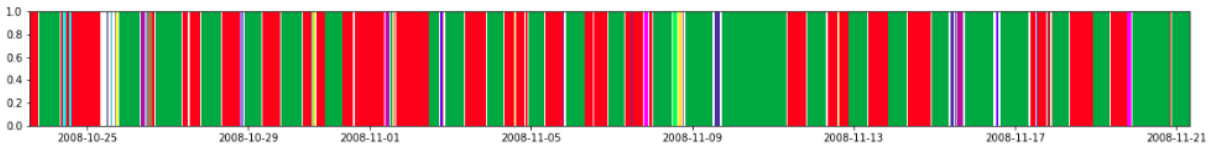
4.1.3 Plot diary

The `TrajDataFrame`'s method `plot_diary` plots the time series of the locations visited by an object. If the column `uid` is present, one object ID must be specified via the argument `user`. This method requires a `TrajDataFrame` with the column `constants.CLUSTER`, which is created by the *scikit-mobility* functions to cluster stops (see 3).

The plot displays time on the *x* axis and shows a series of rectangles of different colors that represent the object's visits to the various stops. The length of a rectangle denotes the duration of the visit: the left edge marks the arrival time, the right edge marks the leaving time. The color of a rectangle denotes the stop's cluster: visits to stops that belong to the same cluster have the same color (the color code is consistent with the one used by the method `plot_stops`). A white rectangle indicates that the object is moving.

We show below an example of a plot generated by the `plot_diary` method:

```
Python> cstdf.plot_diary(user='001')
```



4.2 Visualizing flows

A `FlowDataFrame` has two main plotting methods: `plot_tessellation` plots the tessellation's tiles on a geographic map and `plot_flows` plots, on a geographic map, the lines connecting the centroids of the tessellation's tiles between which flows are present.

4.2.1 Plot tessellation

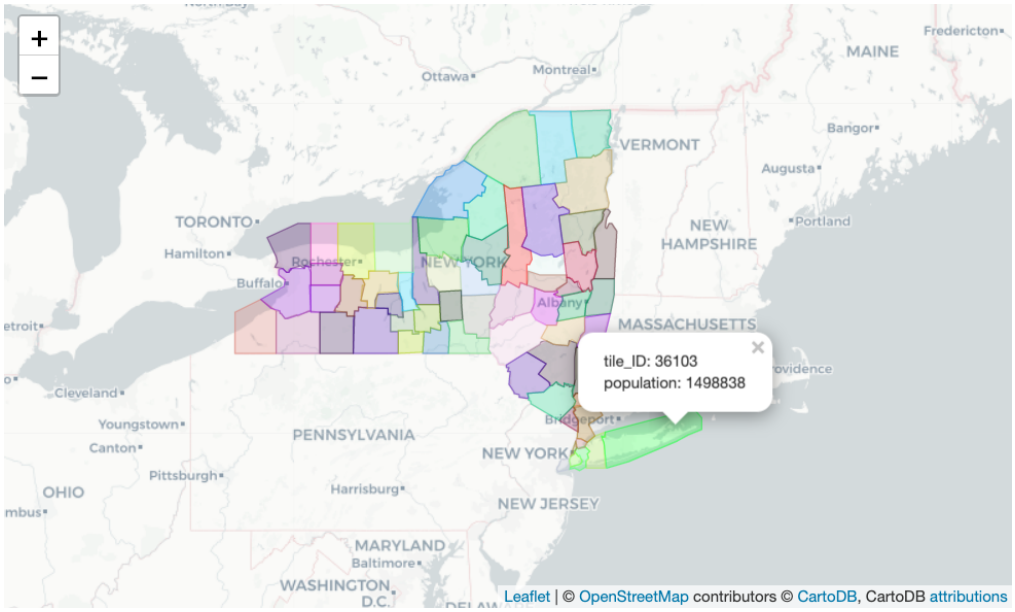
The `FlowDataFrame`'s method `plot_tessellation` plots the `GeoDataFrame` associated with a `FlowDataFrame` on a geographic map. Large tessellations with many tiles can be computationally intensive to visualize. The argument `maxitems` can be used to limit the number of tiles to plot (default: -1, which means that all tiles are displayed).

The plot style can be customized via arguments to specify the color and opacity of the tiles, as well as the type of map tiles to use. The argument `popup_features` (type: `list`, default: `[constants.TILE_ID]`) allows to enhance the plot's interactivity displaying popup windows that appear when the user clicks on a tile and includes information contained in the columns of the tessellation's `GeoDataFrame` specified in the argument's list.

The method returns a `folium.Map` object, which can be used by other *folium* and *scikit-mobility* functions in order to visualize additional data on the same map. A `folium.Map` object can be passed to `plot_flows` via the argument `map_osm` (default: `None`, which means that the tessellation is plotted on a new map).

We show below an example of a plot generated by the `plot_tessellation` method:

```
Python> import geopandas as gpd
Python> from skmob import FlowDataFrame
Python> tessellation = gpd.GeoDataFrame.from_file('./NY_counties_2011.geojson')
Python> fdf = FlowDataFrame.from_file('./NY_commuting_flows_2011.csv',
                                      tessellation=tessellation)
Python> fdf.plot_tessellation(popup_features=['tile_ID', 'population'])
```

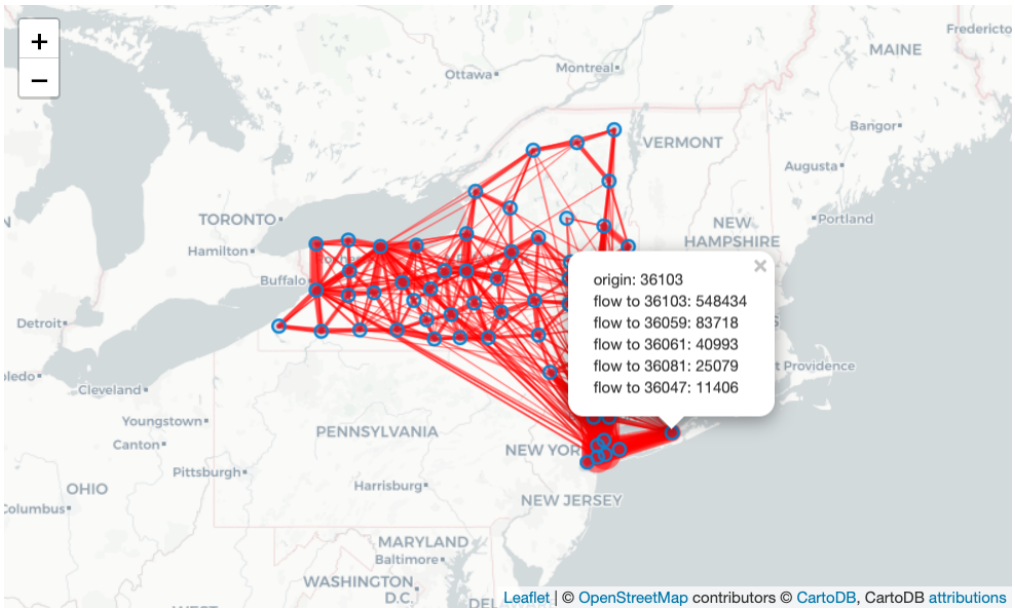


4.2.2 Plot flows

The `FlowDataFrame`'s method `plot_flows` plots the flows on a geographic map as lines between the centroids of the tiles in the `FlowDataFrame`'s tessellation. Large `FlowDataFrames` with many origin-destination pairs can be computationally intensive to visualize. The argument `min_flow` (type: integer, default: 0) can be used to specify that only flows larger than `min_flow` should be displayed. The thickness of each line is a function of the flow and can be specified via the arguments `flow_weight`, `flow_exp` and `style_function`. The plot style can be further customized via arguments to specify the color and opacity of the flow lines, as well as the type of map tiles to use. The arguments `flow_popup` and `tile_popup` allow to enhance the plot's interactivity displaying popup windows that appear when the user clicks on a flow line or a circle in an origin location, respectively, and include information on the flow or the flows from a location. The method returns a `folium.Map` object, which can be used by other *folium* and *scikit-mobility* functions in order to visualize additional data on the same map. A `folium.Map` object can be passed to `plot_flows` via the argument `map_f` (default: `None`, which means that the flows are plotted on a new map).

We show below an example of a plot generated by the `plot_flows` method:

```
Python> fdf.plot_flows(min_flow=50)
```



5 Mobility measures

In the last decade, several measures have been proposed to capture the patterns of human mobility, both at the individual and collective levels. Individual measures summarize the mobility patterns of a single moving object, while collective measures summarize mobility patterns of a population as a whole. For instance, the so-called radius of gyration [González et al., 2008] and its variants [Pappalardo et al., 2015] quantify the characteristic distance traveled by an individual, while several measures inspired by the Shannon entropy have been proposed to quantify the predictability of an individual's movements [Song et al., 2010b].

scikit-mobility provides a wide set of mobility measures, each implemented as a function that takes in input a `TrajDataFrame` and outputs a *pandas* DataFrame. Individual and collective measures are implemented the in `skmob.measure.individual` and the `skmob.measures.collective` modules, respectively.

The code below computes two measures: the distances traveled by the objects and their radius of gyration. First, we import the two functions from the library.

```
Python> import skmob
Python> from skmob.measures.individual import jump_lengths, radius_of_gyration
```

Then, we invoke the two functions on the `TrajDataFrame`, respectively.

```
Python> jl_df = jump_lengths(tdf)
Python> rg_df = radius_of_gyration(tdf)
```

The output of the functions is a *pandas* DataFrame with two columns: `uid` contains the identifier of the object; the second column, the name of which corresponds to the name of the invoked function, contains the computed measure for that object. For example, in the DataFrame `jl_df`, the column `jump_length` of the DataFrame contains a list of all distances traveled by that object.

```
Python> print(jl_df.head())
```

```
   uid                                       jump_lengths
0    0  [19.640467328877936, 0.0, 0.0, 1.7434311010381...
1    1  [6.505330424378251, 46.75436600375988, 53.9284...
2    2  [0.0, 0.0, 0.0, 0.0, 3.6410097195943507, 0.0, ...
3    3  [3861.2706300798827, 4.061631313492122, 5.9163...
4    4  [15511.92758595804, 0.0, 15511.92758595804, 1....
```

Similary, in the DataFrame `rg_df` the column `radius_of_gyration` contains the radius of gyration for that object.

```
Python> print(rg_df.head())

   uid  radius_of_gyration
0    0         1564.436792
1    1         2467.773523
2    2         1439.649774
3    3         1752.604191
4    4         5380.503250
```

Note that, if the optional column `uid` is not present in the input `TrajDataFrame`, a simple Python structure is outputted instead of the *pandas* DataFrame (e.g., a list for function `jump_lengths` and a float for function `radius_of_gyration`).

Collective measures are used in a similar way. The code below computes a collective measure - the number of visits per location (by any object). First, we import the function.

```
Python> import skmob
Python> from skmob.measures.collective import visits_per_location
```

Then, we invoke the function on the `TrajDataFrame`.

```
Python> vpl_df = visits_per_location(tdf)
```

As for the individual measures, the output of the functions is a *pandas* DataFrame. The format of this DataFrame depends on the measures. For example, in the DataFrame `vpl_df` there are three columns: `lat` and `lng` indicate the coordinates of a location, and `n_visits` indicate the number of visits to that location in the `TrajDataFrame`.

```
Python> print(vpl_df.head())


    lat       lng            n_visits
    43.717999 10.902612      5338
    42.785016 11.110376      4717
    42.934046 10.783248      4354
    43.847140 11.142547      4201
    42.930131 10.764460      4169
```

6 Individual Generative Algorithms

The goal of generative algorithms of human mobility is to create a population of agents whose mobility patterns are statistically indistinguishable from those of real individuals [Pappalardo and Simini, 2018]. A generative algorithm typically generates a synthetic trajectory corresponding to a single moving object, assuming that an object is independent of the others. *scikit-mobility* implements the most common individual generative algorithms, such as the Exploration and Preferential Return model [Song et al., 2010a] and its variants [Pappalardo et al., 2016a, Barbosa et al., 2015, Alessandretti et al., 2018], and DITRAS [Pappalardo and Simini, 2018]. Each generative algorithm is a python class. First, we instantiate the algorithm. Then we invoke the `generate` method to start the generation of synthetic trajectories.

The code below shows the code to generate a `TrajDataFrame` describing the synthetic trajectory of 1000 agents that move between the locations of a `Tessellation` and for a period specified in the input. First, we import the class of the generative algorithm (`DensityEPR`) from the library.

```
Python> import skmob
Python> import pandas as pd
Python> import geopandas as gpd
Python> from skmob.models.epr import DensityEPR
```

Then, we load the spatial tessellation on which the agents have to move from a file as a `Tessellation` object, and we specify the start and end times of the simulation as pandas datetime objects.

```
Python> tessellation = gpd.GeoDataFrame.from_file("NY_counties_2011.geojson")
Python> start_time = pd.to_datetime('2019/01/01 08:00:00')
Python> end_time = pd.to_datetime('2019/01/14 08:00:00')
```

Finally, we instantiate the `DensityEPR` model and start the simulation through the `generate` method, which takes in input the start and end times, the `Tessellation`, the number of agents, and other model-specific parameters. The output of the simulation is a `TrajDataFrame` containing the trajectory of the 1000 agents.

```
Python> depr = DensityEPR()
Python> tdf = depr.generate(start_time, end_time, tessellation, n_agents=1000)
Python> print(tdf.head())
```

```
   uid                 datetime        lat        lng
0    1 2019-01-01 08:00:00.000000  40.878457 -72.874961
1    1 2019-01-01 09:50:59.482870  40.848411 -73.862331
2    1 2019-01-01 10:37:10.348375  40.684857 -73.843043
3    1 2019-01-01 11:07:19.660858  40.848411 -73.862331
4    1 2019-01-01 12:01:56.298731  40.684857 -73.843043
```

7 Collective Generative Algorithms

Collective generative algorithms estimate spatial flows between a set of discrete locations. Examples of spatial flows estimated with collective generative algorithms include commuting trips between neighborhoods, migration flows between municipalities, freight shipments between states, and phone calls between regions [Barbosa et al., 2018].

In *scikit-mobility*, a collective generative algorithm takes in input a `Tessellation`.

To be a valid input for a collective algorithm, the `Tessellation` should contain two columns, `geometry` and `relevance`, which are necessary to compute the two variables used by collective algorithms: the distance between tiles and the importance (aka "attractiveness") of each tile. A collective algorithm produces a `FlowDataFrame` that contains the generated flows and the `Tessellation` of which is the one specified as the algorithm's input.

scikit-mobility implements the most common collective generative algorithms: the Gravity model [Zipf, 1946, Wilson, 1971] and the Radiation model [Simini et al., 2012]. We illustrate how to work with generative algorithms in *scikit-mobility* with an example based on the Gravity model.

The class `Gravity`, implementing the Gravity model, has two main methods: `fit`, which calibrates the model's parameters using a training `FlowDataFrame`; and `generate`, which generates the flows on a given tessellation. The following code shows how to use both methods to estimate the commuting flows between the counties in the state of New York. First, we load the tessellation from a file:

```
Python> import skmob
Python> import geopandas as gpd
Python> tessellation = gpd.GeoDataFrame.from_file("NY_counties_2011.geojson")
Python> print(tessellation.head())
```

```
  tile_id  population                                           geometry
0   36019       81716  POLYGON ((-74.006668 44.886017, -74.027389 44....
1   36101       99145  POLYGON ((-77.099754 42.274215, -77.0996569999...
2   36107       50872  POLYGON ((-76.25014899999999 42.296676, -76.24...
3   36059     1346176  POLYGON ((-73.707662 40.727831, -73.700272 40....
4   36011       79693  POLYGON ((-76.279067 42.785866, -76.2753479999...
```

The tessellation contains the column `population`, used as relevance variable for each tile (county). Next, we load the observed commuting flows between the counties from file:

```
Python> import skmob
Python> fdf = skmob.FlowDataFrame.from_file("NY_commuting_flows_2011.csv",
        tessellation=tessellation)
Python> print(fdf.head())
```

```
     flow origin destination
0  121606  36001       36001
1       5  36001       36005
2      29  36001       36007
```

```
3       11  36001       36017
4       30  36001       36019
```

Let us use the observed flows to fit the parameters of a singly-constrained gravity model with the power-law deterrence function (for more details on the gravity models see [Barbosa et al., 2018]. First, we instantiate the model:

```
Python> from skmob.models.gravity import Gravity
Python> gravity = Gravity(gravity_type='singly constrained')
Python> print(gravity)

Gravity(name="Gravity model", deterrence_func_type="power_law",
deterrence_func_args=[-2.0], origin_exp=1.0, destination_exp=1.0,
gravity_type="singly constrained")
```

Then we call the method `fit` to fit the parameters from the previously loaded `FlowDataFrame`:

```
Python> gravity.fit(fdf, relevance_column='population')
Python> print(gravity)

Gravity(name="Gravity model", deterrence_func_type="power_law",
deterrence_func_args=[-1.99471520], origin_exp=1.0, destination_exp=0.64717595,
gravity_type="singly constrained")
```

Finally, we use the fitted model to generate the flows on the same tessellation. Setting the argument `out_format="probabilities"` we specify that in the column `flow` of the returned `FlowDataFrame` we want the probability to observe a unit flow (trip) between two tiles.

```
Python> fdf_fitted = gravity.generate(tessellation,
        relevance_column='population', out_format='probabilities')
Python> print(fdf_fitted.head())

  origin destination      flow
0  36019       36101  0.003340
1  36019       36107  0.002227
2  36019       36059  0.037601
3  36019       36011  0.004859
4  36019       36123  0.001080
```

8 Privacy Risk Assessment

Mobility data is sensitive since the movements of individuals can reveal confidential personal information or allow the re-identification of individuals in a database, creating serious privacy risks [de Montjoye et al., 2013]. Indeed the General Data Protection Regulation (GDPR) explicitly imposes on data controllers an assessment of the impact of data protection for the riskiest data analyses. For this reason, *scikit-mobility* provides scientists in the field of mobility analysis with tools to estimate the privacy risk associated with the analysis of a given data set.

In the literature, privacy risk assessment relies on the concept of re-identification of a moving object in a database through an attack by a malicious adversary [Pellungrini et al., 2017]. A common framework for privacy risk assessment [Pratesi et al., 2018] assumes that during the attack a malicious adversary acquires, in some way, the access to an anonymized mobility data set, i.e., a mobility data set in which the moving object associated with a trajectory is not known. Moreover, it is assumed that the malicious adversary acquires, in some way, information about the trajectory (or a portion of it) of an individual represented in the acquired data set. Based on this information, the risk of re-identification of that individual is computed estimating how unique that individual's mobility data are with respect to the mobility data of the other individuals represented in the acquired data set [Pellungrini et al., 2017].

scikit-mobility provides several attack models, each implemented as a python class. For example in a location attack model, implemented in the `LocationAttack` class, the malicious adversary knows a certain number of locations visited by an individual, but they do not know the temporal order of the visits [Pellungrini et al., 2017]. To instantiate a `LocationAttack` object we can run the following code:

```
Python> import skmob
Python> from skmob.privacy import attacks
Python> at = attacks.LocationAttack(knowledge_length=2)
```

The argument `knowledge_length` specifies how many locations the malicious adversary knows of each object's movement. The re-identification risk is computed based on the worst possible combination of `knowledge_length` locations out of all possible combinations of locations.

To assess the re-identification risk associated with a mobility data set, represented as a `TrajDataFrame`, we specify it as input to the `assess_risk` method, which returns a *pandas* DataFrame that contains the `uid` of each object in the `TrajDataFrame` and the associated re-identification risk as the column `risk` (type: float, range: $[0, 1]$ where 0 indicates minimum risk and 1 maximum risk).

```
Python> tdf = TrajDataFrame.from_file(filename="privacy_sample.csv")
Python> tdf_risk = at.assess_risk(tdf)
Python> print(tdf_risk.head())
```

```
  uid        risk
0   1    0.333333
1   2    0.500000
2   3    0.333333
3   4    0.333333
4   5    0.250000
```

Since risk assessment may be time-consuming for more massive datasets, *scikit-mobility* provides the option to focus only on a subset of the objects with the argument `targets`. For example, in the following code, we compute the re-identification risk for the object with `uid` 1 and 2 only:

```
Python> tdf_risk = at.assess_risk(tdf, targets=[1,2])
Python> print(tdf_risk)
```

```
  uid        risk
0   1    0.333333
1   2    0.500000
```

During the computation, not necessarily all combinations of locations are evaluated when assessing the re-identification risk of a moving object: when the combination with maximum re-identification risk (e.g., risk 1) is found for a moving object, all the other combinations are not computed, so as to make the computation faster. However, if the user wants that all combinations are computed anyway, they can set the argument `force_instances` (type: boolean, default: `False`) to `True`:

```
Python> tdf_risk = at.assess_risk(tdf, targets=[2], force_instances=True)
Python> print(tdf_risk)
```

```
        lat       lon            datetime uid instance instance_elem     risk
0 43.843014 10.507994 2011-02-03 08:34:04   1        1             1 0.333333
1 43.544270 10.326150 2011-02-03 09:34:04   1        1             2 0.333333
2 43.843014 10.507994 2011-02-03 08:34:04   1        2             1 0.250000
3 43.544270 10.326150 2011-02-03 09:34:04   1        2             2 0.250000
4 43.779250 11.246260 2011-02-04 10:34:04   1        3             1 0.250000
5 43.708530 10.403600 2011-02-03 10:34:04   1        3             2 0.250000
```

The result is a *pandas* DataFrame that contains and a reference number of each combination under the attribute `instance` and, for each instance, the `risk` and each of the locations comprising that instance indicated by the attribute `instance_elem`.

9 Conclusion and Future Developments

In this paper, we presented *scikit-mobility*, a new python library for the analysis, generation, and privacy risk assessment of mobility data. *scikit-mobility* allows the user to manage two basic types of mobility data – trajectories and fluxes – and it provides several modules, each dedicated to a specific aspect of mobility data analysis.

	data type	**processing**	**plotting**	**measures**	**models**	**privacy**
scikit-mobility	many	yes	yes	yes	yes	yes
bandicoot	mobile phone	yes	-	yes	-	-
movingpandas	many	yes	-	-	-	-
PySal	many	-	yes	-	-	-

Table 1: Comparison of *scikit-mobility* with other libraries that cover similar aspects of spatio-temporal and mobility data.

We imagine two future directions for the development of *scikit-mobility*. On one side, we plan to add more modules to cover a more extensive range of aspects regarding mobility data analysis. For example, we plan to include algorithms for predicting the next location visited by an individual [Wu et al., 2018], or for performing map matching. i.e., assigning the points of a trajectory to the street network.

On the other hand, we plan to improve the library from a computational point of view. Although in its current version *scikit-mobility* is easy to use and it is rather efficient on mobility data sets in the order of gigabytes, it is not scalable to massive mobility data in the order of terabytes. Since new python libraries similar to *pandas* but more computationally efficient are being developed every year (e.g., *dask*), we plan to re-implement crucial functions in *scikit-mobility* so that they can exploit the computational efficiency of these libraries. This aspect, which is not crucial now, will become so when the library will be (hopefully) largely adopted by the scientific community.

10 Existing tools

A review of state of the art reveals that several libraries are dealing with trajectory data in `R` [Pebesma, 2018]. Unfortunately, none of them covers comprehensively all the aspects that are specific to mobility data analysis, e.g., statistical laws, generative models, privacy risk assessment, pre-processing functions.

Recently, some python libraries have been proposed to manage and manipulate mobility data. In this section, we revise the libraries that are the most similar in their purpose to what we propose in this paper, highlighting the differences between them and *scikit-mobility* (Table 1).

Bandicoot

bandicoot [de Montjoye et al., 2016] is a python library for the analysis of mobile phone metadata that provides the users with functions to compute features related to mobile phone usage. These features are grouped into three categories: *(i)* individual features describe an individual's mobile phone usage and interactions with their contacts; *(ii)* spatial features describe an individual's mobility patterns; *(iii)* social network features describe an individual's social network.

The principal limit of *bandicoot* is that it is specifically designed for managing a specific data type, namely mobile phone data. This design choice makes *bandicoot* unsuitable for the analysis of movements that cannot be captured by mobile phone data, such as car travels, movements of animals, or boat trips. In contrast, *scikit-mobility* gives the user the possibility to deal with a diverse set of mobility data sources (e.g., GPS data, social media data, mobile phone data) and covers a much complete set of standard mobility measures. Moreover, *scikit-mobility* provides a module dedicated to the privacy risk assessment of any mobility data source, a module to create interactive geographic plots, and a module dedicated to generative models of individual and collective mobility, all features that are completely absent in *bandicoot*.

Moving Pandas

movingpandas [Graser, 2019] is an extension to the Python data analysis library *pandas* and its spatial extension *geopandas* to add functionality for dealing with trajectory data. In *movingpandas*, a trajectory is a time-ordered series of geometries. These geometries and associated attributes are stored in a `GeoDataFrame`, a data structure provided by the *geopandas* library. The main advantage of *movingpandas* is that, being based on *geopandas*, it allows the user to perform several operations on trajectories, such as clipping them with polygons and computing intersections with polygons. However, since it is focused on the concept of trajectory, *movingpandas* does not implement any features that are specific of mobility analysis, such as statistical laws of mobility, generative models, standard pre-processing functions, and methods to assess privacy risk in mobility data.

PySal

PySal [Rey and Anselin, 2007] is an open source python library for geospatial data science with an emphasis on geospatial vector data. It supports the development of high-level applications for spatial analysis, such as detection of spatial clusters, construction of graphs from spatial data, spatial regression and statistical modeling on geographically embedded networks, spatial econometrics exploratory spatiotemporal data analysis. A useful feature of *PySal* is its visualization module, which allows the user to visualize several types of geographic maps. Unfortunately, since it is focused on geospatial operations and computational geometry, *PySal* does not provide any function to compute statistical laws of mobility, to generate synthetic data through generative models, to perform pre-processing and to assess privacy risk in mobility data.

Acknowledgments

The development of the library has been partially supported by EU project SoBigData RI grant #654024, EU project Track&Know H2020 grant #780754 and by EPSRC First Grant EP/P012906/1. We thank Anita Graser for the useful discussions and Riccardo Gallotti for his suggestions.

Author contributions

L.P. developed modules and performed experiments, and structured the paper. F.S. performed experiments, tested the code and developed modules. G.B. performed the code, the system design and developed modules. R.P. performed experiments and developed modules. All the authors contributed to writing of the manuscript.

References

[Ahmed et al., 2016] Ahmed, M. N., Barlacchi, G., Braghin, S., Calabrese, F., Ferretti, M., Lonij, V., Nair, R., Novack, R., Paraszczak, J., and Toor, A. S. (2016). A multi-scale approach to data-driven mass migration analysis. In *SoGood@ ECML-PKDD*.

[Alessandretti et al., 2018] Alessandretti, L., Sapiezynski, P., Sekara, V., Lehmann, S., and Baronchelli, A. (2018). Evidence for a conserved quantity in human mobility. *Nature human behaviour*, 2(7):485.

[Barbosa et al., 2018] Barbosa, H., Barthelemy, M., Ghoshal, G., James, C. R., Lenormand, M., Louail, T., Menezes, R., Ramasco, J. J., Simini, F., and Tomasini, M. (2018). Human mobility: Models and applications. *Physics Reports*, 734:1 – 74.

[Barbosa et al., 2015] Barbosa, H., de Lima-Neto, F. B., Evsukoff, A., and Menezes, R. (2015). The effect of recency to human mobility. *EPJ Data Science*, 4(1):21.

[Barlacchi et al., 2015] Barlacchi, G., De Nadai, M., Larcher, R., Casella, A., Chitic, C., Torrisi, G., Antonelli, F., Vespignani, A., Pentland, A., and Lepri, B. (2015). A multi-source dataset of urban life in the city of milan and the province of trentino. *Scientific data*, 2:150055.

[Barlacchi et al., 2017] Barlacchi, G., Perentis, C., Mehrotra, A., Musolesi, M., and Lepri, B. (2017). Are you getting sick? predicting influenza-like symptoms using human mobility behaviors. *EPJ Data Science*, 6(1):27.

[Blondel et al., 2015] Blondel, V. D., Decuyper, A., and Krings, G. (2015). A survey of results on mobile phone datasets analysis. *EPJ Data Science*, 4:1–55.

[de Montjoye et al., 2013] de Montjoye, Y.-A., Hidalgo, C. A., Verleysen, M., and Blondel, V. D. (2013). Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3:1376.

[de Montjoye et al., 2016] de Montjoye, Y.-A., Rocher, L., and Pentland, A. S. (2016). bandicoot: a python toolbox for mobile phone metadata. *Journal of Machine Learning Research*, 17(175):1–5.

[Fernandez Arguedas et al., 2018] Fernandez Arguedas, V., Pallotta, G., and Vespe, M. (2018). Maritime traffic networks: From historical positioning data to unsupervised maritime traffic monitoring. *IEEE Transactions on Intelligent Transportation Systems*, 19(3):722–732.

[Giannotti et al., 2013] Giannotti, F., Pappalardo, L., Pedreschi, D., and Wang, D. (2013). *A Complexity Science Perspective on Human Mobility*, page 297–314. Cambridge University Press.

[González et al., 2008] González, M. C., Hidalgo, C. A., and Barabási, A.-L. (2008). Understanding individual human mobility patterns. *Nature*, 453(7196):779–782.

[Graser, 2019] Graser, A. (2019). Movingpandas: Efficient structures for movement data in python. *Journal of Geographic Information Science*, 1:54–68.

[Hariharan and Toyama, 2004] Hariharan, R. and Toyama, K. (2004). Project lachesis: parsing and modeling location histories. In *International Conference on Geographic Information Science*, pages 106–124. Springer.

[Jiang et al., 2016] Jiang, S., Yang, Y., Gupta, S., Veneziano, D., Athavale, S., and González, M. C. (2016). The timegeo modeling framework for urban mobility without travel surveys. *Proceedings of the National Academy of Sciences*, 113(37):E5370.

[Karamshuk et al., 2011] Karamshuk, D., Boldrini, C., Conti, M., and Passarella, A. (2011). Human mobility models for opportunistic networks. *IEEE Communications Magazine*, 49(12):157–165.

[Monreale et al., 2014] Monreale, A., Rinzivillo, S., Pratesi, F., Giannotti, F., and Pedreschi, D. (2014). Privacy-by-design in big data analytics and social mining. *EPJ Data Science*, 3(1):10.

[Noulas et al., 2012] Noulas, A., Scellato, S., Lambiotte, R., Pontil, M., and Mascolo, C. (2012). A tale of many cities: Universal patterns in human urban mobility. *PLOS ONE*, 7(5):1–10.

[Nyhan et al., 2018] Nyhan, M. M., Kloog, I., Britter, R., Ratti, C., and Koutrakis, P. (2018). Quantifying population exposure to air pollution using individual mobility patterns inferred from mobile phone data. *Journal of Exposure Science & Environmental Epidemiology*, 29:238–247.

[Pappalardo et al., 2013] Pappalardo, L., Rinzivillo, S., Qu, Z., Pedreschi, D., and Giannotti, F. (2013). Understanding the patterns of car travel. *The European Physical Journal Special Topics*, 215(1):61–73.

[Pappalardo et al., 2016a] Pappalardo, L., Rinzivillo, S., and Simini, F. (2016a). Human mobility modelling: Exploration and preferential return meet the gravity model. *Procedia Computer Science*, 83:934 – 939. The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops.

[Pappalardo and Simini, 2018] Pappalardo, L. and Simini, F. (2018). Data-driven generation of spatio-temporal routines in human mobility. *Data Mining and Knowledge Discovery*, 32(3):787–829.

[Pappalardo et al., 2015] Pappalardo, L., Simini, F., Rinzivillo, S., Pedreschi, D., Giannotti, F., and Barabási, A.-L. (2015). Returners and explorers dichotomy in human mobility. *Nature communications*, 6:8166.

[Pappalardo et al., 2016b] Pappalardo, L., Vanhoof, M., Gabrielli, L., Smoreda, Z., Pedreschi, D., and Giannotti, F. (2016b). An analytical framework to nowcast well-being using mobile phone data. *International Journal of Data Science and Analytics*, 2(1):75–92.

[Pebesma, 2018] Pebesma, E. (2018). Cran task view: Handling and analyzing spatio-temporal data.

[Pellungrini et al., 2017] Pellungrini, R., Pappalardo, L., Pratesi, F., and Monreale, A. (2017). A data mining approach to assess privacy risk in human mobility data. *ACM Trans. Intell. Syst. Technol.*, 9(3):31:1–31:27.

[Pratesi et al., 2018] Pratesi, F., Monreale, A., Trasarti, R., Giannotti, F., Pedreschi, D., and Yanagihara, T. (2018). Prudence: a system for assessing privacy risk vs utility in data sharing ecosystems. *Transactions on Data Privacy*, 11(2):139–167.

[Ramos-Fernández et al., 2004] Ramos-Fernández, G., Mateos, J. L., Miramontes, O., Cocho, G., Larralde, H., and Ayala-Orozco, B. (2004). Lévy walk patterns in the foraging movements of spider monkeys. *Behavioral ecology and Sociobiology*, 55(3):223–230.

[Rey and Anselin, 2007] Rey, S. J. and Anselin, L. (2007). PySAL: A Python Library of Spatial Analytical Methods. *The Review of Regional Studies*, 37(1):5–27.

[Rossi et al., 2019] Rossi, A., Barlacchi, G., Bianchini, M., and Lepri, B. (2019). Modelling taxi drivers' behaviour for the next destination prediction. *IEEE Transactions on Intelligent Transportation Systems*.

[Rossi et al., 2018] Rossi, A., Pappalardo, L., Cintia, P., Iaia, M., Fernández, J., and Medina, D. (2018). Effective injury prediction in professional soccer with gps data and machine learning. *PLOS ONE*, 13:1–15.

[Simini et al., 2012] Simini, F., González, M. C., Maritan, A., and Barabási, A.-L. (2012). A universal model for mobility and migration patterns. *Nature*, 484:96–100.

[Song et al., 2010a] Song, C., Koren, T., Wang, P., and Barabási, A. (2010a). Modelling the scaling properties of human mobility. *Nature Physics*, 6(10):818–823.

[Song et al., 2010b] Song, C., Qu, Z., Blumm, N., and Barabási, A.-L. (2010b). Limits of predictability in human mobility. *Science*, 327(5968):1018–1021.

[Tizzoni et al., 2012] Tizzoni, M., Bajardi, P., Poletto, C., Ramasco, J. J., Balcan, D., Gonçalves, B., Perra, N., Colizza, V., and Vespignani, A. (2012). Real-time numerical forecast of global epidemic spreading: case study of 2009 a/h1n1pdm. *BMC Medicine*, 10(1).

[Wang et al., 2011] Wang, D., Pedreschi, D., Song, C., Giannotti, F., and Barabasi, A.-L. (2011). Human mobility, social ties, and link prediction. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1100–1108, New York, NY, USA. ACM.

[Wilson, 1971] Wilson, A. G. (1971). A family of spatial interaction models, and associated developments. *Environment and Planning A*, 3(1):1–32.

[Wu et al., 2018] Wu, R., Luo, G., Shao, J., Tian, L., and Peng, C. (2018). Location prediction on trajectory data: A review. *Big Data Mining and Analytics*, 1(2):108–127.

[Zhang et al., 2017] Zhang, J., Zheng, Y., and Qi, D. (2017). Deep spatio-temporal residual networks for citywide crowd flows prediction. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[Zhao et al., 2016] Zhao, K., Tarkoma, S., Liu, S., and Vo, H. (2016). Urban human mobility data mining: An overview. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1911–1920.

[Zheng, 2015] Zheng, Y. (2015). Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):29.

[Zheng et al., 2014] Zheng, Y., Capra, L., Wolfson, O., and Yang, H. (2014). Urban computing: Concepts, methodologies, and applications. *ACM Transactions on Intelligent System Technologies*, 5(3):38:1–38:55.

[Zheng et al., 2008] Zheng, Y., Wang, L., Zhang, R., Xie, X., and Ma, W.-Y. (2008). Geolife: Managing and understanding your past life over maps. In *Proceedings of the The Ninth International Conference on Mobile Data Management*, MDM '08, pages 211–212, Washington, DC, USA. IEEE Computer Society.

[Zipf, 1946] Zipf, G. K. (1946). The p 1 p 2/d hypothesis: on the intercity movement of persons. *American sociological review*, 11(6):677–686.